# Biotools Cookbook

Martin Asser Hansen

March 18, 2008

John Mattick Group
Institute for Molecular Bioscience
University of Queensland
Australia
E-mail: mail@maasha.dk

# Contents

## List of Figures

# 1. Introduction

Biotools is a selection of simple tools that can be linked together (piped as we shall call it) in a very flexible manner to perform both simple and complex tasks. The fundamental idea is that biotools work on a data stream that will only terminate at the end of an analysis and that this data stream can be passed through several different biotools, each performing one specific task. The advantage of this approach is that a user can perform simple and complex tasks without having to write advanced code. Moreover, since the data format used to pass data between biotools is text based, biotools can be written by different developers in their favorite programming language — and still the biotools will be able to work together.

In the most simple form bioools can be piped together on the command line like this (using the pipe character '|'):

```
read_data | calculate_something | write_result
```

However, a more comprehensive analysis could be composed:

```
read_data | select_entries | convert_entries | search_database
evaluate_results | plot_diagram | plot_another_diagram |
load_to_database
```

The data stream that is piped through the biotools consists of records of key/value pairs in the same way a hash does in order to keep as simple a structure as possible. An example record can be seen below:

```
REC_TYPE: PATSCAN
MATCH: AGATCAAGTG
S_BEG: 7
S_END: 16
ALIGN_LEN: 9
S_ID: piR-t6
STRAND: +
PATTERN: AGATCAAGTG
---
```

The '---' denotes the delimiter of the records, and each key is a word followed by a ':' and a white-space and then the value. By convention the biotools only uses upper case keys (a list of used keys can be seen in Appendix A). Since the records basically are hash structures this mean that the order of the keys in the stream is unordered, and in the above example it is pure coincidence that HIT_BEG is displayed before HIT_END, however, when the order of the keys is importent, the biotools will automagically see to that.

All of the biotools are able to read and write a data stream to and from file as long as the records are in the biotools format. This means that if you are undertaking a lengthy

analysis where one of the steps is time consuming, you may save the stream after this step, and subsequently start one or more analysis from that last step[1]. If you are running a lengthy analysis it is highly recommended that you create a small test sample of the data and run that through the pipeline — and once you are satisfied with the result proceed with the full data set (see 7.1).

All of the biotools can be supplied with long arguments prefixed with -- switches or single character arguments prefixed with - switches that can be grouped together (e.g. -xok). In this cookbook only the long switches are used to emphasize what these switches do.

## 2. Setup

In order to get the biotools to work, you need to add environment settings to include the code binaries, scripts, and modules that constitute the biotools package. Assuming that you are using bash, add the following to your '~/.bashrc' file using your favorite editor. After the changes has been saved you need to either run 'source ~/.bashrc' or relogin.

```
if [ -f "/home/m.hansen/maasha/conf/bashrc" ]; then
    source "/home/m.hansen/maasha/conf/bashrc"
fi
```

## 3. Getting Started

The biotool **list_biotools** lists all the biotools along with a description:

```
list_biotools
align_seq             Align sequences in stream using Muscle.
analyze_seq           Analysis the residue composition of each sequence in stream.
analyze_vals          Determine type, count, min, max, sum and mean for values in stream.
blast_seq             BLAST sequences in stream against a specified database.
blat_seq              BLAT sequences in stream against a specified genome.
complement_seq        Complement sequences in stream.
count_records         Count the number of records in stream.
count_seq             Count sequences in stream.
count_vals            Count the number of times values of given keys exists in stream.
create_blast_db       Create a BLAST database from sequences in stream for use with BLAST.
...
```

To list the biotools for writing different formats, you can use unix's grep like this:

```
list_biotools | grep write
write_align           Write aligned sequences in pretty alignment format.
write_bed             Write records from stream as BED lines.
write_blast           Write BLAST records from stream in BLAST tabular format (-m8 and 9).
write_fasta           Write sequences in FASTA format.
write_psl             Write records from stream in PSL format.
write_tab             Write records from stream as tab separated table.
```

[1]It is a goal that the biotools at some point will be able to dump the data stream to file in case one of the tools fail critically.

In order to find out how a specific biotool works, you just type the program name without any arguments and press return and the usage of the biotool will be displayed. E.g. **read_fasta** <return>:

```
Program name:   read_fasta
Author:         Martin Asser Hansen - Copyright (C) - All rights reserved
Contact:        mail@maasha.dk
Date:           August 2007
License:        GNU General Public License version 2 (http://www.gnu.org/copyleft/gpl.html)
Description:    Read FASTA entries.
Usage:          read_fasta [options] -i <FASTA file(s)>
Options:
   [-i <file(s)> | --data_in=<file(s)>]  -  Comma separated list of files to read.
   [-n <int>     | --num=<int>]          -  Limit number of records to read.
   [-I <file>    | --stream_in=<file>]   -  Read input stream from file  -  Default=STDIN
   [-O <file>    | --stream_out=<file>]  -  Write output stream to file  -  Default=STDOUT
Examples:
   read_fasta -i test.fna             -  Read FASTA entries from file.
   read_fasta -i test1.fna,test2,fna  -  Read FASTA entries from files.
   read_fasta -i '*.fna'              -  Read FASTA entries from files.
   read_fasta -i test.fna -n 10       -  Read first 10 FASTA entries from file.
```

# 4. The Data Stream

## 4.1. How to read the data stream from file?

You want to read a data stream that you previously have saved to file in biotools format. This can be done implicetly or explicitly. The implicit way uses the 'stdout' stream of the Unix terminal:

```
cat | <biotool>
```

cat is the Unix command that reads a file and output the result to 'stdout' — which in this case is piped to any biotool represented by the <biotool>. It is also possible to read the data stream using '<' to direct the 'stdout' stream into the biotool like this:

```
<biotool> < <file>
```

However, that will not work if you pipe more biotools together. Then it is much safer to read the stream from a file explicitly like this:

```
<biotool> --stream_in=<file>
```

Here the filename <file> is explicetly given to the biotool <biotool> with the switch --stream_in. This switch works with all biotools. It is also possible to read in data from multiple sources by repeating the explicit read step:

```
<biotool> --stream_in=<file1> | <biotool> --stream_in=<file2>
```

## 4.2. How to write the data stream to file?

In order to save the output stream from a biotool to file, so you can read in the stream again at a later time, you can do one of two things:

```
<biotool> > <file>
```

All, the biotools write the data stream to 'stdout' by default which can be written to a file by redirecting 'stdout' to file using '>' , however, if one of the biotools for writing other formats is used then the both the biotools records as well as the result output will go to 'stdout' in a mixture causing havock! To avoid this you must use the switch --stream_out that explictly tells the biotool to write the output stream to file:

```
<biotool> --stream_out=<file>
```

The --stream_out switch works with all biotools.

## 4.3. How to terminate the data stream?

The data stream is never stops unless the user want to save the stream or by supplying the --no_stream switch that will terminate the stream:

```
<biotool> --no_stream
```

The --no_stream switch only works with those biotools where it makes sense that the user might want to terminale the data stream, *i.e.* after an analysis step where the user wants to output the result, but not the data stream.

## 4.4. How to write my results to file?

Saving the result of an analysis to file can be done implicitly or explicitly. The implicit way:

```
<biotool> --no_stream > <file>
```

If you use '>' to redirect 'stdout' to file then it is important to use the --no_stream switch to avoid writing a mix of biotools records and result to the same file causing havock. The safe way is to use the --result_out switch which explicetly tells the biotool to write the result to a given file:

```
<biotool> --result_out=<file>
```

Using the above method will not terminate the stream, so it is possible to pipe that into another biotool generating different results:

```
<biotool1> --result_out=<file1> | <biotool2> --result_out=<file2>
```

And still the data stream will continue unless terminated with --no_stream:

```
<biotool> --result_out=<file> --no_stream
```

Or written to file using implicitly or explicity (4.4). The explicit way:

```
<biotool> --result_out=<file1> --stream_out=<file2>
```

### 4.5. How to read data from multiple sources?

To read multiple data sources, with the same type or different type of data do:

```
<biotool1> --data_in=<file1> | <biotool2> --data_in=<file2>
```

where type is the data type a specific biotool reads.

## 5. Reading input

### 5.1. How to read biotools input?

See (4.1).

### 5.2. How to read in data?

Data in different formats can be read with the appropriate biotool for that format. The biotools are typicalled named 'read_<data type>' such as **read_fasta**, **read_bed**, **read_tab**, etc., and all behave in a similar manner. Data can be read by supplying the --data_in switch and a file name to the file containing the data:

```
<biotool> --data_in=<file>
```

It is also possible to read in a saved biotools stream (see 4.1) as well as reading data in one go:

```
<biotool> --stream_in=<file1> --data_in=<file2>
```

If you want to read data from several files you can do this:

```
<biotool> --data_in=<file1> | <biotool> --data_in=<file2>
```

If you have several data files you can read in all explicitly with a comma separated list:

```
<biotool> --data_in=file1,file2,file3
```

And it is also possible to use file globbing[2]:

```
<biotool> --data_in=*.fna
```

Or in a combination:

```
<biotool> --data_in=file1,/dir/*.fna
```

Finally, it is possible to read in data in different formats using the appropriate biotool for each format:

```
<biotool1> --data_in=<file1> | <biotool2> --data_in=<file2> ...
```

## 5.3. How to read FASTA input?

Sequences in FASTA format can be read explicitly using **read_fasta**:

```
read_fasta --data_in=<file>
```

## 5.4. How to read alignment input?

If your alignment if FASTA formatted then you can **read_align**. It is also possible to use **read_fasta** since the data is FASTA formatted, however, with **read_fasta** the key ALIGN will be omitted. The ALIGN key is used to determine which sequences belong to what alignment which is required for **write_align**.

```
read_align --data_in=<file>
```

[2] using the short option will only work if you quote the argument -i '*.fna'

## 5.5. How to read tabular input?

Tabular input can be read with **read_tab** which will read in all rows and chosen columns (separated by a given delimter) from a table in text format.

The table below:

| | | |
|---|---|---|
| Human | ATACGTCAG | 23524 |
| Dog | AGCATGAC | 2442 |
| Mouse | GACTG | 234 |
| Cat | AAATGCA | 2342 |

Can be read using the command:

```
read_tab --data_in=<file>
```

Which will result in four records, one for each row, where the keys V0, V1, V2 are the default keys for the organism, sequence, and count, respectively. It is possible to select a subset of colums to read by using the --cols switch which takes a comma separated list of columns numbers (first column is designated 0) as argument. So to read in only the sequence and the count so that the count comes before the sequence do:

```
read_tab --data_in=<file> --cols=2,1
```

It is also possible to name the columns with the --keys switch:

```
read_tab --data_in=<file> --cols=2,1 --keys=COUNT,SEQ
```

## 5.6. How to read BED input?

The BED (Browser Extensible Data[3]) format is a tabular format for data pertaining to one of the Eukaryotic genomes in the UCSC genome brower[4]. The BED format consists of up to 12 columns, where the first three are mandatory CHR, CHR_BEG, and CHR_END. The mandatory columns and any of the optional columns can all be read in easily with the **read_bed** biotool.

```
read_bed --data_in=<file>
```

It is also possible to read the BED file with **read_tab** (see 5.5), however, that will be more cumbersome because you need to specify the keys:

```
read_tab --data_in=<file> --keys=CHR,CHR_BEG,CHR_END ...
```

[3] http://genome.ucsc.edu/FAQ/FAQformat
[4] http://genome.ucsc.edu/

### 5.7. How to read PSL input?

The PSL format is the output from BLAT and contains 21 mandatory fields that can be read with **read_psl**:

```
read_psl --data_in=<file>
```

## 6. Writing output

All result output can be written explicitly to file using the --result_out switch which all result generating biotools have. It is also possible to write the result to file implicetly by directing 'stdout' to file using '>', however, that requires the --no_stream swich to prevent a mixture of data stream and results in the file. The explicit (and safe) way:

```
... | <biotool> --result_out=<file>
```

The implicit way:

```
... | <biotool> --no_stream > <file>
```

### 6.1. How to write biotools output?

See (4.2).

### 6.2. How to write FASTA output?

FASTA output can be written with **write_fasta**.

```
... | write_fasta --result_out=<file>
```

It is also possible to wrap the sequences to a given width using the --wrap switch allthough wrapping of sequence is generally an evil thing:

```
... | write_fasta --no_stream --wrap=80
```

## 6.3. How to write alignment output?

Pretty alignments with ruler[5] and consensus sequence can be created with **write_align**, what also have the optional --wrap switch to break the alignment into blocks of a given width:

```
... | write_align --result_out=<file> --wrap=80
```

If the number of sequnces in the alignment is 2 then a pairwise alignment will be output otherwise a multiple alignment. And if the sequence type, determined automagically, is protein, then residues and symbols (+, :, .) will be used to show consensus according to the Blosum62 matrix.

## 6.4. How to write tabular output?

Outputting the data stream as a table can be done with **write_tab**, which will write generate one row per record with the values as columns. If you supply the optional --comment switch, when the first row in the table will be a 'comment' line prefixed with a '#':

```
... | write_tab --result_out=<file> --comment
```

You can also change the delimiter from the default (tab) to *e.g.* ',':

```
... | write_tab --result_out=<file> --delimit=','
```

If you want the values output in a specific order you have to supply a comma separated list using the --keys switch that will print only those keys in that order:

```
... | write_tab --result_out=<file> --keys=SEQ_NAME,COUNT
```

Alternatively, if you have some keys that you don't want in the tabular output, use the --no_keys switch. So to print all keys except SEQ and SEQ_TYPE do:

```
... | write_tab --result_out=<file> --no_keys=SEQ,SEQ_TYPE
```

Finally, if you have a stream containing a mix of different records types, *e.g.* records with sequences and records with matches, then you can use **write_tab** to output all the records in tabluar format, however, the --comment, --keys, and --no_keys switches will only respond to records of the first type encountered. The reason is that outputting mixed records is probably not what you want anyway, and you should remove all the unwanted records from the stream before outputting the table: **grab** is your friend (see 7.5).

[5] '.' for every 10 residues, ':' for every 50, and '|' for every 100

### 6.5. How to write a BED output?

Data in BED format can be output if the records contain the mandatory keys CHR, CHR_BEG, and CHR_END using **write_bed**. If the optional keys are also present, they will be output as well:

```
write_bed --result_out=<file>
```

### 6.6. How to write PSL output?

Data in PSL format can be output using **write_psl:**

```
write_psl --result_out=<file>
```

## 7. Manipulating Records

### 7.1. How to select a few records?

To quickly get an overview of your data you can limit the data stream to show a few records. This also very useful to test the pipeline with a few records if you are setting up a complex analysis using several biotools. That way you can inspect that all goes well before analyzing and waiting for the full data set. All of the read_<type> biotools have the --num switch which will take a number as argument and only that number of records will be read. So to read in the first 10 FASTA entries from a file:

```
read_fasta --data_in=test.fna --num=10
```

Another way of doing this is to use **head_records** will limit the stream to show the first 10 records (default):

```
... | head_records
```

Using **head_records** directly after one of the read_<type> biotools will be a lot slower than using the --num switch with the read_<type> biotools, however, **head_records** can also be used to limit the output from all the other biotools. It is also possible to give **head_records** a number of records to show using the --num switch. So to display the first 100 records do:

```
... | head_records --num=100
```

## 7.2. How to select random records?

If you want to inspect a number of random records from the stream this can be done with the **random_records** biotool. So if you have 1 mio records in the stream and you want to select 1000 random records do:

```
... | random_records --num=1000
```

## 7.3. How to count all records in the data stream?

To count all the records in the data stream use **count_records**, which adds one record (which is not included in the count) to the data stream. So to count the number of sequences in a FASTA file you can do this:

```
cat test.fna | read_fasta | count_records --no_stream
```

Which will write the last record containing the count to 'stdout':

```
count_records: 630
---
```

It is also possible to write the count to file using the --result_out switch.

## 7.4. How to get the length of record values?

Use the **length_vals** biotool to get the length of each value for a comma separated list of keys:

```
... | length_vals --keys=HIT,PATTERN
```

## 7.5. How to grab specific records?

The biotool **grab** is related to the Unix grep and locates records based on matching keys and/or values using either a pattern, a Perl regex, or a numerical evaluation. To easily **grab** all records in the stream that has any mentioning of the pattern 'human' just pipe the data stream through **grab** like this:

```
... | grab --pattern=human
```

This will search for the pattern 'human' in all keys and all values. The --pattern switch takes a comma separated list of patterns, so in order to match multiple patterns do:

```
... | grab --pattern=human,mouse
```

It is also possible to use the --pattern_in switch instead of --pattern. --pattern_in is used to read a file with one pattern per line:

```
... | grab --pattern_in=patterns.txt
```

If you want the opposite result — to find all records that does not match the patterns, add the --invert switch, which not only works with the --pattern switch, but also with --regex and --eval:

```
... | grab --pattern=human --invert
```

If you want to search the record keys only, *e.g.* to find all records containing the key SEQ you can add the --keys_only switch. This will prevent matching of SEQ in any record value, and in fact SEQ is a not uncommon peptide sequence you could get an unwanted record. Also, this will give an increase in speed since only the keys are searched:

```
... | grab --pattern=SEQ --keys_only
```

However, if you are interested in finding the peptide sequence SEQ and not the SEQ key, just add the --vals_only switch instead:

```
... | grab --pattern=SEQ --vals_only
```

Also, if you want to grab for certain key/value pairs you can supply a comma separated list of keys whos values will then be searched using the --keys switch. This is handy if your records contain large genomic sequences and you dont want to search the entire sequence for *e.g.* the organism name — it is much faster to tell **grab** which keys to search the value for:

```
... | grab --pattern=human --keys=SEQ_NAME
```

It is also possible to invoke flexible matching using regex (regular expressions) instead of simple pattern matching. In **grab** the regex engine is Perl based and allows use of different type of wild cards, alternatives, *etc*[6]. If you want to **grab** records withs the sequence ATCG or GCTA you can do this:

```
... | grab --regex='ATCG|GCTA'
```

[6] http://perldoc.perl.org/perlreref.html

Or if you want to find sequences beginning with ATCG:

```
... | grab --regex='^ATCG'
```

You can also use **grab** to locate records that fulfill a numerical property using the --eval switch witch takes an expression in three parts. The first part is the key that holds the value we want to evaluate, the second part holds one the six operators:

1. Greater than: >
2. Greater than or equal to: >=
3. Less than: <
4. Less than or equal to: <=
5. Equal to: =
6. Not equal to: !=
7. String wise equal to: eq
8. String wise not equal to: ne

And finally comes the number used in the evaluation. So to **grab** all records with a sequence length greater than 30:

```
... length_seq | grab --eval='SEQ_LEN > 30'
```

If you want to locate all records containing the pattern 'human' and where the sequence length is greater that 30, you do this by running the stream through **grab** twice:

```
... | grab --pattern='human' | length_seq | grab --eval='SEQ_LEN > 30'
```

Finally, it is possible to do fast matching of expressions from a file using the --exact switch. Each of these expressions has to be matched exactly over the entrie length, which if useful if you have a file with accession numbers, that you want to locate in the stream:

```
... | grab --exact acc_no.txt | ...
```

Using --exact is much faster than using --pattern_in, because with --exact the expression has to be complete matches, where --pattern_in looks for subpatterns.

NB! To get the best speed performance, use the most restrictive **grab** first.

### 7.6. How to remove keys from records?

To remove one or more specific keys from all records in the data stream use **remove_keys** like this:

```
... | remove_keys --keys=SEQ,SEQ_NAME
```

In the above example SEQ and SEQ_NAME will be removed from all records if they exists in these. If all keys are removed from a record, then the record will be removed.

### 7.7. How to rename keys in records?

Sometimes you want to rename a record key, *e.g.* if you have read in a two column table with sequence name and sequence in each column (see 5.5) without specifying the key names, then the sequence name will be called V0 and the sequence V1 as default in the **read_tab** biotool. To rename the V0 and V1 keys we need to run the stream through **rename_keys** twice (one for each key to rename):

```
... | rename_keys --keys=V0,SEQ_NAME | rename_keys --keys=V1,SEQ
```

The first instance of **rename_keys** replaces all the V0 keys with SEQ_NAME, and the second instance of **rename_keys** replaces all the V1 keys with SEQ. *Et viola* the data can now be used in the biotools that requires these keys.

## 8. Manipulating Sequences

### 8.1. How to get sequence lengths?

The length for sequences in records can be determined with **length_seq**, which adds the key SEQ_LEN to each record with the sequence length as the value. It also generates an extra record that is emitted last with the key TOTAL_SEQ_LEN showing the total length of all the sequences.

```
read_fasta --data_in=<file> | length_seq
```

It is also possible to determine the sequence length using the generic tool **length_vals** (7.4), which determines the length of the values for a given list of keys:

```
read_fasta --data_in=<file> | length_vals --keys=SEQ
```

To obtain the total length of all sequences use **sum_vals** like this:

```
read_fasta --data_in=<file> | length_vals --keys=SEQ
| sum_vals --keys=SEQ_LEN
```

The biotool **analyze_seq** will also determine the length of each sequence (see 8.2).

## 8.2. How to analyze sequence composition?

If you want to find out the sequence type, composition, length, as well as GC content, indel content and proportions of soft and hard masked sequence, then use **analyze_seq**. This handy biotool will determine all these things per sequence from which it is easy to get an overview using the **write_tab** biotool to output a table (see 6.4). So in order to determine the sequence composition of a FASTA file with just one entry containing the sequence 'ATCG' we just read the data with **read_fasta** and run the output through **analyze_seq** which will add the analysis to the record like this:

```
read_fasta --data_in=test.fna | analyze_seq ...
RES:D: 0
MIX_INDEX: 0.55
RES:W: 0
RES:G: 16
SOFT_MASK%: 63.75
RES:B: 0
RES:V: 0
HARD_MASK%: 0.00
RES:H: 0
RES:S: 0
RES:N: 0
RES:.: 0
GC%: 35.00
RES:A: 8
RES:Y: 0
RES:M: 0
RES:T: 44
SEQ_TYPE: DNA
RES:K: 0
RES:~: 0
SEQ: TTTCAGTTTGGGACGGAGTAAGGCCTTCCtttttttttttttttttttttttttttttgagaccgagtcttgctctgtcg
SEQ_LEN:
80 RES:R: 0
RES:C: 12
RES:-: 0
RES:U: 0
---
```

Now to make a table of how may As, Ts, Cs, and Gs you can add the following:

```
... | analyze_seq | write_tab --keys=RES:A,RES:T,RES:C,RES:G
```

Or if you want to see the proportions of hard and soft masked sequence:

```
... | analyse_seq | write_tab --keys=HARD_MASK%,SOFT_MASK%
```

If you have a stack of sequences in one file and you want to determine the mean GC content you can do it using the **mean_vals** biotool:

```
read_fasta --data_in=test.fna | analyze_seq | mean_vals --keys=GC%
```

Or if you want the total count of Ns you can use **sum_vals** like this:

```
read_fasta --data_in=test.fna | analyze_seq | sum_vals --keys=RES:N
```

The MIX_INDEX key is calculated as the count of the most common residue over the sequence length, and can be used as a cut-off for removing sequence tags consisting of mostly one nucleotide:

```
read_fasta --data_in=test.fna | analyze_seq | grab --eval='MIX_INDEX<0.85'
```

## 8.3. How to extract subsequences?

In order to extract a subsequence from a longer sequence use the biotool extract_seq, which will replace the sequence in the record with the subsequence (this behaviour should probably be modified to be dependant of a --replace or a --no_replace switch). So to extract the first 20 residues from all sequences do (first residue is designated 1):

```
... | extract_seq --beg=1 --len=20
```

You can also specify a begin and end coordinate set:

```
... | extract_seq --beg=20 --end=40
```

If you want the subsequences from position 20 to the sequence end do:

```
... | extract_seq --beg=20
```

If you want to extract subsequences a given distance from the sequence end you can do this by reversing the sequence with the biotool **reverse_seq** (8.6), followed by **extract_seq** to get the subsequence, and then **reverse_seq** again to get the subsequence back in the original orientation:

```
read_fasta --data_in=test.fna | reverse_seq
| extract_seq --beg=10 --len=10 | reverse_seq
```

## 8.4. How to get genomic sequence?

The biotool **get_genomic_seq** can extract subsequences for a given genome specified with the --genome switch explicitly using the --beg and --end/--len switches:

```
get_genome_seq --genome=<genome> --beg=1 --len=100
```

Alternatively, **get_genome_seq** can be used to append the corresponding sequence to BED, PSL, and BLAST records:

```
read_bed --data_in=<BED file> | get_genome_seq --genome=<genome>
```

It is also possible to include flaking sequence using the --flank switch. So to include 50 nucleotides upstream and 50 nucleotides downstream for each BED entry do:

```
read_bed --data_in=<BED file> | get_genome_seq --genome=<genome> --flank=50
```

## 8.5. How to upper-case sequences?

Sequences can be shifted from lower case to upper case using **uppercase_seq**:

```
... | uppercase_seq
```

## 8.6. How to reverse sequences?

The order of residues in a sequence can be reversed using reverse_seq:

```
... | reverse_seq
```

Note that in order to reverse/complement a sequence you also need the **complement_seq** biotool (see 8.7).

## 8.7. How to complement sequences?

DNA and RNA sequences can be complemented with **complement_seq**, which automagically determines the sequence type:

```
... | complement_seq
```

Note that in order to reverse/complement a sequence you also need the **reverse_seq** biotool (see 8.6).

## 8.8. How to remove indels from sequnces?

Indels can be removed from sequences with the **remove_indels** biotool. This is useful if you have aligned some sequences (see 8.18) and extracted (see 8.3) a block of subsequences from the alignment and you want to use these sequence in a search where you need to remove the indels first. '-', '~', and '.' are considered indels:

```
... | remove_indels
```

## 8.9. How to shuffle sequences?

All residues in sequences in the stream can be shuffled to random positions using the **shuffle_seq** biotool:

```
... | shuffle_seq
```

## 8.10. How to split sequences into overlapping subsequences?

Sequences can be slit into overlapping subsequences with the **split_seq** biotool.

```
... | split_seq --word_size=20 --uniq
```

## 8.11. How to determine the oligo frequency?

In order to determine if any oligo usage is over represented in one or more sequences you can determine the frequency of oligos of a given size with **oligo_freq**:

```
... | oligo_freq --word_size=4
```

And if you have more than one sequence and want to accumulate the frequences you need the --all switch:

```
... | oligo_freq --word_size=4 --all
```

To get a meaningful result you need to write the resulting frequencies as a table with **write_tab** (see 6.4), but first it is important to **grab** (see 7.5) the records with the frequencies to avoid full length sequences in the table:

```
... | oligo_freq --word_size=4 --all | grab --pattern=OLIGO --keys_only
| write_tab --no_stream
```

And the resulting frequency table can be sorted with Unix sort (man sort).

## 8.12. How to search for sequences in genomes?

See the following biotool:

- **patscan_seq** (8.13)
- **blat_seq** (8.14)
- **blast_seq** (8.15)
- **vmatch_seq** (8.16)

## 8.13. How to search sequences for a pattern?

It is possible to search sequences in the data stream for patterns using the **patscan_seq** biotool which utilizes the powerful scan_for_matches engine. Consult the documentation for scan_for_matches in order to learn how to define patterns (the documentation is included in Appendix C).

To search all sequences for a simple pattern consisting of the sequence ATCGATCG allowing for 3 mismatches, 2 insertions and 1 deletion:

```
read_fasta --data_in=<file> | patscan_seq --pattern='ATCGATCG[3,2,1]'
```

The --pattern switch takes a comma seperated list of patterns, so if you want to search for more that one pattern do:

```
... | patscan_seq --pattern='ATCGATCG[3,2,1],GCTAGCTA[3,2,1]'
```

It is also possible to have a list of different patterns to search for in a file with one pattern per line. In order to get **patscan_seq** to read these patterns use the --pattern_in switch:

```
... | patscan_seq --pattern_in=<file>
```

To also scan the complementary strand in nucleotide sequences (**patscan_seq** automagically determines the sequence type) you need to add the --comp switch:

```
... | patscan_seq --pattern=<pattern> --comp
```

It is also possible to use **patscan_seq** to output those records that does not contain a certain pattern by using the --invert switch:

```
... | patscan_seq --pattern=<pattern> --invert
```

Finally, **patscan_seq** can also scan for patterns in a given genome sequence, instead of sequences in the stream, using the --genome switch:

```
patscan --pattern=<pattern> --genome=<genome>
```

## 8.14. How to use BLAT for sequence search?

Sequences in the data stream can be matched against supported genomes using **blat_seq** which is a biotool using BLAT as the name might suggest. Currently only Mouse and Human genomes are available and it is not possible to use OOC files since there is still a need for a local repository for genome files. Otherwise it is just:

```
read_fasta --data_in=<file> | blat_seq --genome=<genome>
```

The search results can then be written to file with **write_psl** (see 6.6) or **write_bed** (see 6.5) allthough with **write_bed** some information will be lost). It is also possible to plot chromosome distribution of the search results using **plot_chrdist** (see 9.3) or the distribution of the match lengths using **plot_lendist** (see 9.2) or a karyogram with the hits using **plot_karyogram** (see 9.6).

## 8.15. How to use BLAST for sequence search?

Two biotools exist for blasting sequences: **create_blast_db** is used to create the BLAST database required for BLAST which is queried using the biotool **blast_seq**. So in order to create a BLAST database from sequences in the data stream you simple run:

```
... | create_blast_db --database=my_database --no_stream
```

The type of sequence to use for the database is automagically determined by **create_blast_db**, but don't have a mixture of peptide and nucleic acids sequences in the stream. The --database switch takes a path as argument, but will default to 'blastdb_<time_stamp> if not set.

The resulting database can now be queried with sequences in another data stream using **blast_seq**:

```
... | blast_seq --database=my_database
```

Again, the sequence type is determined automagically and the appropriate BLAST program is guessed (see below table), however, the program name can be overruled with the --program switch.

| Subject sequence | Query sequence | Program guess |
|---|---|---|
| Nucleotide | Nucleotide | blastn |
| Protein | Protein | blastp |
| Protein | Nucleotide | blastx |
| Nucleotide | Protein | tblastn |

Finally, it is also possible to use **blast_seq** for blasting sequences agains a preformatted genome using the --genome switch instead of the --database switch:

```
... | blast_seq --genome=<genome>
```

## 8.16. How to use Vmatch for sequence search?

The powerful suffix array software package Vmatch[7] can be used for exact mapping of sequences against indexed genomes using the biotool **vmatch_seq**, which will e.g. map 700000 ESTs to the human genome locating all 160 mio hits in less than an hour. Only nucleotide sequences and sequences longer than 11 nucleotides will be mapped. It is recommended that sequences consisting of mostly one nucleotide type are removed. This can be done with the **analyze_seq** biotool (8.2).

```
... | vmatch_seq --genome=<genome>
```

It is also possible to allow for mismatches using the --hamming_dist switch. So to allow for 2 mismatches:

```
... | vmatch_seq --genome=<genome> --hamming_dist=2
```

Or to allow for 10% mismathing nucleotides:

```
... | vmatch_seq --genome=<genome> --hamming_dist=10p
```

To allow both indels and mismatches use the --edit_dist switch. So to allow for one mismatch or one indel:

```
... | vmatch_seq --genome=<genome> --hamming_dist=1
```

Or to allow for 5% indels or mismatches:

```
... | vmatch_seq --genome=<genome> --hamming_dist=5p
```

Note that using --hamming_dist or--edit_dist greatly slows down vmatch considerably — use with care.

The resulting SCORE key can be replaced to hold the number of genome matches of a given sequence (multi-mappers) is the --count switch is given.

## 8.17. How to find all matches between sequences?

All matches between two sequences can be determined with the biotool **match_seq**. The match finding engine underneath the hood of **match_seq** is the super fast suffix tree program MUMmer[8], which will locate all forward and reverse matches between huge

[7] http://www.vmatch.de/
[8] http://mummer.sourceforge.net/

sequences in a matter of minutes (if the repeat count is not too high and if the word size used is appropriate). Matching two *Helicobacter pylori* genomes (1.7Mbp) takes around 10 seconds:

```
... | match_seq --word_size=20 --direction=both
```

The output from **match_seq** can be used to generate a dot plot with **plot_matches** (see 9.4).

## 8.18. How to align sequences?

Sequences in the stream can be aligned with the **align_seq** biotool that uses Muscle[9] as aligment engine. Currently you cannot change any of the Muscle alignment parameters and **align_seq** will create an alignment based on the defaults (which are really good!):

```
... | align_seq
```

The aligned output can be written to file in FASTA format using **write_fasta** (see 6.2) or in pretty text using **write_align** (see 6.3).

## 8.19. How to create a weight matrix?

If you want a weight matrix to show the sequence composition of a stack of sequences you can use the biotool create_weight_matrix:

```
... | create_weight_matrix
```

The result can be output in percent using the --percent switch:

```
... | create_weight_matrix --percent
```

The weight matrix can be written as tabular output with **write_tab** (see 6.4) after removeing the records containing SEQ with **grab** (see 7.5):

```
... | create_weight_matrix | grab --invert --keys=SEQ --keys_only
| write_tab --no_stream
```

The V0 column will hold the residue, while the rest of the columns will hold the frequencies for each sequence position.

[9] http://www.drive5.com/muscle/muscle.html

# 9. Plotting

There exists several biotools for plotting. Some of these are based on GNUplot[10], which is an extremely powerful platform to generate all sorts of plots and even though GNUplot has quite a steep learning curve, the biotools utilizing GNUplot are simple to use. GNUplot is able to output a lot of different formats (called terminals in GNUplot), but the biotools focusses on three formats only:

1. The 'dumb' terminal is default to the GNUplot based biotools and will output a plot in crude ASCII text (Fig. 1). This is quite nice for a quick and dirty plot to get an overview of your data .
2. The 'post' or 'postscript' terminal output postscript code which is publication grade graphics that can be viewed with applications such as Ghostview, Photoshop, and Preview.
3. The 'svg' terminal output's scalable vector graphics (SVG) which is a vector based format. SVG is great because you can edit the resulting plot using Photoshop or Inkscape[11] if you want to add additional labels, captions, arrows, and so on and then save the result in different formats, such as postscript without loosing resolution.

The biotools for plotting that are not based on GNUplot only output SVG (that may change in the future).

## 9.1. How to plot a histogram?

A generic histogram for a given value can be plotted with the biotool **plot_histogram** (Fig. 2):

```
... | plot_histogram --key=TISSUE --no_stream
```

(Figure missing)

[10] http://www.gnuplot.info/

[11] Inkscape is a really handy drawing program that is free and open source. Availble at http://www.inkscape.org

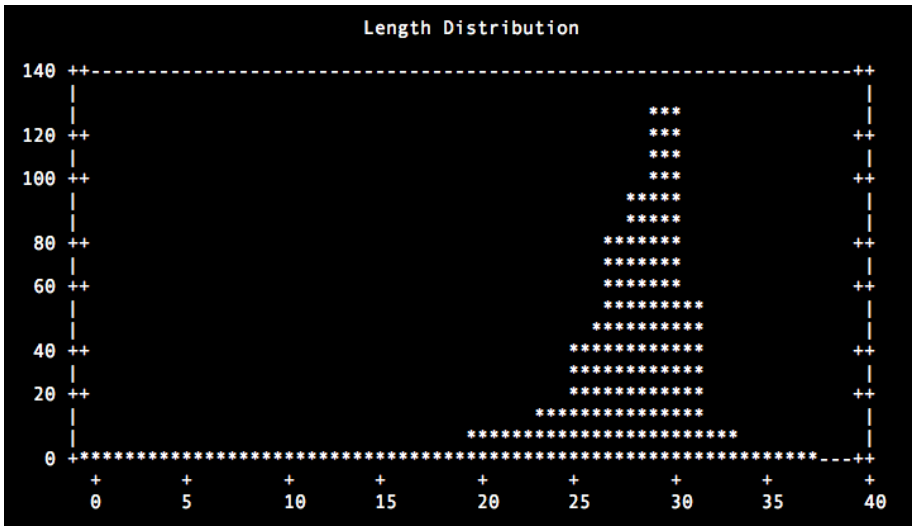


Figure 1: Dumb terminal

The output of a length distribution plot in the default 'dumb terminal' to the terminal window.

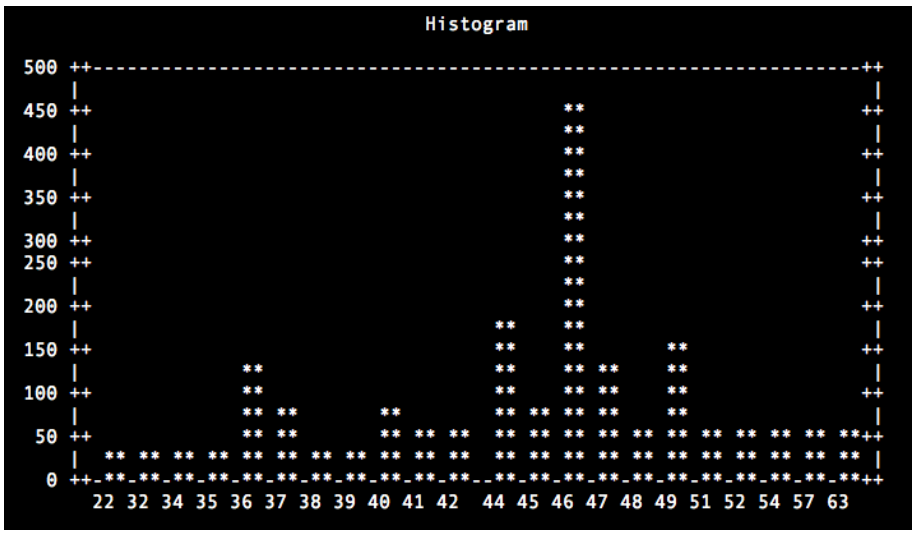


Figure 2: Histogram

## 9.2. How to plot a length distribution?

Plotting of length distributions, weather sequence lengths, patterns lengths, hit lengths, *etc.* is a really handy thing and can be done with the the biotool **plot_lendist**. If you have a file with FASTA entries and want to plot the length distribution you do it like this:

```
read_fasta --data_in=<file> | length_seq
| plot_lendist --key=SEQ_LEN --no_stream
```

The result will be written to the default dumb terminal and will look like Fig. 1.

If you instead want the result in postscript format you can do:

```
... | plot_lendist --key=SEQ_LEN --terminal=post --result_out=file.ps
```

That will generate the plot and save it to file, but not interrupt the data stream which can then be used in further analysis. You can also save the plot implicetly using '>', however, it is then important to terminate the stream with the --no_stream switch:

```
... | plot_lendist --key=SEQ_LEN --terminal=post --no_stream > file.ps
```

The resulting plot can be seen in Fig. 3.

## 9.3. How to plot a chromosome distribution?

If you have the result of a sequence search against a multi chromosome genome, it is very practical to be able to plot the distribution of search hits on the different chromosomes. This can be done with **plot_chrdist**:

```
read_fasta --data_in=<file> | blat_genome | plot_chrdist --no_stream
```

The above example will result in a crude plot using the 'dumb' terminal, and if you want to mess around with the results from the BLAT search you probably want to save the result to file first (see 6.6). To plot the chromosome distribution from the saved search result you can do:

```
read_bed --data_in=file.bed | plot_chrdist --terminal=post --result_out=plot.ps
```

That will result in the output show in Fig. 4.

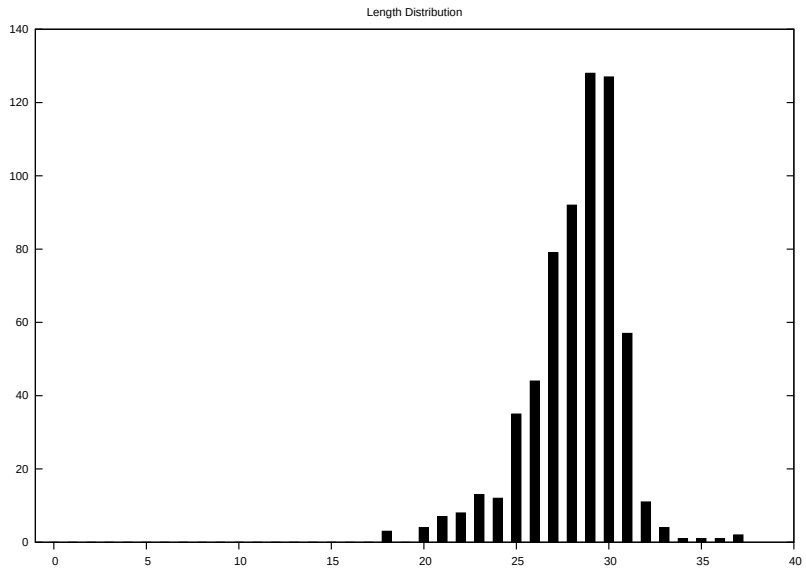


Figure 3: Length distribution

Length distribution of 630 piRNA like RNAs.

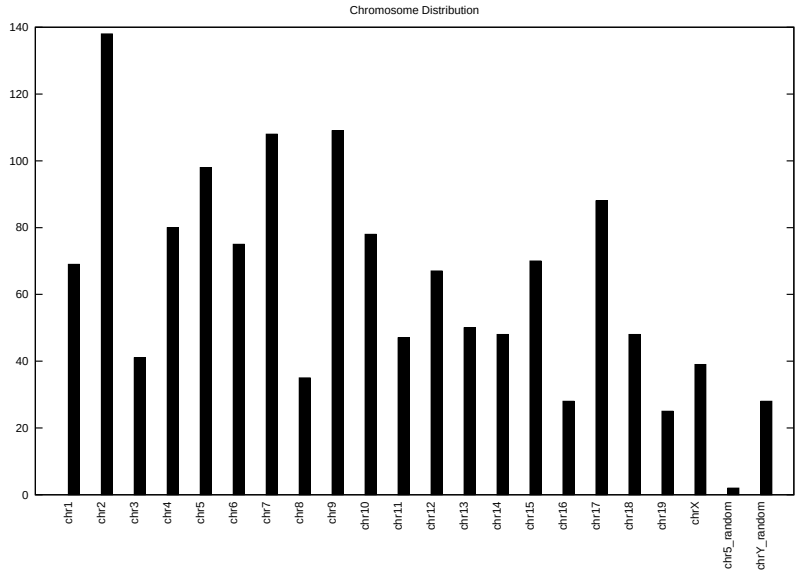


Figure 4: Chromosome distribution

## 9.4. How to generate a dotplot?

A dotplot is a powerful way to get an overview of the size and location of sequence insertions, deletions, and duplications between two sequences. Generating a dotplot with biotools is a two step process where you initially find all matches between two sequences using the tool **match_seq** (see 8.17) and plot the resulting matches with **plot_matches**. Matching and plotting two *Helicobacter pylori* genomes (1.7Mbp) takes around 10 seconds:

```
... | match_seq | plot_matches --terminal=post --result_out=plot.ps
```

The resulting dotplot is in Fig. 5.

## 9.5. How to plot a sequence logo?

Sequence logos can be generate with **plot_seqlogo**. The sequnce type is determined automagically and an entropy scale of 2 bits and 4 bits is used for nucleotide and peptide sequences, respectively[12].

[12] http://www.ccrnp.ncifcrf.gov/~toms/paper/hawaii/latex/node5.html

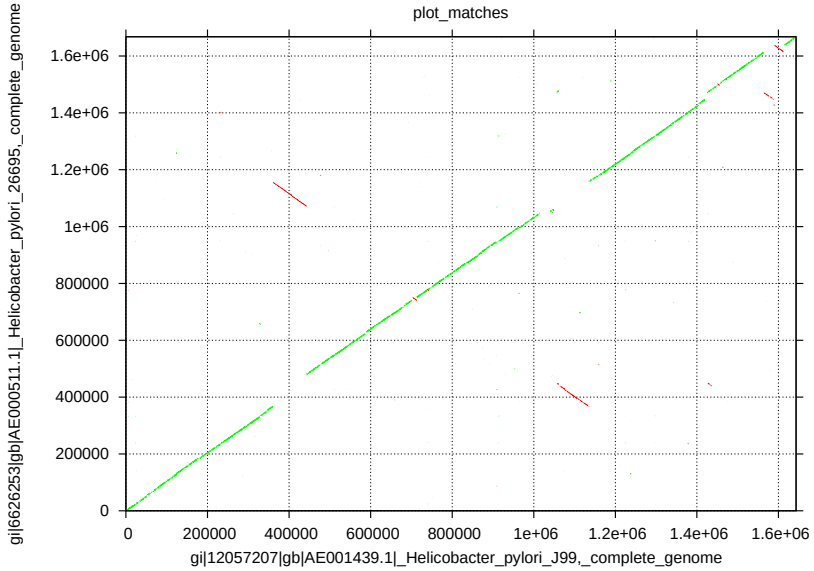


Figure 5: Dotplot

Forward matches are displayed in green while reverse matches are displayed in red.

Figure 6: Sequence logo

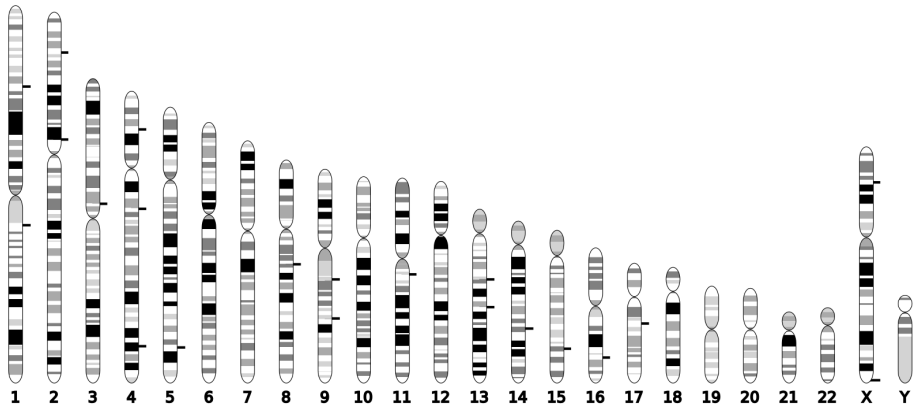


Figure 7: Karyogram

Hits from a search of piRNA like RNAs in the Human genome is displayed as short horizontal bars.

```
... | plot_seqlogo --no_stream --result_out=seqlogo.svg
```

An example of a sequence logo can be seen in Fig. 6.

### 9.6. How to plot a karyogram?

To plot search hits on genomes use **plot_karyogram**, which will output a nice karyogram in SVG graphics:

```
... | plot_karyogram --result_out=karyogram.svg
```

The banding data is taken from the UCSC genome browser database and currently only Human and Mouse is supported. Fig. 7 shows the distribution of piRNA like RNAs matched to the Human genome.

## 10. Uploading Results

### 10.1. How do I display my results in the UCSC Genome Browser?

Results from the list of biotools below can be uploaded directly to a local mirror of the UCSC Genome Browser using the biotool **upload_to_ucsc**:

- patscan_seq (8.13)
- blat_seq (8.14)
- blast_seq (8.15)
- vmatch_seq (8.16)

The syntax for uploading data the most simple way requires two mandatory switches: --database, which is the UCSC database name (such as hg18, mm9, etc.) and--table which should be the users initials followed by an underscore and a short description of the data:

```
... | upload_to_ucsc --database=hg18 --table=mah_snoRNAs
```

The **upload_to_ucsc** biotool modifies the users ~/ucsc/my_tracks.ra file automagically (a backup is created with the name ~/ucsc/my_tracks.ra~) with default values that can be overridden using the following switches:

- --short_label - Short label for track - Default=database->table
- --long_label - Long label for track - Default=database->table
- --group - Track group name - Default=<user name as defined in env>
- --priority - Track display priority - Default=1
- --color - Track color - Default=147,73,42
- --chunk_size - Chunks for loading - Default=10000000
- --visibility - Track visibility - Default=pack

Also, data in BED or PSL format can be uploaded with **upload_to_ucsc** as long as these reference to genomes and chromosomes existing in the UCSC Genome Browser:

```
read_bed --data_in=<bed file> | upload_to_ucsc ...
read_psl --data_in=<psl file> | upload_to_ucsc ...
```

# 11. Power Scripting

It is possible to do commandline scripting of biotool records using Perl. Because a biotool record essentially is a hash structure, you can pass records to **bioscript** command, which is a wrapper around the Perl executable that allows direct manipulations of the records using the power of Perl.

In the below example we replace in all records the value to the CHR key with a forthrunning number:

```
... | bioscript 'while($r=get_record(\*STDIN)){$r->{CHR}=$i++; put_record($r)}'
```

Something more useful would probably be to create custom FASTA headers. E.g. if we read in a BED file, lookup the genomic sequence, create a custom FASTA header with **bioscript** and output FASTA entries:

```
... | bioscript 'while($r=get_record(\*STDIN)){$r->{SEQ_NAME}= //
join("_",$r->{CHR},$r->{CHR_BEG},$r->{CHR_END}); put_record($r)}'
```

And the output:

```
>chr2L_21567527_21567550
taccaaacggatgcctcagacatc
>chr2L_693380_693403
taccaaacggatgcctcagacatc
>chr2L_13859534_13859557
taccaaacggatgcctcagacatc
>chr2L_9005090_9005113
taccaaacggatgcctcagacatc
>chr2L_2106825_2106848
taccaaacggatgcctcagacatc
>chr2L_14649031_14649054
taccaaacggatgcctcagacatc
```

# 12. Trouble shooting

Shoot the messenger!

## A. Keys

HIT

HIT_BEG

HIT_END

HIT_LEN

HIT_NAME

PATTERN

## B. Switches

--stream_in

--stream_out

--no_stream

--data_in

--result_out

--num

## C. scan_for_matches README

```
                        scan_for_matches:
    A Program to Scan Nucleotide or Protein Sequences for Matching Patterns
                        Ross Overbeek
                        MCS
                        Argonne National Laboratory
                        Argonne, IL 60439
                        USA

Scan_for_matches is a utility that we have written to search for
patterns in DNA and protein sequences.  I wrote most of the code,
although David Joerg and Morgan Price wrote sections of an
earlier version.  The whole notion of pattern matching has a rich
history, and we borrowed liberally from many sources.  However, it is
worth noting that we were strongly influenced by the elegant tools
developed and distributed by David Searls.  My intent is to make the
existing tool available to anyone in the research community that might
find it useful.  I will continue to try to fix bugs and make suggested
enhancements, at least until I feel that a superior tool exists.
```

```
Hence, I would appreciate it if all bug reports and suggestions are
directed to me at Overbeek@mcs.anl.gov.
I will try to log all bug fixes and report them to users that send me
their email addresses.  I do not require that you give me your name
and address.  However, if you do give it to me, I will try to notify
you of serious problems as they are discovered.
Getting Started:
    The distribution should contain at least the following programs:
                README                  -     This document
                ggpunit.c               -     One of the two source files
                scan_for_matches.c      -     The second source file

                run_tests               -     A perl script to test things
                show_hits               -     A handy perl script
                test_dna_input          -     Test sequences for DNA
                test_dna_patterns       -     Test patterns for DNA scan
                test_output             -     Desired output from test
                test_prot_input         -     Test protein sequences
                test_prot_patterns      -     Test patterns for proteins
                testit                  -     a perl script used for test
    Only the first three files are required.  The others are useful,
    but only if you have Perl installed on your system.  If you do
    have Perl, I suggest that you type

                which perl
    to find out where it installed.  On my system, I get the following
    response:

                clone% which perl
                /usr/local/bin/perl
    indicating that Perl is installed in /usr/local/bin.  Anyway, once
    you know where it is installed, edit the first line of files
        testit
        show_hits
    replacing /usr/local/bin/perl with the appropriate location.  I
    will assume that you can do this, although it is not critical (it
    is needed only to test the installation and to use the "show_hits"
    utility).  Perl is not required to actually install and run
    scan_for_matches.
    If you do not have Perl, I suggest you get it and install it (it
    is a wonderful utility).  Information about Perl and how to get it
    can be found in the book "Programming Perl" by Larry Wall and
    Randall L. Schwartz, published by O'Reilly & Associates, Inc.
    To get started, you will need to compile the program.   I do this
```

```
    using
        gcc -O -o scan_for_matches  ggpunit.c scan_for_matches.c
    If you do not use GNU C, use
        cc -O -DCC -o scan_for_matches  ggpunit.c scan_for_matches.c
    which works on my Sun.
    Once you have compiled scan_for_matches, you can verify that it
    works with
        clone% run_tests tmp
        clone% diff tmp test_output
    You may get a few strange lines of the sort
        clone% run_tests tmp
        rm: tmp: No such file or directory
        clone% diff tmp test_output
    These should cause no concern.  However, if the "diff" shows that
    tmp and test_output are different, contact me (you have a
    problem).
    You should now be able to use scan_for_matches by following the
    instructions given below (which is all the normal user should have
    to understand, once things are installed properly).
 ============================================================
How to run scan_for_matches:
    To run the program, you type need to create two files
    1.  the first file contains the pattern you wish to scan for; I'll
        call this file pat_file in what follows (but any name is ok)
    2.  the second file contains a set of sequences to scan.  These
        should be in "fasta format".  Just look at the contents of
        test_dna_input to see examples of this format.  Basically,
        each sequence begins with a line of the form
           >sequence_id
        and is followed by one or more lines containing the sequence.
    Once these files have been created, you just use
        scan_for_matches pat_file < input_file
    to scan all of the input sequences for the given pattern.  As an
    example, suppose that pat_file contains a single line of the form
                p1=4...7 3...8 ~p1
    Then,
                scan_for_matches pat_file < test_dna_input
    should produce two "hits".  When I run this on my machine, I get
        clone% scan_for_matches pat_file < test_dna_input
        >tst1:[6,27]
        cguaacc ggttaacc gguuacg
        >tst2:[6,27]
        CGUAACC GGTTAACC GGUUACG
        clone%
```

Simple Patterns Built by Matching Ranges and Reverse Complements

Let me first explain this simple pattern:

```
            p1=4...7 3...8 ~p1
```

The pattern consists of three "pattern units" separated by spaces. The first pattern unit is

```
            p1=4...7
```

which means "match 4 to 7 characters and call them p1". The second pattern unit is

```
            3...8
```

which means "then match 3 to 8 characters". The last pattern unit is

```
            ~p1
```

which means "match the reverse complement of p1". The first reported hit is shown as

```
    >tst1:[6,27]
    cguaacc ggttaacc gguuacg
```

which states that characters 6 through 27 of sequence tst1 were matched. "cguaac" matched the first pattern unit, "ggttaacc" the second, and "gguuacg" the third. This is an example of a common type of pattern used to search for sections of DNA or RNA that would fold into a hairpin loop.

Searching Both Strands

Now for a short aside: scan_for_matches only searched the sequences in the input file; it did not search the opposite strand. With a pattern of the sort we just used, there is not need o search the opposite strand. However, it is normally the case that you will wish to search both the sequence and the opposite strand (i.e., the reverse complement of the sequence). To do that, you would just use the "-c" command line. For example,

```
    scan_for_matches -c pat_file < test_dna_input
```

Hits on the opposite strand will show a beginning location greater than te end location of the match.

Defining Pairing Rules and Allowing Mismatches, Insertions, and Deletions

Let us stop now and ask "What additional features would one need to really find the kinds of loop structures that characterize tRNAs, rRNAs, and so forth?" I can immediately think of two:

a) you will need to be able to allow non-standard pairings (those other than G-C and A-U), and

b) you will need to be able to tolerate some number of mismatches and bulges.

Let me first show you how to handle non-standard "rules for pairing in reverse complements". Consider the following pattern,

which I show as two line (you may use as many lines as you like in forming a pattern, although you can only break a pattern at points where space would be legal):

```
        r1={au,ua,gc,cg,gu,ug,ga,ag}
        p1=2...3 0...4 p2=2...5 1...5 r1~p2 0...4 ~p1
```

The first "pattern unit" does not actually match anything; rather, it defines a "pairing rule" in which standard pairings are allowed, as well as G-A and A-G (in case you wondered, Us and Ts and upper and lower case can be used interchangably; for example r1={AT,UA,gc,cg} could be used to define the "standard rule" for pairings). The second line consists of six pattern units which may be interpreted as follows:

```
        p1=2...3     match 2 or 3 characters (call it p1)
        0...4        match 0 to 4 characters
        p2=2...5     match 2 to 5 characters (call it p2)
        1...5        match 1 to 5 characters
        r1~p2        match the reverse complement of p2,
                        allowing G-A and A-G pairs
        0...4        match 0 to 4 characters
        ~p1          match the reverse complement of p1
                        allowing only G-C, C-G, A-T, and T-A pairs
```

Thus, r1~p2 means "match the reverse complement of p2 using rule r1". Now let us consider the issue of tolerating mismatches and bulges. You may add a "qualifier" to the pattern unit that gives the tolerable number of "mismatches, deletions, and insertions". Thus,

```
            p1=10...10 3...8 ~p1[1,2,1]
```

means that the third pattern unit must match 10 characters, allowing one "mismatch" (a pairing other than G-C, C-G, A-T, or T-A), two deletions (a deletion is a character that occurs in p1, but has been "deleted" from the string matched by ~p1), and one insertion (an "insertion" is a character that occurs in the string matched by ~p1, but not for which no corresponding character occurs in p1). In this case, the pattern would match

```
          ACGTACGTAC GGGGGGGG GCGTTACCT
```

which is, you must admit, a fairly weak loop. It is common to allow mismatches, but you will find yourself using insertions and deletions much more rarely. In any event, you should note that allowing mismatches, insertions, and deletions does force the program to try many additional possible pairings, so it does slow things down a bit.

How Patterns Are Matched

Now is as good a time as any to discuss the basic flow of control when matching patterns. Recall that a "pattern" is a sequence of

```
    "pattern units".  Suppose that the pattern units were
        u1 u2 u3 u4 ... un
    The scan of a sequence S begins by setting the current position
    to 1.  Then, an attempt is made to match u1 starting at the
    current position.  Each attempt to match a pattern unit can
    succeed or fail.  If it succeeds, then an attempt is made to match
    the next unit.  If it fails, then an attempt is made to find an
    alternative match for the immediately preceding pattern unit.  If
    this succeeds, then we proceed forward again to the next unit.  If
    it fails we go back to the preceding unit.  This process is called
    "backtracking".  If there are no previous units, then the current
    position is incremented by one, and everything starts again.  This
    proceeds until either the current position goes past the end of
    the sequence or all of the pattern units succeed.  On success,
    scan_for_matches reports the "hit", the current position is set
    just past the hit, and an attempt is made to find another hit.
    If you wish to limit the scan to simply finding a maximum of, say,
    10 hits, you can use the -n option (-n 10 would set the limit to
    10 reported hits).  For example,
        scan_for_matches -c -n 1 pat_file < test_dna_input
    would search for just the first hit (and would stop searching the
    current sequences or any that follow in the input file).
Searching for repeats:
    In the last section, I discussed almost all of the details
    required to allow you to look for repeats.  Consider the following
    set of patterns:
        p1=6...6 3...8 p1   (find exact 6 character repeat separated
                             by to 8 characters)
        p1=6...6 3..8 p1[1,0,0]   (allow one mismatch)
        p1=3...3 p1[1,0,0] p1[1,0,0] p1[1,0,0]
                            (match 12 characters that are the remains
                             of a 3-character sequence occurring 4 times)

        p1=4...8 0...3 p2=6...8 p1 0...3 p2
                            (This would match things like
                                ATCT G TCTTT ATCT TG TCTTT
                            )
Searching for particular sequences:
    Occasionally, one wishes to match a specific, known sequence.
    In such a case, you can just give the sequence (along with an
    optional statement of the allowable mismatches, insertions, and
    deletions).  Thus,
        p1=6...8 GAGA ~p1    (match a hairpin with GAGA as the loop)
        RRRRYYYY             (match 4 purines followed by 4 pyrimidines)
```

```
        TATAA[1,0,0]          (match TATAA, allowing 1 mismatch)
```

Matches against a "weight matrix":

I will conclude my examples of the types of pattern units available for matching against nucleotide sequences by discussing a crude implemetation of matching using a "weight matrix". While I am less than overwhelmed with the syntax that I chose, I think that the reader should be aware that I was thinking of generating patterns containing such pattern units automatically from alignments (and did not really plan on typing such things in by hand very often). Anyway, suppose that you wanted to match a sequence of eight characters. The "consensus" of these eight characters is GRCACCGS, but the actual "frequencies of occurrence" are given in the matrix below. Thus, the first character is an A 16% the time and a G 84% of the time. The second is an A 57% of the time, a C 10% of the time, a G 29% of the time, and a T 4% of the time.

| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 |
|---|---|---|---|---|---|---|---|---|
| A | 16 | 57 | 0 | 95 | 0 | 18 | 0 | 0 |
| C | 0 | 10 | 80 | 0 | 100 | 60 | 0 | 50 |
| G | 84 | 29 | 0 | 0 | 0 | 20 | 100 | 50 |
| T | 0 | 4 | 20 | 5 | 0 | 2 | 0 | 0 |

One could use the following pattern unit to search for inexact matches related to such a "weight matrix":

```
    {(16,0,84,0),(57,10,29,4),(0,80,0,20),(95,0,0,5),
     (0,100,0,0),(18,60,20,2),(0,0,100,0),(0,50,50,0)} > 450
```

This pattern unit will attempt to match exactly eight characters. For each character in the sequence, the entry in the corresponding tuple is added to an accumulated sum. If the sum is greater than 450, the match succeeds; else it fails.

Recently, this feature was upgraded to allow ranges. Thus,

```
600 >  {(16,0,84,0),(57,10,29,4),(0,80,0,20),(95,0,0,5),
        (0,100,0,0),(18,60,20,2),(0,0,100,0),(0,50,50,0)} > 450
```

will work, as well.

Allowing Alternatives:

Very occasionally, you may wish to allow alternative pattern units (i.e., "match either A or B"). You can do this using something like

```
            ( GAGA | GCGCA)
```

which says "match either GAGA or GCGCA". You may take alternatives of a list of pattern units, for example

```
    (p1=3...3 3...8 ~p1 | p1=5...5 4...4 ~p1 GGG)
```

would match one of two sequences of pattern units. There is one clumsy aspect of the syntax: to match a list of alternatives, you need to fully the request. Thus,

```
(GAGA | (GCGCA | TTCGA))
```

would be needed to try the three alternatives.

One Minor Extension

Sometimes a pattern will contain a sequence of distinct ranges, and you might wish to limit the sum of the lengths of the matched subsequences. For example, suppose that you basically wanted to match something like

ARRYYTT p1=0...5 GCA[1,0,0] p2=1...6 ~p1 4...8 ~p2 p3=4...10 CCT

but that the sum of the lengths of p1, p2, and p3 must not exceed eight characters. To do this, you could add

```
length(p1+p2+p3) < 9
```

as the last pattern unit. It will just succeed or fail (but does not actually match any characters in the sequence).

Matching Protein Sequences

Suppose that the input file contains protein sequences. In this case, you must invoke scan_for_matches with the "-p" option. You cannot use aspects of the language that relate directly to nucleotide sequences (e.g., the -c command line option or pattern constructs referring to the reverse complement of a previously matched unit).

You also have two additional constructs that allow you to match either "one of a set of amino acids" or "any amino acid other than those a given set". For example,

```
p1=0...4 any(HQD) 1...3 notany(HK) p1
```

would successfully match a string like

```
YWV D AA C YWV
```

Using the show_hits Utility

When viewing a large set of complex matches, you might find it convenient to post-process the scan_for_matches output to get a more readable version. We provide a simple post-processor called "show_hits". To see its effect, just pipe the output of a scan_for_matches into show_hits:

Normal Output:

```
clone% scan_for_matches -c pat_file < tmp
>tst1:[1,28]
gtacguaacc  ggttaac cgguuacgtac
>tst1:[28,1]
gtacgtaacc  ggttaac cggttacgtac
>tst2:[2,31]
CGTACGUAAC C GGTTAACC GGUUACGTACG
```

```
    >tst2:[31,2]
    CGTACGTAAC C GGTTAACC GGTTACGTACG
    >tst3:[3,32]
    gtacguaacc g gttaactt cgguuacgtac
    >tst3:[32,3]
    gtacgtaacc g aagttaac cggttacgtac
 Piped Through show_hits:

    clone% scan_for_matches -c pat_file < tmp | show_hits
    tst1:[1,28]:  gtacguaacc   ggttaac  cgguuacgtac
    tst1:[28,1]:  gtacgtaacc   ggttaac  cggttacgtac
    tst2:[2,31]:  CGTACGUAAC C GGTTAACC GGUUACGTACG
    tst2:[31,2]:  CGTACGTAAC C GGTTAACC GGTTACGTACG
    tst3:[3,32]:  gtacguaacc g gttaactt cgguuacgtac
    tst3:[32,3]:  gtacgtaacc g aagttaac cggttacgtac
    clone%
```

Optionally, you can specify which of the "fields" in the matches you wish to sort on, and show_hits will sort them. The field numbers start with 0. So, you might get something like

```
    clone% scan_for_matches -c pat_file < tmp | show_hits 2 1
    tst2:[2,31]:  CGTACGUAAC C GGTTAACC GGUUACGTACG
    tst2:[31,2]:  CGTACGTAAC C GGTTAACC GGTTACGTACG
    tst3:[32,3]:  gtacgtaacc g aagttaac cggttacgtac
    tst1:[1,28]:  gtacguaacc   ggttaac  cgguuacgtac
    tst1:[28,1]:  gtacgtaacc   ggttaac  cggttacgtac
    tst3:[3,32]:  gtacguaacc g gttaactt cgguuacgtac
    clone%
```

In this case, the hits have been sorted on fields 2 and 1 (that is, the third and second matched subfields).

show_hits is just one possible little post-processor, and you might well wish to write a customized one for yourself.

## Reducing the Cost of a Search

The scan_for_matches utility uses a fairly simple search, and may consume large amounts of CPU time for complex patterns. Someday, I may decide to optimize the code. However, until then, let me mention one useful technique.

When you have a complex pattern that includes a number of varying ranges, imprecise matches, and so forth, it is useful to "pipeline" matches. That is, form a simpler pattern that can be used to scan through a large database extracting sections that might be matched by the more complex pattern. Let me illustrate with a short example. Suppose that you really wished to match the pattern

```
p1=3...5 0...8 ~p1[1,1,0] p2=6...7 3...6 AGC 3...5 RYGC ~p2[1,0,0]
```

```
    In this case, the pattern units AGC 3...5 RYGC can be used to rapidly
    constrain the overall search.  You can preprocess the overall
    database using the pattern:
          31...31 AGC 3...5 RYGC 7...7
    Put the complex pattern in pat_file1 and the simpler pattern in
    pat_file2.  Then use,
        scan_for_matches -c pat_file2 < nucleotide_database |
        scan_for_matches pat_file1
    The output will show things like
    >seqid:[232,280][2,47]
    matches pieces
    Then, the actual section of the sequence that was matched can be
    easily computed as [233,278] (remember, the positions start from
    1, not 0).
    Let me finally add, you should do a few short experiments to see
    whether or not such pipelining actually improves performance -- it
    is not always obvious where the time is going, and I have
    sometimes found that the added complexity of pipelining actually
    slowed things up.  It gets its best improvements when there are
    exact matches of more than just a few characters that can be
    rapidly used to eliminate large sections of the database.
=============
Additions:
Feb 9, 1995:   the pattern units ^ and $ now work as in normal regular
               expressions.  That is
                        TTF $
               matches only TTF at the end of the string and
                        ^ TTF
               matches only an initial TTF
               The pattern unit
                        <p1
               matches the reverse of the string named p1.  That is,
               if p1 matched GCAT, then <p1 would match TACG.  Thus,
                   p1=6...6 <p1
               matches a real palindrome (not the biologically common
               meaning of "reverse complement")
```