FFP 3.19 - Feature Frequency Profile Phylogenetics Package

Gregory E. Sims

Lawrence Berkeley National Lab

Preface

This is a collection of programs / utilities for implementing the FFP (Feature Frequency Profile) method of phylogenetic comparison. FFP is a class of alignment-free methods suitable for (whole genome) comparisons from viral to mammalian scale genomes. The fundamental unit of comparison is the based feature. Thus, when we speak of features and the feature profile we are referening to a vector of features which describes the frequencies of those features within the underlying sequences.

This method has been used to perform various phylogenetic analyses:

- Sims GE and Kim SH (2011) Whole-genome phylogeny of Escherichia coli/Shigella group by feature frequency profiles (FFPs). PNAS, 108, 8329-34.
- Jun SR, Sims GE, Wu GA, Kim SH. (2010) Whole-proteome phylogeny of prokaryotes by feature frequency profiles: An alignment-free method with optimal feature resolution. PNAS, 107,133-8.
- Sims GE, Jun SR, Wu GA, Kim SH. (2009) Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions. PNAS, 106,2677-82.
- Sims GE, Jun SR, Wu GA, Kim SH (2009) Whole-genome phylogeny of mammals: evolutionary information in genic and nongenic regions. PNAS. 106,17077-82.

OTHER REQUISITE PROGRAMS

We suggest that you obtain a copy of PHYLIP (http://evolution.genetics.washington.edu/phylip.html) for building trees, however you can use any tree building program which will accept distance matrix input. The utility ffpjsd will produce Phylip style 'infile's as well as raw distance matrices. As of FFP v3.06, a tree building utility, ffptree is included, which will allow you build Newick style tree output directly as part of a ffp pipeline, which is compatible with the Phylip (3.69) utilities.

The FFP Utilities

The utilities are designed to be implemented using unix command pipes. In other words the output of programs can be linked to the input of other programs. Therefore many of the scripts are acceptable as filters to be used in intermediate steps.

This package contains the following programs/scripts:

`ffpgui`	*Experimental.* A perl/Tk based (graphical interface) GUI interface for performing some of the basic FFP operations. This utility doesn't support grid-based/ multiprocessor job flow. Also `ffpgui` is in the beta stage, but it should give an example of what can be done with the utilities listed below. Note: Will not work properly in Cygwin. The Cygwin implementation of perl/Tk appears to have some serious deficiencies.
`ffpry`	Constructs an FFP profile from nucleic acid sequences in FASTA format (.fna).
`ffpaa`	Constructs an FFP profile from amino acid sequences in FASTA format (.faa).
`ffprwn`	This performs row normalization of the raw FFP matrix.
`ffpjsd`	This calculates the Jensen Shannon Divergence between FFPs and outputs a Divergence (Distance) matrix. A variety of other distances/similarity metrics are available as well.

`ffpboot`	This performs bootstrapping or jacknifing permutation of a raw FFP profile produced by `ffpry` or `ffpaa`
`ffpvocab`	This utility counts the number of words which are used more than a paritcular threshold in the FFP profile. This utility is used to determine what is the best range of word lengths to use for a genome collection
`ffpre`	This utility calculates the Relative entropy between the expected and observed frequencies of features of length *l* (specified on the command line) using an *l* - 2 Markov Model.
`ffpvprof`	Shell Script which calculates the word usage for a range of *l*. It executes the compiled executable `ffpvocab`. Use this to determine the lower limit for word lengths.
`ffpreprof`	Shell script which calculates the Relative entropy between observed and predicted frequencies for a range of *l*. Use this script to determine the upper limit for word length. It runs the underlying executable `ffpre`.
`ffpmerge`	This utility merges all rows of an FFP into a single row. Use this for merging segments of an FFP, for example different chromosomes of a larger genome.
`ffpcol`	This utility converts a FFP which has been written out in key/value format to a columnar format, so that each column corresponds to the same feature in each row of the FFP.
`ffptxt`	This utility creates a key/value FFP of text data. This is useful for performing an FFP analysis of human language texts. All non- alphanumeric characters are ignored.
`ffpfilt`	Eliminate high/low frequency features using frequency cutoffs or probability based cutoffs assuming a normal or extreme value distributions.
`ffpcomplex`	Eliminate high/low complexity features using a complexity cutoff or probability based cutoff assuming a normal distribution.
`ffpdf`	Finds clade distinguishing (diagnostic) features. See Sims GE and Kim SH (2011) PNAS, 108, 8329-34.
`ffptree`	Build neighbor joining and UPGMA trees from ffpjsd output.

FFP comparison of a collection of nucleic acid sequences

Assuming your nucleic acid `.fna` files (nucleotide sequences in fasta format) are all in the current working directory and (for the purposes of this example) are named with the `.fna` extension:

```
ffpry -l 3 *.fna | ffpcol | ffprwn | ffpjsd -p species.txt | ffptree > tree
```

To break down the terse syntax of the above examples, lets get an overview of the various utilities used in the commands above. First of all the `|` symbol is known as the pipe symbol, it is used in the construction of pipelines. It indicates that the output (the standard output) of the program should be read into the standard input of the next program. This piping mechanism is extremely powerful and is widely applicable to all areas of linux/unix scripting and programming. As you can see, several programs can be connected together in a chain creating a pipeline of programs of arbitrary length. The first command in the above pipelines is `ffpry`, which is a *l*-mer feature counting tool. You have likely seen what we refer to as an *l*-mer by various other names, such as a *k*-mer, *n*-gram, or *k*-tuple. All of these terms are synonymous, and imply that we are sliding a window of a specific length of letters along an arbitrarly long sequence and observing the frequencies of feature in those windows. `ffpry` reads in fasta file nucleotide sequences and counts the number of times each *l*-mer appears. By fasta input we mean a file of the form:

```
>gi|5835107|ref|NC_001640.1| Equus caballus mitochondrion, complete genome
GTTAATGTAGCTTAATAATATAAAGCAAGGCACTGAAAATGCCTAGATGAGTATTCTTACTCCATAAACA
CATAGGCTTGGTCCTAGCCTTTTTATTAGTTATTAATAGAATTACACATGCAAGTATCCGCACCCCAGTG
AGAATGCCCTCTAAATCACGTCTCTACGATTAAAAGGAGCAGGTATCAAGCACACTAGAAAGTAGCTCAT
AACACCTTGCTCAGCCACACCCCCACGGGACACAGCAGTGATAAAAATTAAGCTATGAACGAAAGTTCGA
CTAAGTCATATTAAATAAGGGTTGGTAAATTTCGTGCCAGCCACCGCGGTCATACGATTAACCCAAATTA
ATAAATCTCCGGCGTAAAGCGTGTCAAAGACTAATACCAAAATAAAGTTAAAACCCAGTTAAGCCGTAAA
AAGCTACAACCAAAGTAAAATAGACTACGAAAGTGACTTTAATACCTCTGACTACACGATAGCTAAGACC
```

```
CAAACTGGGATTAGATACCCCACTATGCTTAGCCCTAAACTAAAATAGCTTACCACAACAAAGCTATTCG
CCAGAGTACTACTAGCAACAGCCTAAAACTCAAAGGACTTGGCGGTGCTTTACATCCCTCTAGAGGAGCC
TGTTCCATAATCGATAAACCCCGATAAACCCCACCATCCCTTGCTAATTCAGCCTATATACCGCCATCTT
CAGCAAACCCTAAACAAGGTACCGAAGTAAGCACAAATATCCAACATAAAAACGTTAGGTCAAGGTGTAG
....
```

where the line begining with a `>` character is refered to as the sequence defline (Definition Line). `ffpry` uses the occurrences of these deflines to determine whether a new FFP is beginning. If your fasta file constains several deflines, i.e. it is a multi-fasta file, then each fasta file will be built into one feature profile. This feature can be disabled with the `-m` option if it is not what you intended (see the detailed section on ffpry for more information) and your file will be treated as multiple chunk of DNA. This might be useful if your file contains fastas, which are representative of more than one organism.

Our first example uses features of length 3, specified with the `-l` option. Lets just see what the output of `ffpry` might look like for the above fasta file for the horse (Equus) mitchondrion sequence, which I have saved as `equus_caballus.fasta`.

```
$ ffpry -l 3 equus_caballus.fasta
RRR  4582 RRY  4186 YRR  4185 YRY  3705
```

The output of ffpry is in key-value form, i.e. pairs of feature sequence followed by the raw count. But something is odd, why are the features printed out with `R` and `Y` characters?

The default behavior of `ffpry` it to count features in RY coded format where R stands for the purine bases A and G and the Y stands for the pyrimidine bases C and T. This transformation can be thought of compressing the alphabet size. We create two classes and recode the sequence using this two class compressed alphabet. In the case of R and Y the in-class members are biochemically related and it has been observed that the probability of mutation between two purines and likewise for the pyrimidines is more likely than a pyrimidine to purine mutation.This classing or coding of nucleotides can be disabled with `-d` if desired. To illustrate this point lets repeat the above example with the `-d` switch enabled.

```
$ ffpry -d -l 3 equus_caballus.fasta
TAA  790  TAC  512  TAG  803  TAT  803  \
GCA  349  GCC  351  GCG  141  GCT  459  \
CGA  240  CGG  194  CGT  221  ATC  589  \
ATG  634  ATT  860  TCA  658  TCC  546  \
TCT  501  GGG  500  GGT  495  CTC  498  \
CTG  389  CTT  517  AAA  770  AAC  577  \
TGG  528  TGT  558  GTC  277  GTG  487  \
CAA  583  ACT  578  TTC  581  CCT  669
```

There are a lot more features! Note that the `\` is our convention to indicate that the string is all printed out as one line, but we have inserted a newline to indicate that the line has been artificially wrapped to fit on the page. Lets take a closer look and examine how the features are actually counted. So lets just count the features in the first 10 characters of the sequence (to avoid the huge chore of manually counting *l*- mers in a long string). Lets save a truncated version of `equus_caballus.fasta` as `eq_ca_short.fasta`.

```
$ cat eq_ca_short.fasta
>gi|5835107|ref|NC_001640.1| Equus caballus mitochondrion, complete genome
GTTAATGTAG
```

We can manually count the 3-mers which will appear by visually scanning a window 3 characters wide, starting from `GTT` and scanning down the sequence, the next 3-mer will be `TTA`. After counting the entire string the set of 3-mers should be:

```
GTT  1    TTA  1    TAA  1    AAT  1    \
ATG  1    TGT  1    GTA  1    TAG  1
```

Lets see if ffpry gives us what we predicted.

```
$ ffpry -d -l eq_ca_short.fasta
```

```
TAA  2    TAC  1    TAG  1    ATG  1    \
ATT  1    AAC  1    TGT  1
```

The features reported don't appear to be the same as those produced by our manual counting:

1. The features appear different.
2. The ordering isn't the same
3. The feature frequencies are different.

So what's going on? Lets tackle them one by one. Well first of all the ordering of the features is different than the order as we encountered them in the truncated mitochondrial sequences. This is easy to explain, the features are printed out in *hash-order*. Features are stored internally in the hash table and printed out sequentially in the order they are encountered within the table -- that order will differ from the ordering in which they were counted.

However that still doesn't explain why there are some features in the output that we never even counted in the sequence, for example the feature `TAC`, is not in the sequence! The simple explanation is that ffpry counts features in both the forward and reverse complement directions. The feature `TAC` is actually the reverse complement of the feature `GTA`, which is in the sequence. So the features reported are printed in a mixture of both the forward and reverse complements. The choice of whether a feature is stored and printed in the forward or reverse direction is decided internally -- but is determined by hash-precedence. Where a feature is placed in the hash table is determined using a numerical *hash index*. As a fasta file is scanned for features, both the hash index of the forward and reverse complement of the word are calculated, which usually are different from each other, except in the case of reverse palindromes. To save space and for efficiency, the word and the associated features are only stored once using the smaller of the two hash indices. Therefore `GTA` is stored as `TAC` because its hash index is smaller.

The final puzzle which seems odd is that the feature `TAA` has a frequency of 2. But it only appears once in the mitochondrial sequence. You must remember to think in terms of the reverse complement word too! `TAA` also occurs on the reverse complement strand, but you see the feature in the mitochondrial sequence as `TTA`.

In the case of multiple input sequences or a fasta file with multiple fasta records, each row corresponds to the features from that sequence. The rows are ordered in the corresponding order of input. Therefore if we build an ffp with both horse and dog mitochondria, we might execute this command:

```
$ ffpry -l 3 equus_caballus.fasta canis_lupus.fasta

RRR  4582 RRY  4186 YRR  4185 YRY  3705
RRR  4496 RRY  4113 YRR  4113 YRY  3999
```

The first row corresponds to the features from Equus and the second to Canis. Names indicating species are currently not attached to the FFP rows at this point -- but are attached at a later stage (ultimately the tree building stage). So you should be aware of the ordering in which the fasta sequences were supplied to the command line. The method that seems easiest is to let your shell expand a wildcard (i.e. `*.fasta`) for you. For example lets say I have several fasta files in my current working directory. To see them I can use:

```
$ echo *.fasta

Bos_taurus.fasta Bubalus_bubalis.fasta Canis_lupus_laniger.fasta \
Colobus_guereza.fasta Equus_asinus.fasta Equus_caballus.fasta \
Eulemur_monogoz.fasta Lemur_catta.fasta Myotis_formosus.fasta \
Nycticebus_coucang.fasta Ovis_aries.fasta Ovis_canadensis.fasta \
Pan_troglodytes.fasta Pongo_abelii.fasta Rangifer_tarandus.fasta \
Rattus_lutreolus.fasta Rattus_tunneyi.fasta Tarsius_bancanus.fasta
```

I can take this listing and save it to a file for later, perhaps changing each name to a shorter name for a label in a phylogenetic tree.

```
$ \ls *.fasta > species.txt
```

This will create a nice newline delimited file containing a list of all the fasta files in your current directory in the same order as your shell might expand `*.fasta`. Note that the `\` in this case indicates we want to unalias any commands which may have been aliased to `ls`. This is useful if for example you are using `ls` with

the `--color` option enabled, and have it aliased to `ls`. If you don't your `species.txt` file will contain ANSI escape codes (e.g. nonsense like: `^[[0m`) which indicate color markup, used to display your `ls` output in fancy file-type specific colors. If I want to build an FFP using all of the fasta files in my current directory we can just issue this command:

```
$ ffpry -l 3 *.fasta > vector
```

Take special note that the order in which `*.fasta` is expanded should be identical to the ordering of fasta files we see above. We have a record of the ordering of genomes in `species.txt`. Therefore the ordering of rows in the output ffp file `vector` will have the same ordering as the shell expansion of `*.fasta`.

Now we're ready to take another look at this pipeline.

```
ffpry -l 3 *.fna | ffpcol | ffprwn | ffpjsd -p species.txt | ffptree > tree
```

The output of `ffpry` is piped to a utility called `ffpcol`, which converts the key value form into a columnar form. This form discards the key labels, and retains the frequency values. In addition the values are transformed in such a manner that the frequency for a given feature across all FFP rows is in the same tabular column -- even if that feature is absent from that FFP row. As an example, given these to FFP rows

```
$ cat vector

RRR  4582 RRY  4186 YRR  4185
RRR  4496 YRR  4113 YRY  3999
```

The `ffpcol` transformation will produce a columnar representation which looks like this:

```
$ ffpcol vector > vector.col

$ cat vector.col

4582 4186 4185 0
4496 0    4113 3999
```

Notice that the the feature `YRY` is absent from row 1 and the feature `RRY` is absent from row 2, therefore zero frequencies are printed in column 2 and 4. Notice that the raw counts corresponds to the same features in every column across both rows. Also, we'll point out here that if you choose to disable the compressed alphabet at the feature counting stage you must ALSO include the ffpcol call, for example:

ffpry -l 3 -d ffp | ffpcol -d > vector.col

The next utility in the pipeline, `ffprwn`, row normalizes each row of the ffp feature matrix so that each element of that row is a relative frequency. Row normalization is simple; add up all the raw frequencies for a row, then divide each element by the row sum. This type of normalization isn't necessary in all cases, but it is necessary to make FFP comparisons using the default distance, the Jensen Shannon divergence, which is calculated with the `ffpjsd` utility. For example the above columnar FFP will appear like so after normalization:

```
$ ffprwn vector.col > vectors.row

$ cat vectors.row

3.54e-01 3.23e-01 3.23e-01 0.00e+00
3.57e-01 0.00e+00 3.26e-01 3.17e-01
```

The row normalized FFP output is now piped to `ffpjsd`. which will produce a divergence (distance) matrix in Phylip style format. It is at this point where we specify the species names of each of the taxon rows by using the `species.txt` file (note however you can name it whatever you want). You should create unique 10 character names in `species.txt` -- that is characters which are unique within the first 10 characters. This is for compatibility with the Phylip tools itself. If you do specify a taxon name longer than 10 characters it will be automatically truncated to 10 characters (with a warning printed to standard error), but Phylip tools or `ffptree` will still generate an error if your names are not unique. The `-p` option is used to point to your `species` file. Note once again, that the names of the species assigned in `species.txt` must match the ordering the original fasta files were given in the call to `ffpry`. The resulting output which is a Phylip style 'infile' is then piped into ffptree, which prints a tree to standard output in Newick format. Additionally tree build progress and a human readable tree are printed to standard error. If you want to save the human readable tree make sure to redirect it via standard error.

The above pipeline mechanism is quite powerful. Of course you can save the output at each step in intermediate files in the following form:

```
ffpry -l 3 *.fna > vectors
ffpcol vectors > vectors.col
ffprwn vectors.col > vectors.row
ffpjsd -p species.txt vectors.row > infile
ffptree infile > tree
```

As you can see, the pipeline form avoids the creation and subsequent cleanup of several intermediate files. However there are legitimate uses for creating and keeping around some of these intermediate files -- in particular performing multiple analyses using an intermediate step as a branching point. For example if you want to perform bootstrapping on the `vectors.col` file using the `ffpboot` utility or if you wanted to try out a few of the different distance measures offered by `ffpjsd` using the same `vectors.row` file.

FFP comparison of a collection of amino acid sequences

The FFP tools can also be used to build phylogenies from amino acid sequences or proteomes. The pipeline is very similar to that used for building a nucleic acid phylogeny. You need only to substitute the initial *k*-mer counter `ffpaa`. However there are some subtleties to keep in mind when using `ffpaa`. Foremost is the method of amino acid classing. By default the 20 amino acids are divided into 11 classes:

`S,T`	The polar hydrophillic amino acids
`D,E`	The negatively charged hydrophillic amino acids
`K,Q,R`	The positively charged hydrophillic amino acids
`I,V,L,M`	The non-polar hydrophobic amino acids
`F,W,Y`	The large non-polary hydrophobic amino acids
`C`	Cysteine - a singleton class
`G`	Glycine - a singleton class
`A`	Alanine - a singleton class
`N`	Asparagine - a singleton class
`H`	Histidine - a singleton class
`P`	Proline - a singleton class

When an amino acid fasta file is scanned for *l*-mer amino acid, those symbols which are members of a multi-character class are substituted for the first symbol of the class. For example, if `Q` or `R` is encountered in an amino acid sequence either is substituted for the character `K`, because `Q` and `R` are part of the (`K,Q,R`) class. This classing can be disabled by using the `-d` option.

The next matter to consider is how your proteome file is parsed and processed. Lets say we have a multi-fasta file, with two records.

```
$ cat fastafile
>tr|C7NM83|C7NM83_HALUD Endo-1,4-beta-xylanase
MSSDKTLRELADKNDLTLGASITADAFRTYPDDPAVAQTLTREFNAVTTGNALKMGPLRP
ERYTYNFEDADAIVNLGVKNDLLVRGHALVWHNQTPGWFYPWEYTDDQLREFLRDHIHTV
AGRYRGKVDVWDVVNEAVADDGTMRETAWYDAMGEEYIDLAFQWANEVAPEADLFYNDYG
IDEINEKADGVYALLERLLDRGVPIDGVGLQMHAFRAQEYVTPEALGENIRRFKDLGLDV
HVTEMDVAYDRENVPEDHLEHQAQYYRDILEACLDNGCDTLVTWGVHDTASWLRNYDQTI
TDDPLLFDEDFDPKPAYFAIKDLLANKD

>tr|C7NP66|C7NP66_HALUD Endo-1,4-beta-xylanase
MSDTLRDVADDNDIKIGAAAAADPIRGDFQYRDALREFNAVTAENAMKMGPLRPDEHTYD
FTDGDLIAEFAREHDMYFRGHVLVWHNQLPEWLLPFQYTDRELRRLLEDHVRTVAARYAG
DVDTWDVVNEAVADDGG*RETPWLRAFGEEYLDKAFEWAHQSAPEADLFYNDYGADGIND
KSDEIYEMVSGMLDRGVPIDGVGLQLHALHDPVDPDSVAENIERFKDLGLAVEITEMDVA
```

```
YTAEDPPEDHQEVQADYYREVVEKAMAAGCDTFVIWGVADHHSWIPHFDDTLTDDPLLLD
DGYDRKPAYDAIVDLLS
```

Lets say we execute the following command:

```
$ ffpaa -d -l 4 fastafile
```

Note that the second sequence has a intervening stop codon. The features are counted from the first sequences up until the end of the sequence. The blank space between the fasta records and the header is ignored. However features do not extend from one sequence to the next. For example the last 4-mer in `C7NM83` is `ANRD`. and the first feature in `C7NP66` is `MSDT`. To make this explict features never span two fasta records. Therefore you will *not* see feature `KDMS` (the first two features from `C7NM83` and the last two from `C7NP66`).

Now what about that `*` character, the stop codon? Features are counted in the sliding windows which do not contain the stop codon. Therefore `DDGG` and `RETP` are counted but `DGG*`, `GG*R`, `G*RE` and `*RET` are completely ignored. In fact all characters which are not the canonical 20 amino acid single letter symbols are completely ignored.

Text comparison

The text data analog of `ffpry` and `ffpaa` is `ffptxt`. Its capabilities are highly simplified relative to these other two utilities. `ffptxt` will take text input, strip out all characters which are not alphabetical and count *l*-mers. The behavior when encountering invalid characters (i.e. non alphabetic characters) is similar to the method used for invalid characters in nucleotide and amino acid sequences. The entire feature window must contain valid characters. Lets look at an example:

```
$ cat README

FFP 3.06 - Feature Frequency Profile Phylogenetics Package
August 24, 2011

$ ffptxt -l 3 README

AGE  1    LEP  1    CYP  1    UEN  1    \
ILE  1    OGE  1    HYL  1    EPH  1    \
ETI  1    QUE  1    PRO  1    ATU  1    \
EAT  1    EAU  1    REF  1    REQ  1    \
URE  1    SPA  1    NET  1    EQU  1    \
CKA  1    CSP  1    LOG  1    AUG  1    \
UGU  1    FEA  1    FIL  1    EFR  1    \
UST  1    PHY  1    ENC  1    ENE  1    \
KAG  1    YLO  1    ICS  1    YPR  1    \
GEA  1    GEN  1    TIC  1    FFP  1    \
PAC  1    TUR  1    ROF  1    OFI  1    \
NCY  1    GUS  1    FRE  1    ACK  1
```

Notice that the punctuation, spaces, and numbers have not been counted in any features. `ffptxt` only counts alphabetic features. Also notice that there is no concept of records in a text file, therefore the entire text file is always interpretted as one piece of data. There is no `-m` option.

Bootstrapping your data

First of all, what is bootstrapping? It is a statistical concept introduced to phylogenetics by Joseph Felsenstein. It involves resampling the data one has at hand to create a pseudo-replicate of the data. The point of such an operation is to determine the robustness of conclusions one has made from the phylogenetic tree. An FFP based tree which depends heavily on the presence of a specific feature or small set of features may not be very robust (ultimately the tree may still be correct, even though it lacks robustness). Generally we assign higher certainty to trees which are more robust -- i.e. they are less likely a statistical fluke. The FFP utility used for resampling is `ffpboot`. It will perform two main kinds of resampling: bootstrapping and jackknifing. Lets show the difference between the two using an example. Lets say we have a columnar FFP matrix:

```
$ cat ffp.col
2    1    3    0    4
1    4    0    2    3
3    3    1    3    4
```

Bootstrapping involves creating a new FFP matrix from this original matrix. The new matrix will also contain 6 columns, but we will randomly select columns from the original matrix to choose in the new matrix. This new matrix is called a *pseudo-replicate*. Note that some columns may be sampled more than once, or sometimes not at all. We can produce a pseudo-replicate of the above matrix using the default mode of `ffpboot`

```
$ ffpboot ffp.col
3    3    2    4    1
0    0    1    3    4
1    1    3    4    3
```

Notice that column 3 was sampled twice, columns 1,2, and 5 were sampled once, and column 4 was sampled zero times.

The other form of resampling is jackknifing, which is implemented with the `-j` option. In this case each of the n columns in the matrix have a fixed probability of deletion. The deletion probability is by default 1-1/e (which works out to about 0.367, for those of you who can't take powers of e in your head), but can be set with the Lets-poption.

```
$ ffpboot -j ffp.col
     1    3    0    4
4    0    2    3
3    1    3    4
```

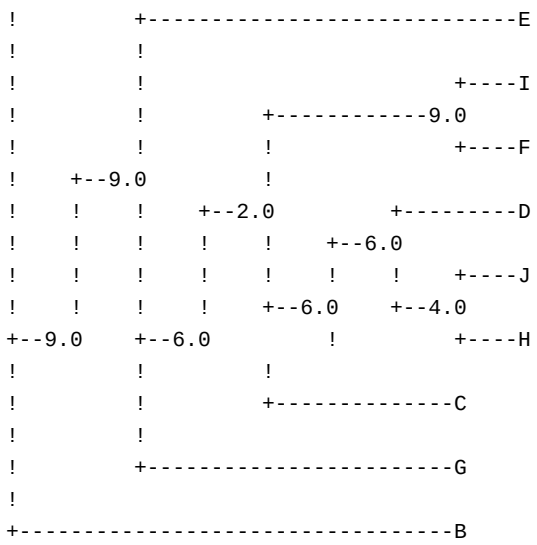
You will notice that column 1 of the matrix was selectively deleted. Typically, I will use jackknife runs which remove a large fraction of the features typically with a deletion probability of 60-70%.

Lets see how we can incorporate this bootstrapping concept into an FFP pipeline. We first need to decide how many pseudoreplicates to make -- there is no firm number on this, but lets say 100 replicates is enough. Lets create our replicates using a bash shell loop:

```
for i in {1..100} ; do
ffpboot ffp.col | ffprwn | ffpjsd -p species.txt | ffptree -q
done > intree
```

The statement on the third line, `done > tree`, indicates to redirect the output of all the commands within the do-while loop to the file `intree`, which will contain 100 trees produced from the pseudo-replicates. This collection of trees can be passed to a consensus tree finder, such as the Phylip program `consense`. The kind of output you might get from consense from a consensus tree might look like the eaxample below (Note this is from the Phylip documentation). Given the following input data (note there are no branch lengths, typically FFP trees will have branch lengths), which represent 9 replicates:

```
$ cat intree
(A,(B,(H,(D,(J,(((G,E),(F,I)),C))))));
(A,(B,(D,((J,H),(((G,E),(F,I)),C))));
(A,(B,(D,(H,(J,(((G,E),(F,I)),C))))));
(A,(B,(E,(G,((F,I),((J,(H,D)),C))))));
(A,(B,(E,(G,((F,I),(((J,H),D),C))))));
(A,(B,(E,((F,I),(G,((J,(H,D)),C))))));
(A,(B,(E,((F,I),(G,(((J,H),D),C))))));
(A,(B,(E,((G,(F,I)),((J,(H,D)),C))));
(A,(B,(E,((G,(F,I)),(((J,H),D),C))));
$ consense
+-------------------------------------A
!
```



The numbers displayed on the internal nodes represent how many trees in which that particular clading (to the right of the node) was observed. You can see that the clade I,F,D,J,H and C has relatively low support among the replicate trees. This grouping is only observed in 2 of the 9 trees. In this way we can identify the strong and weak clades in our phylogenetic tree.

Finding the right length.

We've used a lot of different examples of running ffp pipelines with different lengths of *l*-mers, but which one if the best to use? Well it turns out to be a difficult question to answer, but the package has a number of tools included to help you narrow down the right range of lengths to use. There are ultimately two main tools: `ffpvocab` and `ffpre` (in addition to a couple of wrapper scripts) which help you zoom in on the right lengths. First lets illustrate the identification of the lower limit using the `ffpvocab` wrapper script `ffpvprof` which creates a *vocabulary feature profile* over a range of feature lengths. We define a *vocabulary feature* to be a feature which occurs more than once in the genome. Hence this feature is part of the genome's regular *vocabulary.* Think of this in terms of a human's vocabulary. You have several words you use on a daily basis and there are certain words you *overuse*. These are words that are part of your repoitore. Sure there are other lofty really long words that you've diligently commited to memory and you look for opportunities to use, but the occasion only happens rarely. If we plot those words by their length, only recording the number of words that you used at least twice in the day, you will observe a distribution of frequencies which has a pronounced peak in the distribution at a particular length. This states that the words which you use most commonly have this particular length. This is a phenomenon observable in all forms of data -- including sequence information. Lets see how it works with a real genome sequence (in this case an *E. coli* strain):

```
$ ffpvprof -f 2 -d -r NC_008253.fna
1 4.000000e+00
2 1.600000e+01
3 6.400000e+01
4 2.560000e+02
5 1.024000e+03
6 4.096000e+03
7 1.533500e+04
8 4.001100e+04
9 6.263300e+04
10 6.327900e+04
11 5.089400e+04
12 2.813500e+04
13 1.139600e+04
```

```
14 4.127000e+03
15 1.672000e+03
16 8.110000e+02
17 5.980000e+02
18 4.540000e+02
19 4.050000e+02
20 3.830000e+02
```

The first column is the feature length and the second column is the number of features which occur at least 2 times at that length. Notice that the peak in the distribution occurs at length 10. This means that this *E. coli* genome reuses words that have a length of 10 most frequently. We demark this peak at 10 as the lower limit for phylogenetic analysis. Words at this length are very commonly used in many different context and have very little information content.

To find the upper limit you can use the utility `ffpre` and the wrapper script `ffpreprof`. Lets investigate what `ffpre` does. It calculates the relative entropy error between the observed frequency of an *l*-mer and an *l*-2 Markov model. To illustrate lets say we are reading the children's book Peter Pan and we are curious about the different contexts in which we observer the word "watch". and lets pull out an extra letter to the left and right of watch:

```
to watch her         owatchh
were watching them   ewatchi
and watch them       dwatcht
Michael watched them lwatche
was watching from    swatchi
kept watch outside   twatcho
keep watching that   pwatchi
your watch. Peter    rwatchp
```

You can see that watch occurs in many different contexts. We want to extend the letters around watch until we can predict the context of the those letters or they are unique in the sequence. By extending the length of the word by just two letters the context becomes unique. Therefore if I encounter the word 'owatchh', I know it is part of the larger phrase 'towatchher'.

The *l*-2 Markov model says that I can predict the frequency of a word of length *l* using the frequencies of its *l*-1 and *l*-2 subwords. The formula for doing this is:

$$f_l = f_{2..n} \frac{f_{1..n-1}}{f_{2..n-1}}$$

or put another way, using the feature GGCC as an example,

$$f_{GGCC} = f_{GGC} \frac{f_{GCC}}{f_{GC}}$$

The relative entropy measure that is calculated by `ffpre` makes use of this equation to arrive at an estimate of frequencies for all words in a sequence found at length *l*, then compares that estimate to the actual frequency. The smaller the relative entropy then the closer the estimate is to reality. When the relative entropy is very small then generally this means that we can predict the frequencies of all words from smaller sub-words. We have defined this as the upper limit. `ffpre` will read a sequence and calculate the relative entropy measure for specific values of *l*. If you want to calculate this value for a range of *l*, then you should use `ffpreprof`, which works in an analogous way as does `ffpvprof`. In this case instead of looking for the peak, you should be looking for an elbow point, for *l* when the relative entropy approaches zero (i.e. when it is very close to zero). Here is an example from the *E.*coli genome:

```
$ ffpreprof -e 30 NC_008253.fna
3 -0.379976
4 2.677588
5 -0.204627
6 1.299080
7 -0.060150
8 0.797391
```

```
9 0.060862
10 0.594577
11 0.175684
12 0.490803
13 0.272426
14 0.424031
15 0.263974
16 0.348154
17 0.226826
18 0.187225
19 0.133277
20 0.120039
21 0.081689
22 0.181143
23 0.414640
24 0.015185
25 0.172817
26 0.001127
27 0.000283
28 0.000207
```

We approach zero above *l*=26. Therefore this is the upper limit for this genome.

Feature filtering

Under some conditions you may want to remove some features from your key-value form FFP. The package contains two utilties, `ffpfilt` and `ffpcomplex`, which implement two separate filtration methods.

The utility `ffpfilt` will perform filtration of features by their frequency -- specifically filtration of features which have either unusually low or unusually high frequencies. How high or low that frequency needs to be can be determined in a number of ways -- the simplest being a defined frequency threshold. Lets assume I have my key-valued ffp stored in the file, `keyvalue.ffp` and I want to remove all features which occur more that 20 times.

```
ffpfilt < keyvalue.ffp | ffpfilt -u 20 > ffp.filt
```

In addition we can specify a lower threshold if for example we wanted to remove features which occur less than 2 times.

```
ffpfilt < keyvalue.ffp | ffpfilt -l 2 -u 20 > ffp.filt
```

In this case the `-l` indicates a *lower* threshold, don't confuse it with the switch representing length in the various other utilities. To avoid the confusion you can use the long format options:

```
ffpfilt < keyvalue.ffp | ffpfilt --lower 2 --upper 20 > ffp.filt
```

The other mode of `ffpfilt` allows you to identify and remove features which have frequencies lie that outside a given probability range assuming a specific underlying cummulative distribution function. The two distributions which can be used are the normal distribution (`-n`) extreme value distribution (`.-e`) In either of these modes the arguments to `--lower` and `--upper` are floating point values which represent a probability threshold in the cummulative distribution threshold. Here is a concrete example, Lets say I want to remove features which have frequencies which are in the lower 10% of the extreme value distribution (i.e. rare features):

```
ffpfilt < keyvalue.ffp | ffpfilt -e --lower 0.10 > ffp.filt
```

We can also combine this with an upper limit.

```
ffpfilt < keyvalue.ffp | ffpfilt -e --lower 0.10 --upper 0.90 > ffp.filt
```

which removes features which are both common and rare. Filtering with the normal distribution works in a similar fashion.

The utility `ffpcomplex` is used to filter out features by their sequence complexity. To give a simple example `ATATATATATAT`, is a low complexity feature, but `ATGAATGAGACC` is a higher complexity feature. Numerically, the complexity of a feature is determined by finding the total entropy of all individual sub features for all *k* less than *l*, which is the length of the feature. For example given the feature

```
GCGCGCGC
```

determine the entropy of *k*=1, which which is $H_{k=1} = f_G \log f_G + f_C \log f_C$, where f_G for example is the relative frequency of `G` in the original feature of length *l* above. Also note for simplicity log is actually $\log_2$. Next determine the entropy of all *k*=2, sub-mers, $H_{k=2} = f_{GC} \log f_{GC} + f_{CG} \log f_{CG}$. Repeat for all *k* less than *l*. The total complexity of the *l*-mer is $H_l = H_1 + H_2 + \cdots + H_{l-1}$. So lets see what the complexity of

```
$ cat feature.ffp
GCGCGCGC  1

$ ffpcomplex -d --stats feature.ffp
3.671582 0.000000
```

The first value indicates that `GCGCGCGC` has a complexity of 3.67. Ignore the second value for now. Likewise a more complex feature, a`GTAGGTGA`has Lets look at a real sequence to see how these complexity numbers really work. Lets print out some statistics:

```
$ ffpry -l 10 Bos_taurus.fasta | ffpcomplex --stats
4.900724 0.212182
```

This tells us that the average complexity of the features in `Bos_taurus.fasta` with a length of 10 is `4.900724` and the standard deviation is `0.212182`. We can use these numerical values to filter out sequences based upon the raw complexity or we can use probabilities from an assumed normal distribution.

Here we use a raw probability threshold.

```
ffpcomplex -l 4.0
```

Here we assume a cummulative normal distribution and upper and lower limits.

```
ffpcomplex -n -l 0.1 -u 0.95
```

Finding clade distinguishing features

Okay so I've got a tree. Now you'd like to backtrack and figure out what features make the tree have this specific topology. What features are indicative of the pattern of clading (grouping) which you see in the tree? A distinguishing feature (DF) is a feature which is present in all FFPs of a given group, but absent in all other groups. Given these example rows from an FFP matrix,

```
Taxon 1 ... ATG 3 CAC 2 TGA 3 ...
Taxon 2 ... ATG 2 CAC 1 TTG 1 ...
Taxon 3 ... ATG 4 TGC 3 GGA 2 ...
```

and the additional information that taxa 1 and 2 belong to group A and taxon 3 belongs to group B, we can give an example of a DF. The feature, ATG, is not a DF for either group A or B, since it is present in all groups. However feature CAC is a DF for group A, since it is present in only group A and not group B. Likewise feature TGA and TTG are not DFs of group A because neither is present in all members of that group.

To find distinguishing features in an FFP, first you must create an FFP, using one of the feature counting utilities, ffpry, ffpaa, or ffptxt. The example below illustrates the method for RY-coded nucleotide sequences, where we want to find features which are universally shared among all taxa specified in the FFP input file.

```
ffpry -l 6 test*.fna > vector
ffpdf vector > dfs
```

or alternatively, combining the separate commands as one pipeline:

```
ffpry -l 6 test*.fna | ffpdf > dfs
```

The above example assumes every row (genome, species, or sequences) is a member of the same group (clade). For multiple clades, use the --group-file option to specify the clade structure. This is defined by

building a newline delimited group-file definition. Lets say there are 5 genomes and there are two clades, the 2nd and 3rd rows of the FFP, belong to clade A and the remainder belong to clade B, then the group file should have the format below

```
$ cat groupfile
B
A
A
B
B
```

Now that clades have been defined we can extract out the DFs for clade A or B separately (A in this case below).

```
ffpry -l 6 test*.fna | ffpdf --group-file="groups.txt" --group="A" > dfs.A
```

Multi-character alphanumeric symbols can also be specified for group names. Note however, any white-space will be stripped from group names. Also, note that by default a DF is defined as a feature which is present in all members of a group. This defintion can be relaxed slightly by using the -p option to specify the percentage of taxa within a given group which must have a feature in order for it to be considered a DF. For example if we wanted to find out which features are present in 2 out of 3 taxa in group B we can use this syntax:

```
ffpry -l 6 test*.fna | ffpdf -p 0.66 --group-file="groups.txt" --group="A" > dfs.B
```

FFP with large genomes

So far we haven't mentioned how to apply FFP with large genomes. Typically a large genome, such as a Eukaryote, might already divided up into into individual files a the chromosome level -- or perhaps into individual contig assemblies. You might even be interested in comparing chromosomes from the same species or cross comparing chromosomes across species. Typically though, you'll probably want to merge the individual chromosome based FFPs into a single species level FFP. Lets say we're interested in the human genome, and we've downloaded individual chromosomes and named them conveniently as chr1.fna, chr2.fna, ... chrY.fna. We can use some bash shell magic (You'll find an equivalent in any other shell variant as well):

```
for i in {1..22} X Y ; do
ffpry -l 15 chr${i}.fna > chr${i}.ffp
done
```

The above code snippet is a bash shell loop that iterates iterates over the 22 numbered chromosomes and the sex chromosomes. Lets neglect the fact that each one of these steps may take some time for now -- we'll discuss setting FFP up to run on a multiprocessor grid as well. If we want a Human species level ffp we should merge the individual chromosome ffps. This can be done using `ffpmerge`.

```
ffpmerge -k chr*.ffp > human.ffp
# Or more explicitly using shell substitutions
ffpmerge -k chr{1..22}.fna chr{X,Y}.fna > human.ffp
```

This produces a merged FFP file consisting of the entire set of features from the files specified as arguments. Note that the default mode of ffpmerge creates columnar output, therefore if you plan on comparing several species which are the result of mergings you should use the `-k` option. So, lets say we have several species FFPs which we have calculated and saved, in the manner as above, say human, chimp, gorilla, etc. We can concatenate them into a single ffp file if we choose, or specify them all as file arguments:

```
$ cat > species.txt
human
chimp
gorilla
ctrl-d
# Here are 2 options for doing the same procedure
$ cat human.ffp chimp.ffp gorilla.ffp | ffpcol | ffprwn | ffpjsd -p species.txt |
```

```
ffptree
$ ffpcol human.ffp chimp.ffp gorilla.ffp | ffprwn ffpjsd -p species.txt | ffptree
```

Of ccourse, we've been gently introducing you to the idea that the operations above can be considered as atomic processes. The most effective way to exploit atomic processes and get the job done quickly is to use a multi-processor architecture. The examples below assume you are using a grid architecture like Sun Grid Engine.

We calculate the FFP of each chromosome on a different processor. If your cluster machine uses a qeueing system (i.e. Grid engine) then you can create individual shell scripts to give to the scheduler and then merge unit vector files after all scheduled jobs have completed. A simple example using grid engine employs the $SGE_TASK_ID variable.

Save a file containing the paths to your sequences There are 10 files total:

```
$ cat > sequences.txt
seq.fna
seq2.fna
seq3.fna
...
Ctrl-D

$ cat > submit.sh
#!/bin/bash
#submit.sh

FILE=‘head -n $SGE_TASK_ID < $1 | tail -n 1‘
ffpry -l 10 $FILE > $FILE.vector
Ctrl-d

$ chmod +x submit.sh
$ qsub -a 1-10 submit.sh sequences.txt
```

This creates a shell script which can be used in a SGE Job Array. The way a job array works is essentially identical code is submitted to the grid with the exception of the value of `SGE_TASK_ID` "which is set by SGE." The qsub line option `-a` specifies that we want a job array consisting of 10 jobs. When a job is run on the grid `SGE_TASK_ID` will contain a value between 1 and 10. The script we've created, `submit.sh`, uses the `SGE_TASK_ID` to extract the ith (or more properly the `SGE_TASK_IDth` which is truly a mouthful) line from the file `sequences.txt`. It should be fairly easy to extend this to all the other utilities used in the previous pipelines in this tutorial. One additional option to be aware of for especially large FFPs is the `-r` option to `ffpjsd`. It allows you to calculate the nth row of a ffp matrix. In this way you can allow one processor to calculate a single row of a matrix, later "assembling the matrix" by concatenation into a single matrix. For example a submit script might look like this:

```
#!/bin/bash
#mat_submit.sh


ffpjsd -r $SGE_TASK_ID file.ffprwn > row.$SGE_TASK_ID
```

We can then submit this script as a job array. Now the matrix can be assembled via a simple concatenation

```
#Lets say the matrix had ten rows
cat row.{1..10} > matrix
```