UPPSALA
UNIVERSITET

UPTEC X 13 015
Examensarbete 30 hp
Maj 2015

Stealth tRNAs: Strategies for mining orthogonal tRNA candidates from genomic data

Ingemar Ohlsson

UPPSALA UNIVERSITET

Bioinformatics Engineering Program

Uppsala University School of Engineering

UPTEC X 13 015	Date of issue 2015-05

Author

Ingemar Ohlsson

Title (English)

Stealth tRNAs: Strategies for mining orthogonal tRNA candidates from genomic data

Title (Swedish)

Abstract

Pairs of orthogonal tRNAs and aminoacyl-tRNA synthetases can potentially be used to augment the genetic code of a chosen host organism. Contemporary methods for finding candidate orthogonal tRNAs - ones that do not interact with the host's aminoacylation enzymes - are based on resource-intensive *in vivo* assays. In this project, I have evaluated several bioinformatics approaches to finding candidate orthogonal tRNAs, dubbed "Stealth tRNAs." Information logos obtained with the logofun software package, and rough set classification using the ROSETTA software package, show some ability to distinguish known orthogonal tRNAs from others. With further study and proper adaptation of the software, mining Stealth tRNAs from genomic data appears entirely possible.

Keywords

Bioinformatics, tRNA, orthogonal, genomic, data-mining

Supervisors

David H. Ardell
University of California, Merced

Scientific reviewer

Suparna Chandra Sanyal
Uppsala University

Project name	Sponsors
Language **English**	Security
ISSN 1401-2138	Classification
Supplementary bibliographical information	Pages **43**

Biology Education Centre Biomedical Center Husargatan 3 Uppsala
Box 592 S-75124 Uppsala Tel +46 (0)18 4710000 Fax +46 (0)18 471 4687

Stealth tRNAs:

Strategies for mining orthogonal tRNA candidates from genomic data

Ingemar Ohlsson

Populärvetenskaplig sammanfattning

Proteinkodande gener i alla levande organismer skrivs av från DNA till messenger-RNA (mRNA) som utgör en sekvens av instruktioner till ribosomen som sätter samman proteiner, de viktigaste komponenterna i biologiska mekanismer. Instruktionerna i mRNA läses av i kodon (avsnitt om tre nukleinsyror i taget) som var för sig korresponderar till en viss aminosyra, byggstenarna som kedjas ihop till proteiner av ribosomen.

Denna korrespondens mellan 64 kodon och 20 aminosyror utgör den genetiska koden, som bibehålls av transport-RNA (tRNA) - molekyler som binder till ett specifikt kodon och en specifik aminosyra - och de aminoacyl-tRNA-syntetas-enzymer (AARS) som laddar ett specifikt tRNA med sin associerade aminosyra.

Den genetiska koden kan variera mellan organismer, men inbegriper i princip endast 20 aminosyror. Genom att hitta par av tRNA och AARS som är ortogonala, dvs. inte interagerar med cellmaskineriet i en viss organism, kan man utöka den genetiska koden i denna organism med en extra symbol. Denna symbol kan vara en modifierad aminosyra, till exempel märkt med en radioaktiv isotop, eller potentiellt mer komplexa komponenter av nanomaskiner, som sedan kan sättas ihop av cellens ribosomer.

Hittills har mycket få ortogonala par publicerats, eftersom det kräver djup detaljerad kunskap om målorganismens biokemi för att hitta dem. I denna studie var målet att undersöka några möjliga metoder för att snabba upp denna process genom att på bioinformatisk väg hitta sannolikt ortogonala kandidater bland tRNA-gener i arvsmassan från sekvenserade organismer. I studien benämns dessa potentiellt ortogonala tRNA “Stealth tRNAs”.

Examensarbete 30 hp

Civilingenjörsprogrammet Bioinformatik

Uppsala universitet, maj 2015

Table of Contents

Introduction ***7***

Data & preprocessing ***9***

Selecting example sequences 9

Preprocessing 11

Notes on nomenclature 11

TRNA and Operon DataBase (TROPDB) 11

Methods ***12***

TFAM 12

HMM 13

Function logo information plots 14

SVM 15

ROSETTA 16

Results ***17***

TFAM 18

HMM 18

Function logo information plots 18

SVM 18

ROSETTA 18

Discussion ***19***

TFAM: hampered by excessive abstraction? 19

HMM: unsuitable for distinguishing highly-conserved sequences? 19

SVM: incompatible with discrete data? 20

ROSETTA: partial success and great promise 20

Function logo information plots: partial success and unexpected patterns 20

Acknowledgements ***21***

References ***22***

Appendix 1: Stealth tRNA Assessment Pipeline ***23***

Appendix 2: Flogiston User's Guide ***25***

Appendix 3: Flogiston Source Code (flogiston.pl) ***26***

Appendix 4: tRNAscan-SE Output Processing Script (tse2fa.pl) ***42***

Introduction

For all the obvious diversity among the living organisms on this planet, there are many basic and essential components that are very similar throughout the Tree of Life. The translation mechanism, that is, the translation of informational messenger RNA (mRNA) into proteins, is one such component. All organisms have coding genes that are transcribed to mRNA, which is read by the ribosome, matching a cognate transport RNA (tRNA) to each trinucleotide codon.

In many areas of life science, the ability to alter these basic mechanisms could be very useful to research and development.[1,2] In studies of protein folding, for example, it may be useful to selectively replace certain amino acid residues with radioactively labeled ones, or with a subtly altered variant that changes the protein's shape. In synthetic biology, the ability to modify or expand the genetic code in an organism could be useful both to elucidate the workings of natural organisms, and to engineer complex subcellular structures using the cell's own protein production line.

Orthogonal pairs are perhaps the most important tool[3] for manipulating the genetic code,consisting of a tRNA and its associated aminoacylation enzyme. They must be modified from the host organism's own translation machinery, or more commonly, imported from another, preferably genetically distant organism. Currently they are not easy to find, and very few orthogonal pairs have been documented.

The genetic code of most organisms uses trinucleotide codons. This means that there are 64 (4·4·4) possible codons, each corresponding to a certain elongator tRNA class, a start or a stop signal. The different tRNA species are each charged with one of typically 20 amino acids, which are assembled by the ribosome into the organism's proteins. There are normally more tRNA species than amino acids, and the collection of tRNA species associated with a certain amino acid is referred to as a "tRNA functional class".

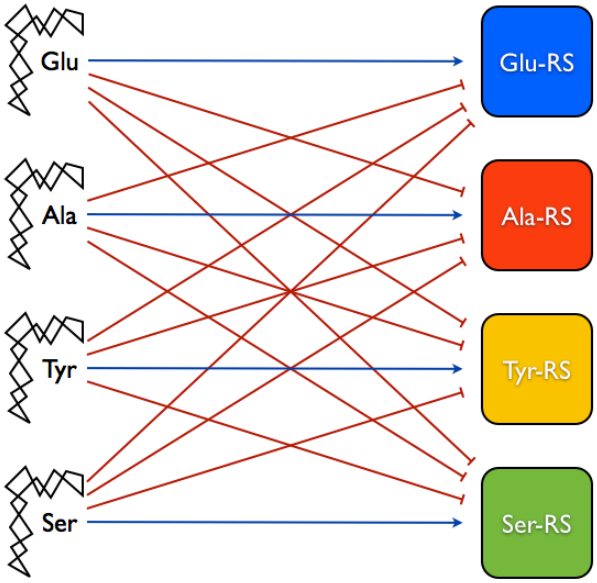

Figure 1: Sketch of tRNA recognition by different AARSs. In order for protein translation to function with any degree of accuracy, specific tRNAs must be charged with specific amino acids. Each tRNA species has certain identity elements that either promote recognition (blue arrows) or inhibit recognition (red T-arrows) by different aminoacyl-tRNA synthetases.

The task of charging each elongator tRNA with its assigned amino acid falls to the aminoacyl-tRNA synthetases (AARSs). There is at least one for each tRNA class, which specifically binds the appropriate tRNAs and attaches the appropriate amino acid.

In order to ensure that proteins are assembled correctly, the AARS must bind only the right tRNA species. Certain features of the tRNA molecules cause them to be either recognized or rejected by different AARSs (Fig.1). Some studies have been conducted into the mechanisms behind this specific recognition[4,5], but knowledge of these recognition elements has yet to reach a point where a scientist can deduce the potential interactions of a tRNA directly from its sequence.

Figure 2: Sketch of sequence alignment and resulting Profile Matrix. The analysis software created during this project made frequent use of Profile Matrices, recording the number of occurrences of each DNA sequence character (including gaps, "-") at each position in a multiple alignment of all sequences involved. The top graph illustrates an example alignment of five DNA sequences, resulting in a gapped alignment of length 10. The lower graph shows the 5x10 Profile Matrix P for that alignment; entry $P_{i,j}$ is the number of sequences with character i at position j in the alignment.

An orthogonal tRNA is one that is in working order - it is expressed and folded correctly, and could be used in translation - but is not recognised by any AARS in the organism. If it is not recognised by any AARS it does not get aminoacylated, and does not perform any constructive function in protein expression.

If, on the other hand, an orthogonal tRNA is engineered into an organism together with its cognate AARS - and provided that AARS is also non-interacting with native tRNAs - they form an orthogonal pair. This pair can act as a new aminoacylation pathway, separate from the native ones. If the host organism's genome is altered so as to leave a codon "vacant", and the orthogonal tRNA is allocated that codon, it becomes possible to change which amino acid corresponds to that codon. This effectively changes the genetic code, changing the nucleic acid-to-amino acid dictionary the ribosome uses to translate RNA into proteins. If the orthogonal pair replaces one codon for a degenerate tRNA class (one with multiple associated codons), the genetic code can be expanded beyond its usual 20 classes, for example adding some exotic amino acid to the alphabet[6] - or potentially any small molecule that can be attached to a tRNA and connected to a nascent polypeptide chain.

To date, the conventional methods for finding orthogonal tRNA-AARS pairs are heavily based on experiments *in vivo*[7,8], transfecting tRNAs from other organisms into the model and testing for interaction with the native translation machinery. There is not at all much information available on the principles of

tRNA recognition by AARS's, so potential orthogonal pairs must be found through close familiarity with both model organisms and the source organism for the orthogonal pair.

The purpose of this study is to explore some options for finding candidate orthogonal tRNAs by mining genomic sequence and publicly available annotations. If a tool could be programmed that screens the tRNA-ome of source organisms and suggests tRNAs that may escape recognition in the chosen model organism, that would surely be helpful in finding more verified orthogonal pairs. More such pairs could provide more tools and venues for studies into synthetic biology and expanding the genetic code, as well as more data for exploring the mechanics of tRNA-AARS interactions.

In this report I chose to call the candidate orthogonal tRNAs "Stealth tRNAs", in order to emphasize the differences. True orthogonal tRNAs must be verified experimentally, and are mainly useful in conjunction with an orthogonal AARS. Stealth tRNAs on the other hand are tDNA sequences that show weak recognition signals and/or strong antirecognition signals that *suggest* that they *may* be orthogonal. The problem of finding "Stealth AARS's" that would be required to use the Stealth tRNAs is outside the scope of this study.

Over the course of the project I focused on five different approaches to separating non-interacting tRNAs from interacting ones: separation by TFAM[9] score, Hidden Markov Models[10], Support Vector Machines[11], function logo[4] information plots, and Rough Set classification using ROSETTA[12]. Software for training of Hidden Markov Models appeared difficult to adapt to the problem at hand, so that approach was abandoned before practical implementation for the benefit of the other approaches.

In all implementations of the remaining approaches, the sequences from known orthogonal tRNAs were used as "positive control" samples. Most of the currently known orthogonal pairs were established in the bacterium *Escherichia coli*, which is why *E. coli* was most often chosen as the target organism.

Support Vector Machines also encountered problems with how to present tRNA data in a form required by the software, which effectively prohibited implementation of the SVM method.

TFAM scores were easy to obtain and use, since TFAM was used in preprocessing stages for sequence alignment and supplemental functional classification. However, attempts to find a clear discriminator between interacting and orthogonal tRNAs were unsuccessful.

Using function logos and inverse function logos as scoring matrices of a sort, and plotting the "total inverse function information value" of a tRNA versus its "total function information value", some scatterplots showed orthogonal tRNAs grouping separate from indigenous target tRNAs.

Rough Set classification in ROSETTA also showed promise. Classification rules trained on the *E. coli* tRNA-ome managed to avoid grouping known orthogonal tRNAs with any indigenous functional class.

Data & preprocessing

Selecting example sequences

For the purpose of detecting Stealth tRNAs, it will be necessary to consider the sequence and structure of tRNAs that belong to confirmed orthogonal pairs. At the time of writing, the selection of known orthogonal sets was very limited, and the number of targets for those orthogonal sets was even smaller. Although a few orthogonal tRNAs are known for the mammal *H. sapiens* and the fungus *S.*

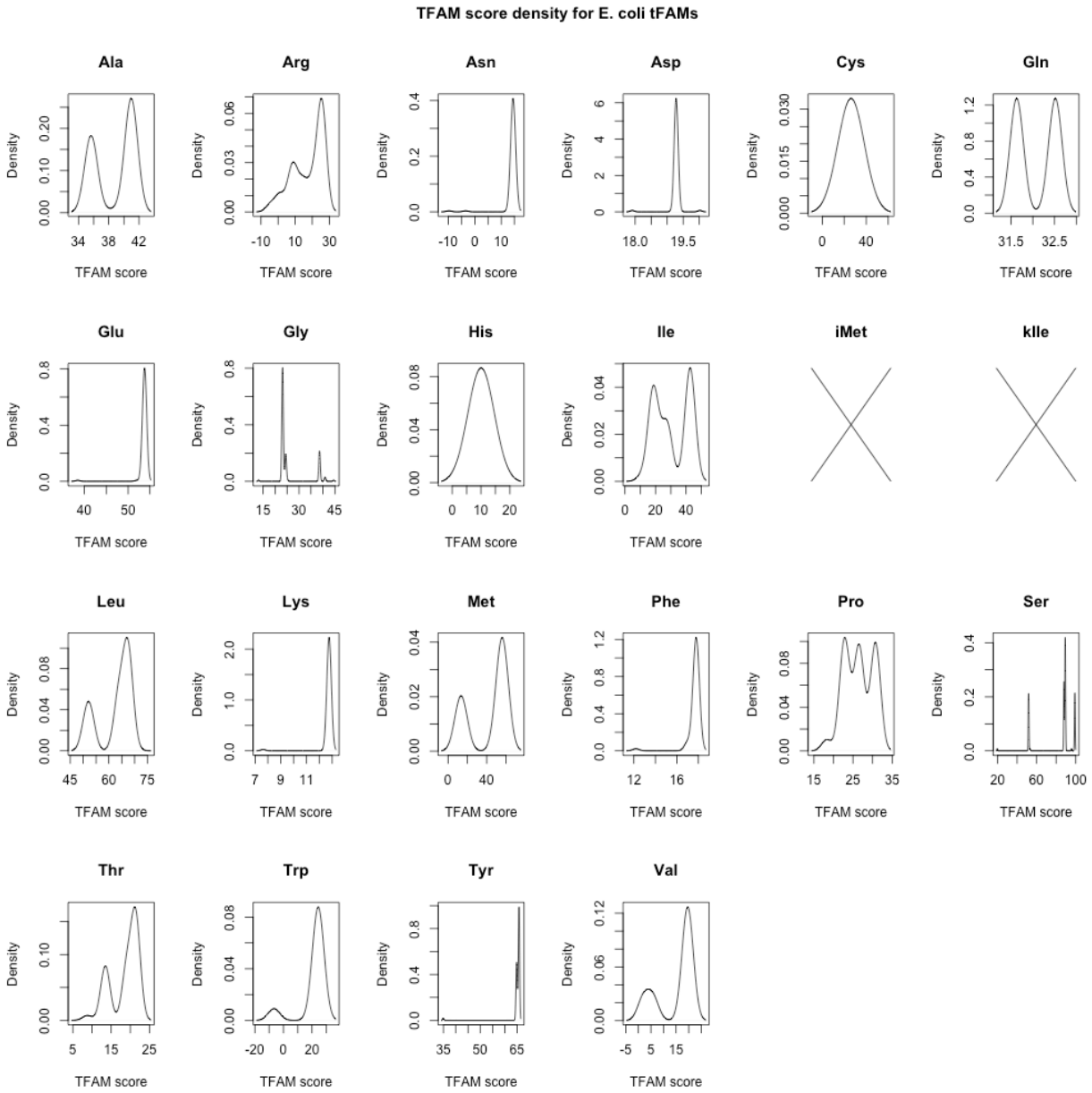

Figure 3: Density plots of TFAM scores for indigenous *E. coli* tRNAs. The plots above show, for each *E. coli* identity class, the density function calculated from tDNAs matching that class; they hint at the distribution of scores for true positive hits against the tFAMs in *E. coli*. The iMet and kIle plots are crossed out since, for this particular set of tDNAs and TFAM parameters, no tDNAs were assigned to those classes. It is interesting to see that what constitutes a "passing grade" varies strongly between identity classes. For the purpose of Stealth tRNA identification, this may mean that scoring requires separate approaches for every identity class.

cerevisiae, most have been determined for *E. coli*[2]. For all these targets, archaea seem to be the preferred source realm for Stealth tRNA candidates. This makes intuitive sense, since archaea are evolutionarily distinct[13] from the other realms of life, and therefore more likely to possess tRNAs that are sufficiently dissimilar in sequence to display functional orthogonality.

In fact, a previous study has shown that a $tRNA^{Tyr}$ - tyrosyl-tRNA-synthetase pair from the archaeon *Methanococcus jannaschii* can be used to generate orthogonal pairs in *E. coli*[8]. That made *M. jannaschii* $tRNA^{Tyr}$ a natural choice for a "positive control" - a foreign

tRNA that has previously been proven to work as part of an orthogonal pair in a model organism. If the Stealth tRNA detection algorithm can consistently detect "positive controls" like *M. jannaschii*-tRNA^{Tyr} for our model organism, then it might also be able to detect novel candidate Stealth tRNAs. Actually verifying the orthogonality of a putative Stealth tRNA is, however, well outside the scope of this project. Currently orthogonal pairs can only be confirmed through *in vivo* methods.

As previously mentioned, there appear to be few documented orthogonal pairs, but they do exist. A number of them are listed in a paper by Xie and Schultz[2]. The orthogonal tRNA-synthetase pairs for use in *E. coli* mentioned therein (all derived from archaea) include a TyrRS-tRNA^{Tyr} pair from *Methanococcus jannaschii*, LysRS-tRNA^{Tyr} from *Pyrococcus horikoshii*, GluRS-tRNA^{Glu} from *Methanosarcina mazei* as well as the heterogenous pairing of a LeuRS from *Methanobacterium thermoautotrophicum* and a mutant tRNA^{Leu} from *Halobacterium sp*. For use in yeast, the article mentions a TyrRS-tRNA^{Tyr} pair from *E. coli*, a LeuRS-tRNA^{Leu} pair also from *E. coli*, as well as *E. coli* GlnRS paired with human initiator tRNA.

Preprocessing

Each of these selected genomes were downloaded in .fna (FASTA) format from the NCBI FTP server. To extract the tDNA sequences from these genomes, tRNAscan-SE[14] (tSE) was run on each file. The resulting tDNA gene records were also preprocessed by condensing the FASTA sequence headers to a shorter unique identifier, free of whitespace characters. This was sometimes necessary since output from some programs later in the process tends to truncate long sequence names. In the worst case this can lead to sequences being unidentifiable after analysis. The preprocessing Perl scripts were designed to output a sequence legend file that shows the full header of the original tSE output alongside the new short-form header. The new headers also contain the tRNA functional class designation as provided by tSE.

The format used internally in the main script package in this project was "`>TAG_XXX-Y-ZZZZZZ`". `TAG` is either "`TGT`" for "Target" or "`QRY`" for "Query", stating the purpose of the tDNA in the current study (see the following subsection "Notes on nomenclature"). `XXX` is the anticodon in the tDNA, and `Y` is the single-character tRNA class identity, as respectively identified by tSE. `ZZZZZZ` is a six-digit integer, identifying tDNAs in the order that they are encountered by the scripts. This means that the script software is currently limited to 10^6 - 1 tDNA sequences each in the Target and Query sets.

Notes on nomenclature

Throughout the method development, implementation and testing process, I used a simple nomenclature to separate the sequence sets used. In my code, and in the following sections of this report, I use the terms "Target" and "Query". "Target organism" denotes the organism currently selected as host for the potential orthogonal pair. This is the organism whose "Target tRNAs" are identified with "Target classes" which an orthogonal tRNA should evade. The "Query organisms" are those selected to provide "Query tRNAs" to be tested for "stealthiness".

TRNA and Operon DataBase (TROPDB)

The Ardell lab has previously developed a Perl-based pipeline for detecting genomic features and storing them in a MySQL database for easy access and use by other bioinformatics applications. This is called the tRNA and Operon Database (TROPDB).

For large-scale bioinformatics studies, TROPDB can be used as a unified and uniform repository for sequence and annotations, stored on a local server or even an internal (and sufficiently large) hard drive.

For this study and others, it offered the possibility to easily share datasets, compare results and feed new annotations and knowledge back into the database. It was intended to have all software produced in this project integrated with TROPDB.

However, some difficulties quickly arose that ultimately led to the integration plans being abandoned in order to focus on exploration of the actual methods. The main problem was that TROPDB was programmed to import genome sequence and annotations in GENBANK[15] format. This format is mainly used in NCBI's GenBank database, which meant that tying the new software to TROPDB would limit its data sources to NCBI only, at least until a new import method could be designed. A conversion script for reformatting other sequence and annotation files to GENBANK was sketched, but not fully implemented.

Methods

TFAM

TFAM is a perl script application that uses alignment by covariance models to establish the functional class identity of tRNAs. TFAM takes its name from its product. Ardell & Andersson[2] coined the term "tFAM" to describe a family of logical rules that determine the charging identity of a tRNA, analogous to how a pFAM characterizes a family of proteins.

The tFAMs are created from multiple alignments of tDNA sequences from the model organism. The entire sequence set is initially aligned using COVEMF. For each functional identity class, the aligned sequences are separated into a 'positive' set of tDNAs belonging to the class, and a 'negative' set containing the complement - all tDNAs of other classes.

For each class, a "tFAM matrix" is then generated. At each position in the alignment, the total presence of each DNA base (A, C, G, T), as well as gaps (-), is counted. Fig. 2 shows an example of the process of recording such counts. Note that the example in the figure is not a finished tFAM matrix, but a "profile matrix", which was used for other methods in this study.

From these counts TFAM calculates the odds of encountering that character, at that position, in a tDNA belonging to the current class versus any other class (count in positive set divided by count in negative set), and takes the logarithm of the odds. The resulting log-odds are recorded in a 5xL matrix (one row per base plus gap characters, and L columns where L is the length of the multiple alignment).

TFAM scores test tDNAs against these matrices by stepping through the sequence and summing the log-odds values for the encountered base at each position. Matching the positive consensus sequence will give a stronger positive contribution to the score at positions where the matched character is more strongly related to the positive set - i.e., where the odds versus finding that character in the negative set are better than average. Conversely, at positions where the odds for the matched character are bad, it gives a negative contribution to the score, and a weak contribution where the odds are average.

The end result of this process is, for each tRNA in the input, one score (called TFAM score in the following) against each tRNA functional class, and a class prediction chosen as the highest-scoring functional class.

Using TFAM was a natural choice for several reasons. Partly because of the lab's familiarity with the software, and because TFAM can identify some special cases of functional

classes[2], including for example initiatior tRNAs. The output also automatically includes multiple alignments of the sequences involved, which are useful for many approaches.

Since TFAM employs sequence profiles of each tRNA class to score test sequences, those scores do, to some degree, reflect a scored tRNA's level of similarity to tRNAs of a given class. This similarity measure might be enough to establish a discriminator that can detect possible orthogonal tRNAs.

A simple way to test this is to perform TFAM classification of query and target tRNAs against the target organism. A query tRNA that scores lower than all target tRNAs against all charging identities present in the target organism may be a candidate orthogonal tRNA. Fig. 3 shows density plots of the TFAM scores for different classes of *E. coli* tRNAs. Different classes appear to have distinct and complex profiles, meaning that selecting a proper cutoff may be difficult, and highly class-specific.

HMM

Hidden Markov Models are a well-established type of statistical model that can be trained on pre-existing sequential data to recognize and classify new data. In bioinformatics, HMMs have long been used to find biological sequences - DNA, RNA and protein - matching certain patterns[10]. By assuming that sequences of the targeted family are produced as emissions of a Markov process, and training the model with positive examples, HMMs can be made to detect members of the targeted family with great accuracy.

There are many HMM-based softwares available, for nucleic acid or amino acid sequences. Any particular HMM software is typically designed with a given task in mind, such as recognizing proteins of a certain family, aligning sequences to a reference, or finding tRNA genes in genomic sequence.

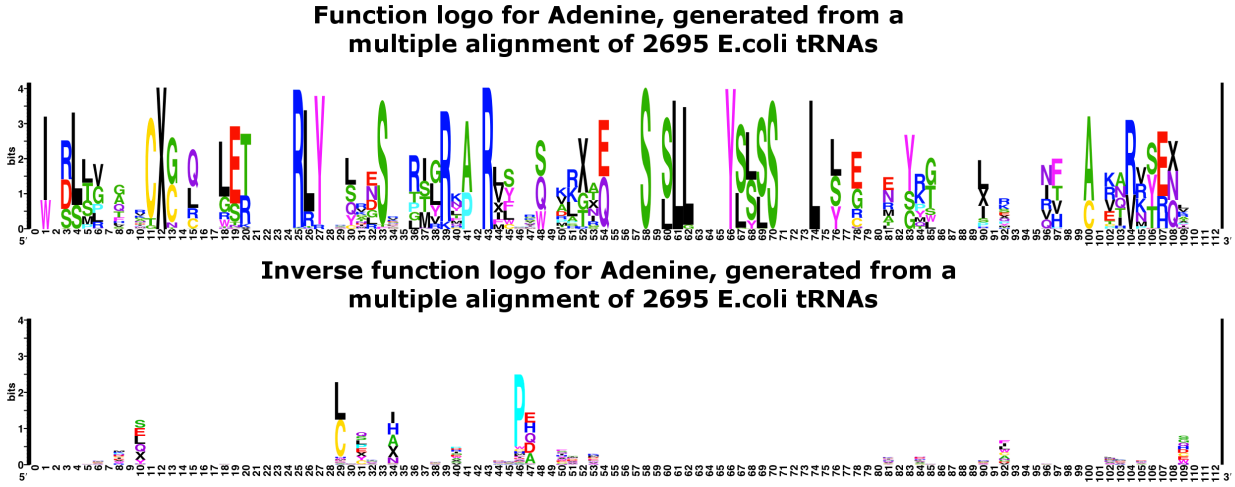

Figure 4: Example tRNA function logo and inverse function logo. The logofun software package produces function logos like the above examples from profile matrix information. These logos were generated from the profile matrices for 22 tRNA classes, which were in turn generated based on a multiple alignment (length 113 bases) of 2695 *E. coli* tDNA sequences extracted from 70 *E. coli* genomes downloaded from NCBI. The topmost function logo shows the functional identity information provided by an A (adenine) character at each position in the alignment. The letters show which functional identity is supported by the presence of an A, and the letter height indicates the strength of the identity signal. The graph below is an inverse function logo, which instead indicates the information provided *against* each identity class by the presence of an A at each position. In summary, the top *function* logo indicates where in the alignment and how strongly an A is a *determinant for* different identity classes; and the bottom *inverse function* logo indicates where in the alignment and how strongly an A is an *antideterminant against* different identity classes.

If a HMM training software can be adapted for tDNA functional classification, one can train a model to recognize tDNAs belonging to the identity classes of a given target organism. Since it is as yet unclear, and likely very case-specific what exactly makes an orthogonal tRNA orthogonal, it might be best to use target tDNAs as positive examples and try to find Stealth tDNAs by what they do not match.

To accomplish this, one could conceivably train one HMM for each functional identity class using target tDNAs. Query tDNAs can then be scored against each HMM, resulting in one emission probability for each model. The "stealthiness" of each query sequence would then be judged by the number of failed matches - a Stealth tRNA should ideally be a poor match for each class.

Function logo information plots

When tRNA sequences are run through TFAM, the output gives each sequence a score against each tRNA class in the model. Each score is a single real value based on its log-odds scores versus the class's positive and negative tFAM matrices. Since matches to the positive matrix give a positive contribution and matches against the negative matrix make a negative contribution, a tRNA that matches the tFAM for class X better than anything else will get a high positive against class X; conversely, a sequence that matches some other class better, or none at all, will get a strong negative score. Intuitively, tRNAs that carry no signal - positive nor negative - for class X should get scores closer to 0 by randomly matching both positive and negative.

When considering what these matches imply, some new questions arise. Could one sequence base with a strong negative signal be enough to completely disqualify a tRNA from class X? If the tRNA has this signal, can it be drowned out by sufficiently many weak positive signals? Do documented determinants and antideterminants for class X actually give stronger contributions than less-informative positions?

Logically, a tRNA matching class X should contain either more positive information for class X, or more negative information against every other class. The tRNA should either be actively selected by the AARS for class X, or rejected by every other AARS. A putative Stealth tRNA should contain as little positive information as possible for *all* classes, and preferably much negative information as well.

Logofun is a piece of software that produces "function logos"[4] from alignments of peptide- or amino acid sequences. Similarly to TFAM, it gathers character counts along the alignment. These character counts are recalculated into information values. Fig. 4 shows example function and inverse function logos, calculated from a set of 2695 *E. coli* tRNAs, for adenine.

The input is a series of profile matrices, one for each tRNA functional class to be studied. The output is one logo graph for each sequence character in the alignment - A,C,G,T and -. For graph A, the letter height of character S at position 51 can be roughly interpreted as "the signal strength for identification by a Ser-RS carried by an adenine residue at alignment position 51".

Logofun can also generate inverse function logos, which are constructed similarly to the regular variety, but the letter heights indicate information speaking *against* classification with the corresponding class.

It may be possible to find some way to discriminate between Stealth tRNAs and interacting tRNAs using the information values stored in function and inverse logos generated from a target organism's tDNAs. The main approach tested was to use the logos as a form of scoring matrices, summing the function logo information values for a tDNA, likewise summing the information values from the

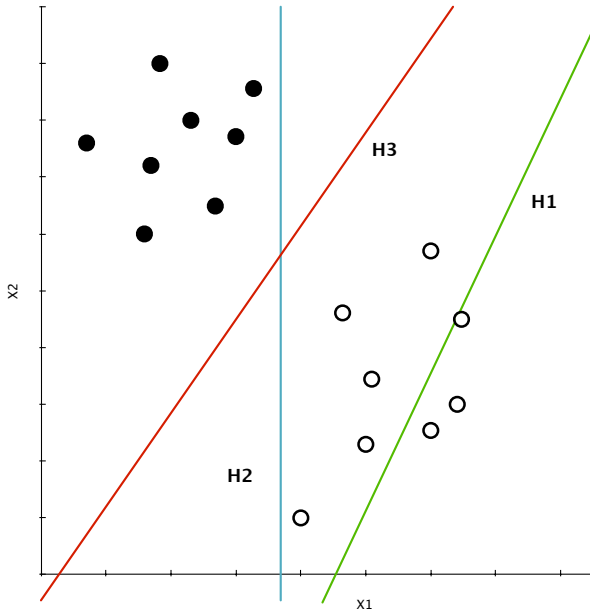

Figure 5: Sketch of SVM partition of 2D samples. The graph exemplifies the behaviour of a Support Vector Machine that is given a set of samples with two real-valued attributes: *x1* and *x2*. The hollow circles represent samples marked negative, and the filled circles represent samples marked positive. The SVM algorithm will propose an initial 1D discriminator (line H1) and determine if the samples are separated. They are not, so the discriminator is adjusted (H2) and evaluated again. The samples are now successfully separated, but the discriminator can be improved further. The algorithm iteratively refines the discriminator until the sum distance of the sample sets to the discriminator reach a maximum (H3). The separating hyperplane (in this case, line) should now be able to classify new data points as positive or negative with an optimal margin.

corresponding inverse function logos, and plotting the latter information total versus the former.

SVM

Any classification procedure could be generally described as an attempt to draw boundaries around and between the different categories in the given parameter space. Support vector machines[11] (SVMs) approach this quite directly by constructing a hyperplane that separates the samples of two different classes in a training set. Where a line would be enough to separate two sets of points that have two coordinates, you will need a hyperplane of n - 1 dimensions to separate two sets of points in a n-dimensional attribute space. Figure 5 provides a sketch of a simple SVM classification of 2D data.

In order to classify tRNAs using a SVM, we would represent the molecules as vectors with length on the order of 75-120, with each element corresponding to the nucleotide present at a consensus position in a tRNA multiple alignment.

It is important to note that SVM implementations are normally designed to work with samples that have real-valued attributes. This does not mesh well with the discrete nature of base sequence data, so in order to attempt SVM-based classification of Stealth tRNA candidates, some layer of abstraction is necessary to somehow describe a tRNA in terms of a set of real values.

The simplest way to make a tDNA sequence numeric would be to simply assign a value to each base; A - 1.0, T - 2.0, G - 3.0 and C - 4.0. However, putting all the bases on the same continuous axis may cause problems. Consider, for example, if the SVM algorithm generates, for a certain alignment position in a family of tRNAs, the cutoff value 2.35. tRNAs with A or T at the position get a positive signal, and those with G or C get a negative signal. But what does that mean, biochemically? If the positive training set that generated this value had mostly T and a few G at the position, we may now get false positives with A and miss true positives with G at this alignment position. Also, any position with small differences between the counts will receive a cutoff around the middle of the range, so that tested sequences will be arbitrarily scored positive or negative when that position should actually carry very little information at all.

Reducing the choice to a [0, 1] scale with purine residues and pyrimidine residues scored at opposite extremes might make the labeling and cutoff make marginally more biochemical sense, but some fidelity is lost. In addition,

since many of the nucleobases in the tRNA are exposed, they may be involved in recognition, and thus the exact base identity is likely important for orthogonality.

ROSETTA

In rough set analysis and boolean reasoning, information systems and decision tables are used to classify samples with a number of measured attributes into given decision classes. The data samples in an information system all have the same attributes, but individual samples may lack values for any attribute. Among the strengths of the rough set and boolean reasoning approaches is that they can be implemented with a high tolerance against missing data.

The information system may be presented in a table, with each attribute as a separate column. A decision system is an information system with a decision attribute appended to the sample vectors. The decision attribute contains the classification of the samples, and is necessary to construct rules that can determine the classification of new samples.

There are various algorithms available that can reduce the attribute set of a decision system to reducts: a minimal set of attributes needed to separate samples of the different classes (without necessarily preserving the discernibility of different samples within a class). From such reducts, one can generate boolean rules that classify samples based on their values for the reduced attribute set.

ROSETTA[12] is a toolkit for rough set analysis developed by A. Øhrn in the late '90s. It provides a versatile environment for training various types of classification rules on datasets (in table form) and classifying samples based on the rules generated. It can be used to create classification pipelines for continuous data, but that data needs to be discretized before creating rules. Luckily, this is not necessary for nucleotide sequences, which are (in most interpretations) discrete by nature. On the other hand they must be presented in a tabular form that makes sense for further classification.

Since this study uses discrete data (tDNA sequence) to perform supervised classification with discrete labels (tRNA functional classes), the problem is ideally suited for boolean reasoning approaches.

After producing multiple alignments of all tRNAs in the study, the PERL implementation of this method produces ROSETTA-readable CSV tables from the alignment. Each row represents one tRNA, and each position in the alignment has its own column. In addition, the TFAM-determined charging identity of the tRNA is recorded in the final column. This serves as the decision attribute in constructing classification rules.

After the decision system was loaded into ROSETTA, the data was first separated into target and query sets by sequence header. The target set was randomly split 80-20 into a training and testing set. From the training set, reducts were generated using Johnson's algorithm and a genetic algorithm, in both cases using default parameters.

After generating classification rules from the reducts, the testing set could be classified with those rules in order to test their sensitivity and specificity. ROSETTA provides a confusion matrix showing predicted class versus actual class. In the confusion matrix for the testing set, with a perfectly performing set of rules, there should only be entries on the diagonal - meaning ROSETTA's predictions always match the input.

Entries anywhere other than on the diagonal means that the tDNA has been misclassified. If a tDNA matches none of the generated rules, it will remain unclassified - and that is how we may find potential Stealth tRNAs.

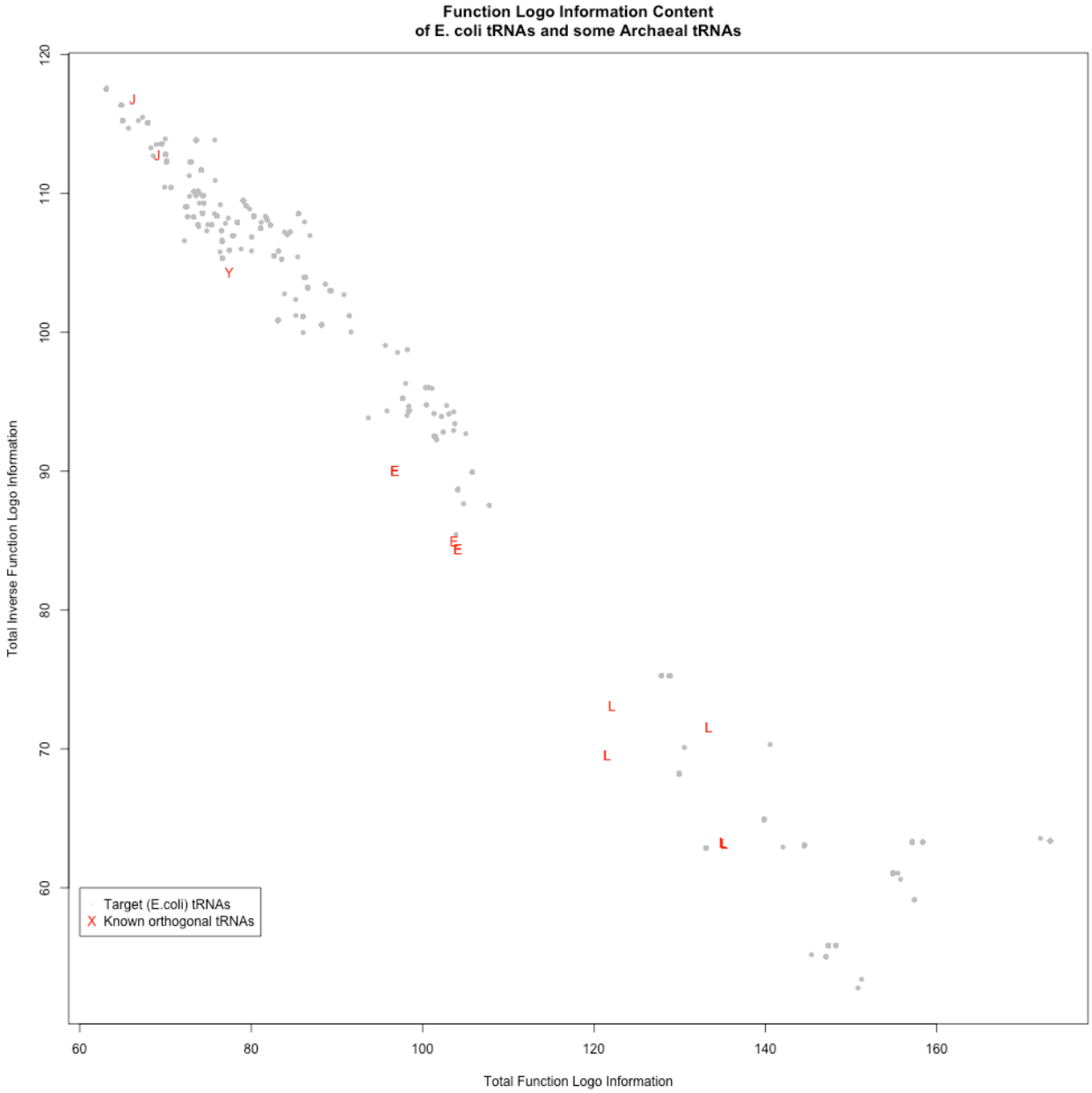

Figure 6: Example Function Logo Information Plot. The graph above shows the total inverse function logo information content of tDNAs versus the total function logo information. These totals were calculated by treating function logos and inverse function logos as a form of scoring matrices and taking the sum of total stack heights, at each position, for each character in the sequence (including gaps). Class-specific letter height was ignored, taking only the total stack height for the given position from the logo corresponding to the given character. The gray dots represent 2695 *E. coli* tDNA sequences; the red letters represent a set of tDNAs known to be orthogonal in *E. coli.* Note that some, particularly the "E" orthogonal tDNAs plot slightly outside of the "cloud" of target tDNAs. Also interesting is that all tDNAs appear clustered around a negative diagonal line, hinting that the sum of the total functional information and inverse functional information in a tDNA may be near-constant. Whether this is an artifact of the logo generation process is unknown.

Results

To the greatest possible extent, preprocessing and analysis steps were automated in a Perl driver script with the working title "flogiston" (Appendix 3; pipeline specification and flogiston instructions are included in Appendices 1 & 2, respectively). This includes processing and organizing input and output files, and running the various other programs required for analysis. In order to leave users more freedom to construct their tDNA datasets, the preprocessing steps of extracting tDNA

sequences with tRNAscan-SE and sorting them by taxa etc. was left out of the pipeline.

TFAM

Using TFAM for classification was an attractive option, because of the lab's familiarity with the tool and the pre-existing code. However, with further study, it became apparent that the TFAM scores may abstract recognition signals too much to be of use; boiling down the contributions of the entire tRNA sequence into a single score discards much potentially relevant information.

As no heuristic could be established to find the cutoff between interacting and non-interacting tRNAs, this approach was abandoned.

HMM

Although the popularity and successful history of HMMs made their use an obvious candidate for Stealth tRNA detection, their sequence-specific and data-driven nature made them less useful in practice.

In this project, the objective was to find or construct a tool that can detect potential orthogonal tRNAs by sequence alone. However, it appeared necessary to locate and study features of the query tRNA's sequence in ways that are not best done by Markov modeling. Designing such a combined-signal HMM is regrettably beyond the author's ability. The HMM approach was therefore abandoned for the benefit of other methods.

Function logo information plots

Some of the scatter graphs generated by plotting function logo information versus inverse logo information for indigenous tRNAs and known orthogonal tRNAs showed great promise. Figure 6 shows an example of this. When plotting the information values for tRNAs from *E. coli* and known orthogonal tRNAs, orthogonal $tRNA^{Glu}$ and $tRNA^{Leu}$ were noticeably separated from the *E. coli* "cloud".

An unexpected feature of these plots was the clear clustering of indigenous tRNAs around a diagonal, the slope of which indicates that the sum of function logo information and inverse function logo information for each tRNA is more or less constant in an organism's tRNA-ome.

SVM

Using state vector machines to separate putative stealth tRNAs from interacting tRNAs seemed like a sound approach, because of several success stories with binary discrimination. However, as explained in the Methods section, it is difficult to express a tRNA as a string of real values. As a result, SVM analysis was not fully implemented in the course of this study.

ROSETTA

Classification using ROSETTA went further than some other approaches. Rule sets trained on *E. coli* tDNAs notably failed to classify known orthogonal tDNAs. This is a good outcome, as Stealth tRNAs should remain unclassified. Other tDNAs from the same organisms as the orthogonal sequences were occasionally misclassified with some *E. coli* identity class, but were also generally unclassified. The Johnson algorithm worked very quickly but generated a single, very compact reduct. The genetic algorithm reducts could take much longer depending on sample sizes and parameters, but generated more and varying reducts.

For reasons that could not be determined, the ROC (Receiver Operating Characteristic, indicating the effectiveness of the discriminator) curves for these classifications versus *E. coli* rules suffered from strange errors. A recurring problem was that all ROC parameters - area under the curve, standard

error, thresholds - were assigned a placeholder value for “infinity”. This could be due to emulation errors. ROSETTA is a Windows-specific program, but was run in a virtual machine using Wine (on a Macintosh computer).

Discussion

TFAM: hampered by excessive abstraction?

Applying tFAMs to the task of detecting stealth tRNAs was ultimately unsuccessful. TFAM scores on indigenous tRNAs versus orthogonal queries did not show any obvious tendencies that might be used for detection of stealth tRNAs. It is likely that the TFAM algorithm, while useful for scoring tRNAs based on their positive recognition by a certain class of AARSs, abstract too much of the interaction signals by condensing them to a number.

This is analogous to how biologists recognize tRNAs versus how AARSs recognize them. Associating a tRNA with a certain amino acid gives a researcher a simple overview of the function and importance of that tRNA. A AARS enzyme on the other hand cannot analyze the entire sequence of a tRNA and compare it to libraries of similar sequences. Whether or not it treats a given tRNA as a substrate depends on any number of residue-level physical interactions which cannot be adequately summed up by a single score.

The TFAM score is also heavily dependent on the availability of data. As the score is in part calculated from the logarithm of number of observations for divided by number of observations against, the mere amount of sequences available for either side will affect the magnitude of the score in ways that are not easily normalized between tests.

For the task at hand, this data volume dependency is a serious problem, as very few orthogonal pairs are known. This is also specific to each model organism, and for any given organism, the number of known interacting tRNA sequences is very likely to grow much faster than the number of known orthogonal sequences, for the foreseeable future.

HMM: unsuitable for distinguishing highly-conserved sequences?

A HMM-based stealth tRNA detection method could not be established within the timeframe of this project. This was mainly due to difficulties in reconciling the efficient pattern recognition of HMMs with the strong conservation of tRNAs, and the fact that tRNA recognition signals are poorly characterized.

HMMs are very good at finding sequences that match the consensus training set - primary or secondary sequences, depending on the implementation - within margins also dictated by the variability within the training set. However, as part of the protein synthesis machinery, both the primary and secondary sequence of tRNAs are highly conserved, even between widely divergent taxa. This brings an additional set of challenges to the problem of implementing a stealth tRNA-targeting HMM.

If a stealth tRNA-finding HMM were trained on the primary sequence of known orthogonal tRNAs alone, the rarity of known orthogonal pairs would mean that there is very little data available with which to construct the model. Because of the high degree of sequence conservation in tRNAs, the resulting model would likely give good “stealthiness” scores to many non-orthogonal tRNAs.

If the model were trained on the host organism’s indigenous tRNAs instead, higher scores should indicate similarity to normally-interacting tRNAs, and lower scores might show that query tRNAs are non-interacting. However, the main problem in this approach is how to differentiate between a non-interacting stealth tRNA and a sequence that is simply not

functional as a tRNA. Again, as tRNA sequence is highly conserved, this type of model would essentially score high for query sequences that are likely to fold into viable tRNAs, and since the mechanics of recognition and anti-recognition are not well understood, it is hard to say whether a HMM would be able to properly represent those signals.

If HMMs trained on whole sequences are intrinsically too sequence-specific to effectively separate non-interacting tRNAs from the interacting, then a possible solution is to train a model on RNA structural features other than the primary base sequence. The effect on AARS recognition by certain features like the base present at a given coordinate, or the presence, absence or size of the variable arm, can be inferred from function logos and by other means. If these features could then be described by a HMM, in some combination of whole sequence matching and motif matching, disregarding uninformative regions of the tRNA, it might be possible to get scores that more clearly discriminate between stealth tRNAs and others.

However, this would require extensive modification of tRNA HMM structures. Normal HMM implementations recognize simple sequence signals of a single type - nucleotides or amino acids occurring in sequence. To enhance stealth tRNA detection, it may be necessary to combine base sequence with other signals that might be recognized by the AARS, such as steric qualities like shape and size of an arm, or whether a sequence region is hydrophobic or hydrophilic on the detectable surface. To train such a model, the software would need to record not just the nucleic acid symbols in order, but also detectable qualities of single bases or sequence regions of various sizes.

At the time of writing, no such HMM is publicly available. It is possible that normal HMM alignment to several interacting tRNA classes could be combined with other sequence annotation and analysis outside of the Markov model, but that is left as an exercise for future investigators.

SVM: incompatible with discrete data?

The main problem preventing implementation of a SVM stealth tRNA detection algorithm is that most publicly available SVM training software uses exclusively continuous, real-valued sample data. As mentioned in the Methods section, while it is trivial to simply translate RNA sequence to numbers, the meaning of those numbers runs a severe risk of being distorted by mathematical operations.

A way to incorporate discrete base sequences in SVM analysis could not be found during the run of this project. With no compatible data to train a state vector machine on, it was regrettably impossible even to include SVMs in a compounded analysis across different methods.

The power of SVMs in dataset compartmentalization is indisputable, but until a discrete-continuous hybrid SVM is introduced, their usefulness in tRNA sequence analysis is limited.

ROSETTA: partial success and great promise

Rough set approaches using ROSETTA appeared quite successful. Another potential advantage is that there are few preprocessing steps between raw tRNA sequence and classification - mostly alignment and reformatting. ROSETTA-based Stealth tRNA detection needs to be evaluated in greater detail, with larger tests and more varied parameters. In this study, Johnson reducts and boolean reasoning were used for rules generation; many options remain to be tested, and better classification performance seems very possible.

Function logo information plots: partial success and unexpected patterns

The function logo information plots were also interesting, particularly the unexpected fitting to a diagonal. It makes some intuitive sense that the information content is limited; with 22 possible classes the absolute maximum information content in bits should be roughly $\log_2(22) * L$, where L is the length of the sequence. It is less clear why that maximum information should be divided between functional and inverse functional signals. Future researchers more familiar with the mathematics of logo generation would be welcome to establish whether this is an artifact of the mathematics employed, or something that may have actual bioinformatic significance. Also, in order to make practical use of these plots for Stealth tRNA detection, some way to automatically isolate potential orthogonal tDNAs would be necessary, instead of analyzing each graph by eye.

It must be noted that all of these studies were done entirely from primary sequence data. More information could and should be integrated - secondary structure information to begin with, and interactions with relevant proteins if available. Generally, detailed interaction information is rare. A database of tRNA-protein interactions with standardized format, or a way to estimate interactions from sequence info, would be immensely useful.

It is also important to remember that bioinformatical approaches are unlikely to entirely replace laboratory methods. Ultimately, the best these *in silico* methods can do is suggest candidates for experimental verification. Orthogonal pairs can as yet only be established by *in vivo* tests.

Acknowledgements

This project was carried out over the period between September 6th, 2011 and March 6th, 2012, at Prof. David H. Ardell's lab at the University of California Merced campus.

I would like to thank Prof. David Ardell for offering me the excellent opportunity to do advanced bioinformatics research for my degree project, and for entrusting me as a green pseudo-graduate with entirely exploratory research.

Thanks to Julie Phillips and family for all the support on and off work; without you I would have been starving on the street for six months.

Thanks to Katie Harris and Wes Swingley for invaluable help with software and methods, as well as being great co-workers.

Thanks also to Prof. Suparna Sanyal at the Dept. of Molecular and Cell Biology, Uppsala University, for her help in reviewing this report; and to Lars-Göran Josefsson, student faculty coordinator at the Biology Education Centre, Uppsala University, for his great patience and helpfulness in managing my degree project.

Final thanks go to my family and my friends in Uppsala, for all your support and for inspiring and helping me to carry out my dream project halfway across the globe.

References

1 Kevin M. Esvelt, Harris H. Wang: **Genome-scale engineering for systems and synthetic biology,** Molecular Systems Biology, Vol. 9, No. 1. 22 January 2013

2 Jianming Xie, Peter G. Schultz: **Adding amino acids to the genetic repertoire**, Current Opinion in Chemical Biology, Volume 9, Issue 6, December 2005, pp. 548 - 554, ISSN 1367-5931

3 Qian Wang, Angela R. Parrish, Lei Wang: **Expanding the Genetic Code for Biological Studies**, Chemistry & biology, volume 16 issue 3, 27 March 2009, pp. 323 - 336

4 Eva Freyhult, Vincent Moulton, David H. Ardell: **Visualizing bacterial tRNA identity determinants and antideterminants using function logos and inverse function logos**, Nucleic Acids Research, Vol. 34, No. 3, 2006, pp. 905–916

5 Jing Yuan, Tasos Gogakos, Arianne M. Babina, Dieter Söll, Lennart Randau: **Change of tRNA identity leads to a divergent orthogonal histidyl-tRNA synthetase/tRNA^{His} pair,** Nucleic Acids Research, 2011, Vol. 39, No. 6, pp. 2286-2293

6 David R. Liu, Thomas J. Magliery, Miro Pastrnak, Peter G. Schultz: **Engineering a tRNA and aminoacyl-tRNA synthetase for the site-specific incorporation of unnatural amino acids into proteins *in vivo***, Proc. Natl. Acad. Sci. USA, Vol. 94, pp. 10092–10097, September 1997

7 Heinz Neumann, Adrian L. Slusarczyk, Jason W. Chin: **De Novo Generation of Mutually Orthogonal Aminoacyl-tRNA Synthetase/tRNA Pairs**, Journal of the American Chemical Society, Vol. 0, No. 0, 1 February 2010

8 Lei Wang, Peter G. Schultz: **A general approach for the generation of orthogonal tRNAs**, Chemistry & biology, Volume 8, issue 9, pp.883 - 890, September 2001

9 David H. Ardell, Siv G. E. Andersson: **TFAM detects co-evolution of tRNA identity rules with lateral transfer of histidyl-tRNA synthetase**, Nucleic Acids Research, 2006, Vol. 34, No. 3, pp.893–904

10 Sean R. Eddy: **Profile hidden Markov models**, Bioinformatics, Vol. 14, No. 9, 1 January 1998, pp. 755-763

11 Corinna Cortes, Vladimir Vapnik: **Support-Vector Networks Machine Learning**, Vol. 20, No. 3. 1 September 1995, pp. 273-297

12 Aleksander Øhrn, Jan Komorowski: **ROSETTA: A Rough Set Toolkit for Analysis of Data**, Proc. Third International Joint Conference on Information Sciences, Fifth International Workshop on Rough Sets and Soft Computing (RSSC'97), Durham, NC, USA, March 1-5, Vol. 3, pp. 403-407, 1997
See also: ROSETTA project homepage, http://www.lcb.uu.se/tools/rosetta/

13 Norman R. Pace: **Time for a change,** Nature, Vol. 441, No. 7091, 17 May 2006, pp. 289-289

14 Todd M. Lowe, Sean R. Eddy: **tRNAscan-SE: a program for improved detection of transfer RNA genes in genomic sequence**, Nucleic acids research, Vol. 25, No. 5, 1 March 1997, pp. 955-964

15 Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, David L. Wheeler: **GenBank,** Nucleic Acids Research, Vol. 33, No. suppl 1, 01 January 2005, pp. D34-D38

Appendix 1: Stealth tRNA Assessment Pipeline

Note on using the scripts provided:

The scripts provided in these appendices are all written in the Perl scripting language. They require a working installation of the Perl runtime of version 5.10.0 or later, and are executed from the command line using the syntax:

```
perl <script.pl> <options> <files and arguments>
```

Options are indicated with a dash and may be followed by an argument (e.g. -f out.fa). What filenames and arguments are required for the script to run is detailed in each script's usage information, accessed by executing

```
perl <script.pl> -h
```

1. Choose target organism and query organism(s).

Potentially any organism could be chosen. However, a more thoroughly studied target organism will bring with it more sequence annotations and experimental data that can be used to vet any candidate orthogonal tRNAs produced by this pipeline.

A good source for genomic data is ftp://ftp.ncbi.nlm.nih.gov/genomes/

2. Download genome sequences for all organisms.

Sequences must be in FASTA format to qualify as input for tRNAscan-SE. If your sequences are not readily available in this format, there are many tools available for format conversion. One online implementation is http://www.ebi.ac.uk/Tools/sfc/readseq/.

3. Extract all tRNA genes using tRNAscan-SE.

For this step, I generally used a tRNA model matching the target organism, e.g. built-in bacterial model for *E. coli*. This is set by option flag -B; an archaeal model is set by -A. See tRNAscan-SE manual or command-line help (-h) for more options.

IMPORTANT: tRNAscan-SE does not output FASTA-formatted sequences. To get those, use the option -f <filename>, which will save secondary structure predictions (including primare sequence) to the specified file. Then, use the tse2fa.pl script (Appendix 4) to convert this output to a multi-FASTA file of the detected tRNAs.

4. Tag sequences for later identification

5. Run flogiston.pl script on the tRNA gene multi-FASTA files.

See Appendix 2 for available options for the flogiston script.

6. ROSETTA analysis:

Please refer to the ROSETTA manual for detailed usage instructions and feature descriptions.

a. Open the the outputfile suffixed with ".rosetta"

Use the "Rosetta table import format" when prompted.

b. Separate target and query sequences

Target and query sequences will be co-aligned in one table. Right-click the table and select "Duplicate". Delete the query sequence rows from one table, and the target sequence rows from the other, to end up with separate target and query tables.

c. Generate reducts

Right-click your target sequence table and select any option under “Reduce”. Johnson’s algorithm qill generate a single, naïve reduct quickly. Other methods may take longer but may also give better reducts.

d. Generate rules

Right-click a reduct and select “Generate rules ...”. Several options are available, including the quick-and-dirty Johnson algorithm and a Genetic Algorithm, which will take longer but will typically give more discriminating rules.

e. Classify query sequences

Right-click your query sequence table and select “Classify ...”. Check the “Log individual results to file” option in the dialog that appears, and input a file path where you want the classifications to be saved. (Without this option, you will only see the classification statistics for the whole dataset, i.e. *if* there are candidate orthogonal tRNAs, but not *which* sequences are candidates.)

Click “Parameters ...” in the Classifier box to show a dialog where you can select the classification rule set to be used, among other parameters. Any rules you have created should be available in the drop-down list.

Double-clicking the classification you just generated will show you a confusion matrix. Rows represent the “actual” tRNA class (as read from the .rosetta file). Columns represent the predicted class according to the rule set you generated. Entries on the diagonal will have been classified with the same class that TFAM gave them; i.e., they are likely interacting with the target’s AARS’s. Entries off the diagonal have been misclassified, and may be of interest. Most interesting are the entries in the “Undefined” column, those that were not given a classification by the generated rules. These could be considered Stealth tRNAs, and should be further studied to assess their orthogonality.

The reader is encouraged to read up on and test the effects of the many, many options and parameters that ROSETTA offers. The program also allows batch scripting, meaning that this process could be partially or entirely automated.

Appendix 2: Flogiston User's Guide

Flogiston Command-line Help:

```
flogiston.pl: (F)unction (Log)o (I)nformation-based (S)tealth-(t)RNA detecti(ON) v. 0.3
Usage: perl flogiston.pl [Options] <target.fa> <query.fa> [<legend_filename>]

-h              Print this help and exit
-t <str>        Set prefix tag for this project (default "new_")
-c              Output tRNAs' scores vs target functional classes to "<prefix>_clspec_scores"
-e              Use existing function logos & inverse logos
                (format: "<logo filename prefix>:<inverse logo filename prefix>")
-x #:# Exclude region in alignment from scoring
                (format: "a:b" excludes from position a to pos b)
                If the first two elements are 'save:info', info value for the excluded regions
                will be saved (e.g. "save:info:56:77")
-g              Score gaps (default NO)
-l              Score only for the largest signal (default NO)
-m [A/E]        Select TFAM tRNA model: A for archaeal, E for eukaryotic. Default bacterial.
-p              Score basepair function logos
-s <file>       Output all tRNA's scores vs Profile Information Matrix and Inverse ditto to <file>
-r              Refactor headers in input FASTA files
```

Requirements:

UNIX-like system (only tested on Apple Macintosh computers running Snow Leopard or later)

Perl v5.10.0

BioPerl v.1.6.910

TFAM v.1.3

logofun 1.0

bplogofun 0.3

The script is not guaranteed to work with other versions of these software dependencies.

Detailed Options:

-c	Functional class-specific function logo information scores will be output to <prefix>_clspec_scores
-e <file>:<file>	Can be used to skip the function logo generation step by using existing function logo files (in .eps format). Assumes Argument format: <function_logo_prefix>:<inverse_function_logo_prefix> You must have 10 logo files in total, with names of the format <function/inverse prefix>_<A,C,G,T or ->.eps
-g	Toggles gap scoring on. When this is off, the function logo information values for gaps in the alignment are ignored when calculating information scores.
-l	When this is on, only the largest functional signal in the function logo will be recorded for scoring, instead of the sum of all signals.
-m [A/E]	Specifies which tRNA recognition model TFAM should use: E for eukaryote, A for archaeal. If this option is left out, the default is bacterial.
-p	Experimental option using bplogofun instead of regular logofun. This generates logos for base-pairs in the RNA secondary structure as well, instead of just single nucleotides.
-r	Refactors the headers of all input tRNAs; target sequence headers will start with ">TGT" and query sequence headers with ">QRY". Useful if TFAM causes problems by truncating sequence headers. A header key will be saved to "<prefix>legend"
-s <file>	Outputs function logo information scores for all tested tRNAs to the specified filename.
-t <prefix>	All output filenames will be prefixed with this tag.
-x a:b	Exclude the region (in multiple-alignment positions) from function logo information-scoring. Several regions can be specified in sequence, i.e. "a:b:c:d ...". If the first pair reads "save:info", the information values of the excluded region will be saved to the file specified by option -s, under the column "Excluded".

Appendix 3: Flogiston Source Code (flogiston.pl)

```
#!/opt/local/bin/perl -w
## Copyright (C) 2011-2013 by Jan Ingemar Ohlsson
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# Contact information: ingotron (at) gmail (dot) com

use strict;
use Getopt::Std;
use Bio::AlignIO;
use Statistics::Descriptive;
use Time::localtime;
use Chart::Graph::Gnuplot qw(gnuplot);
use List::Util qw(max);
use subs 'timestamp', 'zeroes', 'log2';

###################
# Initialization #
###################
# Version 0.3
my $version = "0.3";
my @blick = split('/',$0);
my $name = pop(@blick);

# Help and instructions
my $help_string;
$help_string = "$name: (F)unction (Log)o (I)nformation-based (S)tealth-(t)RNA detecti(ON)
v. $version\n";
$help_string .= "Usage: perl $name [Options] <target.fa> <query.fa> [<legend_filename>]
\n";
$help_string .= "\n";
$help_string .= "-h\t\tPrint this help and exit\n";
$help_string .= "-t <str>\tSet prefix tag for this project (default \"new_\")\n";
$help_string .= "-c \t\tOutput tRNAs' scores vs target functional classes to
\"<prefix>_clspec_scores\"\n";
$help_string .= "-e \tUse existing function logos & inverse logos (format: \"<logo
filename prefix>:<inverse logo filename prefix>\")\n";
$help_string .= "-x #:#\tExclude region in alignment from scoring (format: \"a:b\"
excludes from position a to pos b)\n";
$help_string .= "\t\tIf the first two elements are 'save:info', info value for the
excluded regions will be saved (e.g. \"save:info:56:77\")\n";
$help_string .= "-g\t\tScore gaps (default NO)\n";
$help_string .= "-l\t\tScore only for the largest signal (default NO)\n";
$help_string .= "-m [A/E]\tSelect TFAM tRNA model: A for archaeal, E for eukaryotic.
Default bacterial.\n";
$help_string .= "-p\t\tScore basepair function logos\n";
$help_string .= "-s <file>\tOutput all tRNA's scores vs Profile Information Matrix and
Inverse ditto to <file>\n";
$help_string .= "-r\t\tRefactor headers in input FASTA files\n";
$help_string .= "\n";

my %Opts;
&getopts('ce:ghlm:prs:t:x:',\%Opts);
```

```
my $opt_h = $Opts{'h'}; # Print help and die
my $opt_t = ($Opts{'t'} ? $Opts{'t'} : "new_"); # Set project tag for labeling output
my $opt_c = $Opts{'c'}; # Output scores vs target's func classes
my $opt_e = $Opts{'e'}; # Use existing logos
my $opt_x = $Opts{'x'}; # Exclude this region in alingment from scoring - necessary to
know the alignment beforehand
my $opt_g = $Opts{'g'}; # Score gaps too
my $opt_l = $Opts{'l'}; # Score for largest signal only
my $opt_m = $Opts{'m'}; # TFAM tRNA model: A for archaeal, E for eukaryote; default
Bacterial
my $opt_p = $Opts{'p'}; # Gnuplot function info vs inv info
my $opt_r = $Opts{'r'}; # Refactor FASTA headers
my $opt_s = $Opts{'s'}; # Output qry & tgt scores vs info & inv info

if ($opt_m) {
	unless ($opt_m eq 'E' || $opt_m eq 'A') {
		$opt_m = 0;
	}
}

die $help_string unless @ARGV;
die $help_string if $opt_h;

# Check exclusion regions
my $xstart = 500;
my $xend = -1;
my @xlimits;
my $savexc = 0;
if ($opt_x) {
	@xlimits = split(':',$opt_x);
	if (scalar(@xlimits) % 2 != 0) {
		die "ERROR: Odd number of region delimiters in \"$opt_x\". Will not exclude!
\n";
	} elsif (scalar(@xlimits) == 0) {
		@xlimits = ($xstart, $xend);
	}
}


# If the prefix tag contains a folder, make sure it exists
my @pfxbits = split('/',$opt_t);
if (scalar(@pfxbits) > 1) {
	pop @pfxbits;
	my $path = join('/',@pfxbits);
	system "mkdir -p $path";
}

# Store AA names & codes and generate lookup tables; might need it when dealing with TFAM
output
my $aa_string = 'ACDEFGHIKLMNPQRSTVWXY';
my @aa_letters = split(//,$aa_string);
my @aa_abbrevs =
split(/,/,'Ala,Cys,Asp,Glu,Phe,Gly,His,Ile,Lys,Leu,Met,Asn,Pro,Gln,Arg,Ser,Thr,Val,Trp,iM
et,Tyr');
my %aa_let2int;
my %aa_abb2int;
my $i = 0;
foreach (@aa_letters) {
	$aa_let2int{$_} = $i;
	$i++;
}
$i = 0;
foreach (@aa_abbrevs) {
	$aa_abb2int{$_} = $i;
	$i++;
}
```

```
$i = 0;

my $aa_disjunction = join('|',@aa_abbrevs);
print "AA abbreviation disjunction: $aa_disjunction\n";
my @presentaas;

my @nts = ('A','C','G','T','-');

# (# samples)x25 matrices (cols: seq ID, AA pred, 23 AA scores)
# Note: this will disregard seqs without AA pred
my @target_scores;
my @query_scores; # added bc of errors

# Definitions
my $profmatfile;
my $alnlen;
my %query_seqs;
my %target_seqs;
my $flname;
my $ilname;
if ($opt_e) {
	($flname, $ilname) = split(':', $opt_e);
}

####################
# Echo parameters #
####################
my ($tfile, $qfile, $lfile) = @ARGV;

print "Parameters:\nOutput files prefixed with $opt_t\n";
foreach my $opt (keys(%Opts)) {
	print "$opt\t".$Opts{$opt}."\n";
}
print "\nInput:\nTarget\t$tfile\nQuery\t$qfile\n\n";


############################
# Generate new TFAM scores #
############################
if (1) { # TODO: allow options to load scores from tropDB or file
	unless ($lfile) {$lfile = $opt_t."legend";}
	open TIN, $tfile or die "ERROR: Cannot open TARGET fasta named $tfile for reading!
\n";
	open QIN, $qfile or die "ERROR: Cannot open QUERY fasta named $qfile for reading!
\n";
	open MIXOUT, ">$opt_t"."combined.fa";

if ($opt_r) {
	print STDERR "Will refactor headers\n";
	# Refactoring sequence headers so that TFAM doesn't obscure names; max 10^11 tRNAs
per file!
	open LOUT, ">$lfile" or die "ERROR: Cannot open naming legend file named $lfile
for writing!\n";

	### TODO: Make this instead apply "tSE-type" headers that TFAM can understand
	my $tcount = 0;
	while (<TIN>) {
		if (/^>.+/) {
			my ($cl, $ac) = $_ =~ /T:($aa_disjunction) A:(\S{3})/; # Capture
functional annotation
			if($cl){$cl = $aa_letters[$aa_abb2int{$cl}];}
			unless ($cl) {
				print "WARNING: Target tRNA $_ has an unsupported functional
annotation. Skipping!\n";
				next;
			}
```

```
                        print MIXOUT ">TGT_$ac-$cl-".sprintf('%06s',$tcount)."\n";
                        print LOUT ">TGT_$ac-$cl-".sprintf('%06s',$tcount)."\t$_";
                        $_ = <TIN>;
                        print MIXOUT;
                        $tcount++
                }
        }

        my $qcount = 0;
        while (<QIN>) {
                if (/^>.+/) {
                        my ($cl, $ac) = $_ =~ /T:($aa_disjunction) A:(\S{3})/; # Capture
functional annotation
                        if($cl){$cl = $aa_letters[$aa_abb2int{$cl}];} else {$cl = 'Undet';}
                        $ac = 'NNN' unless ($ac);
                        print MIXOUT ">QRY_$ac-$cl-".sprintf('%06s',$qcount)."\n";
                        print LOUT ">QRY_$ac-$cl-".sprintf('%06s',$qcount)."\t$_";
                        $_ = <QIN>;
                        print MIXOUT;
                        $qcount++;
                }
        }
        close LOUT;
} else {
        # If NOT refactoring headers, combine all tRNAs into one file
        while (<TIN>) {
                print MIXOUT;
        }
        while (<QIN>) {
                print MIXOUT;
        }
}
        close TIN;
        close QIN;
        close MIXOUT;

        # Run tfam with all the tRNAs in one big blob
        print "TFAM started at ".timestamp."\n";
        system "tfam ".($opt_m ? "-$opt_m" : '')." -s -t $opt_t"."combined.fa $opt_t >
$opt_t"."tfamlog 2>&1";
        system "rm $opt_t.?.fas";
        print "TFAM finished at ".timestamp."\n";

        print "Parsing TFAM scores ... ";
        open RESIN, "$opt_t" or die "ERROR: Cannot open TFAM result file \"$opt_t\"!\n";
        my @lineparts;
        while (<RESIN>) {
                if (/TGT_/) {
                        @lineparts = split(/\s+/, $_);
                        shift @lineparts if ($lineparts[0] eq ''); # TFAM right-adjusts seq
names by adding leading spaces, resulting in a blank first element after split()
                        if ($lineparts[2] eq 'undet') {
                                splice(@lineparts, 2, 2); # If input class is unknown, save
TFAM class prediction (remove input & match columns)
                        } else {
                                splice(@lineparts, 1, 1); # Else, save pre-TFAM class (remove
TFAM class column)
                                splice(@lineparts, 3, 1); # (remove match column)
                        }
                        push(@target_scores,[@lineparts]);
                        #print "Found target score!\n"
                }
                if (/QRY_/) {
                        @lineparts = split(/\s+/, $_);
                        shift @lineparts if ($lineparts[0] eq '');
                        splice(@lineparts, 2, 2); # to remove input & match columns
```

```
                        push(@query_scores,[@lineparts]);
                }
        }
        close RESIN;
        # Peek at first tRNA to get alignment length, then reopen file
        open ALNIN, "$opt_t.aln.fas" or die "ERROR: Cannot open TFAM alignment file
\"$opt_t.aln.fas\" for reading!\n";
        my $line = <ALNIN>;
        $line = <ALNIN>;
        chomp $line;
        $alnlen = scalar(split('',$line));
        close ALNIN;
        print "done!\n";
        print "$0 read alignment length $alnlen from alignment file \"$opt_t.aln.fas\". If
this does not match TFAM output, redo analysis!\n";

        # Set up profile matrices
        my %profile_matrices;
        foreach my $aa (@aa_letters) {
                for (my $i = 0; $i < $alnlen; $i++) {
                        $profile_matrices{$aa}{'A'}[$i] = 0;
                        $profile_matrices{$aa}{'C'}[$i] = 0;
                        $profile_matrices{$aa}{'G'}[$i] = 0;
                        $profile_matrices{$aa}{'T'}[$i] = 0;
                        $profile_matrices{$aa}{'-'}[$i] = 0;
                }
        }

        print "Parsing TFAM alignment ... ";
        open ALNIN, "$opt_t.aln.fas";
        while (<ALNIN>) {
                if (/>TGT_/) { # Make sure that only target sequences are used to build
logos
                        my $tid = $_;
                        chomp $tid;
                        my $class = '';
                        my $tfclass = '';
                        ($class,$tfclass) = /-(\S*)-.+\sTFAM:([A-Z?#])/;#/>.+TFAM:([A-Z])/;

                        # In case of nonstandard headers, try again to catch TFAM
classification
                        if ($tfclass eq '') {
                                $tfclass = /TFAM:([A-Z?#])/;
                        }
                        if ($tfclass eq 'X' || $tfclass eq 'J' || $class eq '' || $class eq
'undet' || $class eq '???') {
                                $class = $tfclass; # If original prediction was undetermined,
take tfam classification
                        }

                        my $seq;
                        unless ($class eq '') {
                                $seq = <ALNIN>;
                                chomp $seq;
                                my @seq = split('',$seq);
                                $target_seqs{$tid} = [$class, [@seq]];
                                my $pos = 0;
                                foreach my $base (@seq) {
                                        $base =~ tr/a-z/A-Z/;
                                        my $unbase = "ACGT-";
                                        $unbase =~ tr/$base//; # Remove current base from base
list
                                        my @others = split('',$unbase); # Create base-complement
list
                                        $profile_matrices{$class}{$base}[$pos]++;
                                        $pos++;
```

```
                        }
                }

                # If creating basepair logos later, separate seqs by class
                if ($opt_p) {
                        open COUT, ">>$opt_t"."_$class.fas";
                        print COUT "$tid\n$seq\n";
                        close COUT;
                }
        }
        if (/>QRY_/) {
                my $qid = $_;
                chomp $qid;
                my $class = '';
                my $tfclass = '';
                ($class,$tfclass) = /-(\S*)-.+\sTFAM:([A-Z?#])/;#/>.+TFAM:([A-Z])/;

                # In case of nonstandard headers, try again to catch TFAM
classification
                if ($tfclass eq '') {
                        $tfclass = /TFAM:([A-Z?#])/;
                }

                if ($class eq '?' || $tfclass eq 'X' || $tfclass eq 'J' || $class eq
'' || $class eq 'undet' || $class eq '???') {
                        $class = $tfclass; # If original prediction was undetermined,
take tfam classification
                }

                my $seq = <ALNIN>;
                chomp $seq;
                $query_seqs{$qid} = [$class, [split('',$seq)]];
        }
}
close ALNIN;
print "done!\n";

# If making basepair logos later, convert alignments to clustal
if ($opt_p) {
        foreach my $aa (@aa_letters) {
                my $fstub = "$opt_t\_$aa";
                my $in = Bio::AlignIO->new( -file => "$fstub.fas", -format =>
'fasta');
                my $out = Bio::AlignIO->new( -file => ">$fstub.aln", -format =>
'clustalw');
                while (my $aln = $in->next_aln) {
                        $out->write_aln($aln);
                }
                system "rm $fstub.fas";
        }
}

unless ($opt_e) {
open PMATOUT, ">$opt_t"."profile_matrix" or die "ERROR: Cannot open profile matrix
file \"$opt_t"."profile_matrix\" for writing!\n";
my $aacount = 0;
my $nullcount = 0;
my $nullstr = '';
foreach my $class (@aa_letters) {
        $aacount++;

        # Check for zero matrix
        my $temp = 0;
        foreach (@nts) {
                $temp += $profile_matrices{$class}{$_}[0];
        }
```

```
            if ($temp==0) {
                  $nullcount++;
                  $nullstr .= $class;
            }

            for (my $i = 0; $i < $alnlen; $i++) {
                  print PMATOUT $profile_matrices{$class}{'A'}[$i]." ";
                  print PMATOUT $profile_matrices{$class}{'C'}[$i]." ";
                  print PMATOUT $profile_matrices{$class}{'G'}[$i]." ";
                  print PMATOUT $profile_matrices{$class}{'T'}[$i]." ";
                  print PMATOUT $profile_matrices{$class}{'-'}[$i]."\n";
            }
            print PMATOUT "\n"; # Class-separating blank line
      }
      close PMATOUT;


      $profmatfile = "$opt_t"."profile_matrix";
      print "Profile matrix file written to \"$profmatfile\" with $aacount matrices\n";
      if ($nullcount) {
            die "ERROR! $nullcount profile matrices ($nullstr) are zero!\n";
      }
      }
}

####################################
# Generate B- & T-heaps for each AA #
####################################
if (0) {
my %non_heaps; # Stores, per AA, scores for all Bkg & Tgt tRNAs NOT identified as that AA
my %tgt_heaps; # Stores, per AA, scores for Target RNAs identified as that AA

foreach my $amino (@aa_abbrevs) {
      $non_heaps{$amino} = [[],[]]; # Two arrays: [values, samplenames]
      $tgt_heaps{$amino} = [[],[]]; # Two arrays: [values, samplenames]
}

foreach my $tgtrecord (@target_scores) {
      my $rec_rna = @{$tgtrecord}[0]; # Store record's RNA identifier
      my $rec_aa = @{$tgtrecord}[1]; # Store record's AA identity
      for (my $i = 2; $i < 25; $i++) { # BEWARE! HARD CODED!
            if (($i-2) == $aa_abb2int{$rec_aa}) { # trigger for the record's aa
                  push (@{$tgt_heaps{$rec_aa}[0]}, @{$tgtrecord}[$i]); # push record's
aa's score onto tgt-heap
                  push (@{$tgt_heaps{$rec_aa}[1]}, @{$tgtrecord}[0]); # push record's
rna's identifier onto tgt-heap
            } else {
                  push (@{$non_heaps{$rec_aa}[0]}, @{$tgtrecord}[$i]); # push record's
aa's score onto non-heap
                  push (@{$non_heaps{$rec_aa}[1]}, @{$tgtrecord}[0]); # push record's
rna's identifier onto non-heap
            }
      }
}

# Each hash value is an array of mean, std dev, range
my %target_stats;
my %other_stats;

print "\nAA\tTmean\tTstdev\tTrange\tOmean\tOstdev\tOrange\n";
print   "--\t-----\t------\t------\t-----\t------\t------\n";

foreach my $aa (@aa_abbrevs) {
      my $stat = Statistics::Descriptive::Full->new();
      $stat->add_data(@{$tgt_heaps{$aa}[0]});
```

```
	$target_stats{$aa}=[$stat->mean(),$stat->standard_deviation(),$stat-
>sample_range()];
	$stat = Statistics::Descriptive::Full->new();
	$stat->add_data(@{$non_heaps{$aa}[0]});
	$other_stats{$aa}=[$stat->mean(),$stat->standard_deviation(),$stat-
>sample_range()];
	print "$aa\t".join("\t",map(sprintf('%.6g',
$_),@{$target_stats{$aa}})."\t".join("\t",map(sprintf('%.6g',
$_),@{$other_stats{$aa}}))."\n";

}

}

###############
# Run Logofun #
###############
unless ($opt_e) {
$flname = $opt_t."funlogo";
$ilname = $opt_t."invlogo";
# Make function logos
print "Function logo generation started at ".timestamp."\n";
system "python /sw/logofun-1.0/logofun --function --states 'ACGT-' --classes $aa_string
--exact 1 --output $flname --title -d d $profmatfile";
print "Function logo generation finished at ".timestamp."\n";

# Make inverse function logos
print "Inverse function logo generation started at ".timestamp."\n";
system "python /sw/logofun-1.0/logofun --function --inverse --states 'ACGT-' --classes
$aa_string --exact 1 --output $ilname --title -d d $profmatfile";
print "Inverse function logo generation finished at ".timestamp."\n";
} else {
	print "Parsing function logo files $flname\_{ACGT-} and inverse function logo
files $ilname\_{ACGT-}.\n";
}

#################
# Run bplogofun #
#################
if ($opt_p) {
	open AIN, $opt_t."combined.fa.coveaf";
	open CSOUT, ">$opt_t"."cs";

	my $in = Bio::AlignIO->new( -file => "$opt_t.aln.fas",
						-format => 'fasta');
	my $out = Bio::AlignIO->new( -file => ">$opt_t.clustalw",
						-format => 'clustalw');
	while( my $aln = $in->next_aln ) {
		$out->write_aln($aln);
	} # Translate multiple aln to clustalw

	while (<AIN>) {
		if (/#=CS/) {
			print CSOUT;
		}
	}
	close AIN;
	close CSOUT;

	print "Basepair function logo generation started at ".timestamp."\n";
	system "bplogofun3 -c $opt_t"."cs $opt_t 2> /dev/null";# > $opt_t"."bplog 2>&1";
	print "Basepair function logo generation finished at ".timestamp."\n";

}

###############
```

```
# Parse logos #
###############
my %pim; # for Profile Information Matrix
my %pim_inv; # ditto for inverse logos
my %fldata;
my %ildata;
for (my $i = 0; $i < $alnlen; $i++) {
	foreach my $nt (@nts) {
		$pim{$i}{$nt} = 0;
		$pim_inv{$i}{$nt} = 0;
		foreach my $aa (@aa_abbrevs) {
			$fldata{$i}{$nt}{$aa} = 0;
			$ildata{$i}{$nt}{$aa} = 0;
		}
	}
}

# Excluded region limits, if option -x #:# is given
my %pim_exc;
if ($opt_x) {
	my @dumparr = @xlimits;
	my $limitstring;
	if (($dumparr[0] eq 'save') && ($dumparr[1] eq 'info')) {
		# If the first two elements of the exclusion argument are 'save' and 'info',
store excluded info
		shift @dumparr;
		shift @dumparr;
		shift @xlimits;
		shift @xlimits;
		$savexc = 1;
	}
	while (@dumparr) {
		$limitstring .= '['.shift(@dumparr).','.shift(@dumparr).'], ';
	}
	print "Excluding alignment region(s) $limitstring"."from scoring.\n";
}

foreach my $nt (@nts) {
	open FLIN, $flname."_$nt.eps" or die "ERROR: Cannot open function logo file
\"$flname\_$nt.eps\" for reading!\n";
	open ILIN, $ilname."_$nt.eps" or die "ERROR: Cannot open inverse function logo
file \"$ilname\_$nt.eps\" for reading!\n";

	my $coord;
	my $include = 1;
	while (<FLIN>) {
		if ($_=~/^numbering \{\((\d{1,3})\) makenumber\} if/){
			# Encountered new alignment coordinate!
			$coord=$1; # Shift alignment-coordinate cursor
			if ($opt_x && ($coord==$xlimits[0])) {
				# Toggle inclusion flag if crossing an exclusion region border
				$include = !$include;
			}
		}
		if ($_=~/^ (\d+\.\d+) \(([A-Z])\) numchar/){
			# Encountered a logo character!
			my $data=$1; my $aa=$2;
			if ($include) {
				$pim{$coord}{$nt}+=$data;
				# Sum all information for each position
				$fldata{$coord}{$nt}{$aa} = $data; # Save function data for
INCLUDED regions
			} elsif ($savexc) {
				$pim_exc{$coord}{$nt}+=$data;
			}
		}
```

```
	}
	while (<ILIN>) {
		if ($_=~/^numbering \{\((\d{1,3})\) makenumber\} if/){
			# Encountered new alignment coordinate!
			$coord=$1; # Shift alignment-coordinate cursor
			if ($opt_x && ($coord==$xlimits[0])) {
				# Toggle inclusion flag if crossing an exclusion region border
				$include = !$include;
			}

		}
		if ($_=~/^ (\d+\.\d+) \(([A-Z])\) numchar/){
			# Encountered a logo character!
			my $data=$1; my $aa=$2;
			if ($include) {
				$pim_inv{$coord}{$nt}+=$data;
				$ildata{$coord}{$nt}{$aa} = $data; # Save inverse data for
INCLUDED regions
			} # Sum all information for each position
			elsif ($savexc) {
				$pim_exc{$coord}{$nt}+=$data;
			}
		}
	}
	close FLIN;
	close ILIN;
}

############################
# Score vs target's classes #
############################
if ($opt_c) {
	open CLOUT, ">$opt_t\_clspec_scores" or die "ERROR: Cannot open score file
\"$opt_t\_clspec_scores\" for writing!\n";
	print CLOUT "Sequence\tClass";
	# Print SVMlight-readable output
	open SVMOUT, ">$opt_t"."clspec_svmlight";
	print SVMOUT"#Class";

	foreach (sort @aa_letters) {
		print CLOUT "\tTGT_$_\tTGT_$_\_inverse";
		print SVMOUT " TGT_$_ TGT_$_\_inverse";
	}
	print CLOUT "\n";
	print SVMOUT " Seqname\n";
	open COMPOUT, ">$opt_t\_compound_scores" or die "ERROR: Cannot open score file
\"$opt_t\_compound_scores\" for writing!\n";
	print COMPOUT "Sequence\tClass\tCompound_score\n";

	# Calculate total information per class
	my %total_info;
	my $suminv = 0;
	print "Total information (obverse, inverse) per class:\n";
	foreach my $class (sort @aa_letters) {
		no warnings; # Avoid uninitialized element warnings
		for (my $i = 0; $i < $alnlen; $i++) {
			$total_info{$class}{'function'} += $fldata{$i}{'A'}{$class};
			$total_info{$class}{'function'} += $fldata{$i}{'C'}{$class};
			$total_info{$class}{'function'} += $fldata{$i}{'G'}{$class};
			$total_info{$class}{'function'} += $fldata{$i}{'T'}{$class};

			$total_info{$class}{'inverse'} += $ildata{$i}{'A'}{$class};
			$total_info{$class}{'inverse'} += $ildata{$i}{'C'}{$class};
			$total_info{$class}{'inverse'} += $ildata{$i}{'G'}{$class};
			$total_info{$class}{'inverse'} += $ildata{$i}{'T'}{$class};
		}
```

```
            print "$class:\t".$total_info{$class}{'function'}."\t".$total_info{$class}
{'inverse'}."\n";
            $suminv += $total_info{$class}{'inverse'};
      }
      my @qkeys = sort(keys(%query_seqs));
      my @tkeys = sort(keys(%target_seqs));

      # In order to keep scores positive for extreme differences between function logo &
inverse information, add a multiple of theoretical max information
      my $aacount = scalar(@aa_letters);
      my $offset = $alnlen * 4 * log2($aacount);
      $offset = $suminv;
      print sprintf("Safety offset: %.2f\n",$offset);

      my ($maxscore, $minscore) = (-10e6, 10e6);

      # Score QUERY seqs
      foreach my $qkey (@qkeys) {

            my @seq = @{$query_seqs{$qkey}[1]};
            my $qcl = $query_seqs{$qkey}[0];

            print CLOUT "$qkey\t$qcl";
            print SVMOUT "1";
            my $featcount = 0;
            my $prod_over_classes = 1;

            foreach my $class (sort @aa_letters) {
                  my $temp_flscore = 0;
                  my $temp_flsum = 0;
                  my $temp_ilscore = 0;
                  my $temp_ilsum = 0;
                  my $temp_totscore = 0;

                  # Make a list of the complement of the current class
                  my $unclass = join('',@aa_letters);
                  $unclass =~ tr/$class//;
                  my @unclasses = split('',$unclass);

                  for (my $i = 0; $i < $alnlen; $i++) {
                        no warnings;
                        # Sum of information heights for the current class
                        $temp_flsum += $fldata{$i}{$seq[$i]}{$class};
                        $temp_flscore += $fldata{$i}{$seq[$i]}{$class}/
$total_info{$class}{'function'};

                        $temp_ilsum += $ildata{$i}{$seq[$i]}{$class};
                        # Sum of inverse information heights for all classes BUT the
current class
                        foreach $unclass (@unclasses) {
                              $temp_ilscore += ($ildata{$i}{$seq[$i]}{$unclass})/
$total_info{$unclass}{'inverse'};
                        }
                  }
                  print CLOUT "\t$temp_flscore\t$temp_ilscore";

                  # Print SVMlight-readable output
                  $featcount++;
                  print SVMOUT " $featcount:$temp_flsum";
                  $featcount++;
                  print SVMOUT " $featcount:$temp_ilsum";

                  $temp_totscore = $offset + $temp_flscore - $temp_ilscore;#
$temp_flscore/$total_info{$class}{'function'} - $temp_ilscore/$total_info{$class}
{'inverse'};
                  $prod_over_classes *= $temp_totscore;
```

```
                $maxscore = $temp_totscore if ($temp_totscore > $maxscore);
                $minscore = $temp_totscore if ($temp_totscore < $minscore);
            }
            my $comp_score = $prod_over_classes ** (1/$aacount); # N-th root of the
product, where N is the number of functional classes

            print COMPOUT "$qkey\t$qcl\t$comp_score\n";
            print CLOUT "\n";
            print SVMOUT " #$qkey\n";
      }

      # score TARGET seqs
      foreach my $tkey (@tkeys) {

            my @seq = @{$target_seqs{$tkey}[1]};
            my $tcl = $target_seqs{$tkey}[0];

            print CLOUT "$tkey\t$tcl";

            my $prod_over_classes = 1;

            print SVMOUT "-1";
            my $featcount = 0;

            foreach my $class (sort @aa_letters) {
                  my $temp_flscore = 0;
                  my $temp_ilscore = 0;
                  my $temp_totscore = 0;
                  my $temp_flsum = 0;
                  my $temp_ilsum = 0;

                  # Make a list of the complement of the current class
                  my $unclass = join('',@aa_letters);
                  $unclass =~ tr/$class//;
                  my @unclasses = split('',$unclass);

                  for (my $i = 0; $i < $alnlen; $i++) {
                        no warnings;
                        # Sum of information heights for the current class
                        $temp_flsum += $fldata{$i}{$seq[$i]}{$class};
                        $temp_flscore += $fldata{$i}{$seq[$i]}{$class}/
$total_info{$class}{'function'};

                        $temp_ilsum += $ildata{$i}{$seq[$i]}{$class};
                        # Sum of inverse information heights for all classes BUT the
current class
                        foreach $unclass (@unclasses) {
                              #$temp_ilscore += $ildata{$i}{$seq[$i]}{$unclass};
                              $temp_ilscore += ($ildata{$i}{$seq[$i]}{$unclass})/
$total_info{$unclass}{'inverse'};
                        }
                  }



                  print CLOUT "\t$temp_flscore\t$temp_ilscore";

                  # Print SVMlight-readable output
                  $featcount++;
                  print SVMOUT " $featcount:$temp_flsum";
                  $featcount++;
                  print SVMOUT " $featcount:$temp_ilsum";
```

```
                    $temp_totscore = $offset + $temp_flscore - $temp_ilscore;#
$temp_flscore/$total_info{$class}{'function'} - $temp_ilscore/$total_info{$class}
{'inverse'};
                    $prod_over_classes *= $temp_totscore;

                    $maxscore = $temp_totscore if ($temp_totscore > $maxscore);
                    $minscore = $temp_totscore if ($temp_totscore < $minscore);
             }

             my $comp_score = $prod_over_classes ** (1/$aacount); # N-th root of the
product, where N is the number of functional classes

             print COMPOUT "$tkey\t$tcl\t$comp_score\n";
             print CLOUT "\n";
             print SVMOUT " #$tkey\n";
      }

      print "Maxscore: $maxscore\nMinscore: $minscore\n";
      close CLOUT;
      close COMPOUT;
      close SVMOUT;
}



#########################################################
# Record total information content in function logos #
#########################################################
my %qscores;
my @tgt_ftotals; # X values (total function logo information) for plots
my @tgt_itotals; # Y values (total inverse function logo information) for plots
my @qry_ftotals; # X values (total function logo information) for plots
my @qry_itotals; # Y values (total inverse function logo information) for plots

open SCOUT, ">$opt_t\_scores" or die "ERROR: Cannot open score file \"$opt_t\_scores\"
for writing!\n";
print SCOUT "Sequence\tPred_class\tScore\n";
my @qkeys = sort(keys(%query_seqs));
foreach my $qkey (@qkeys) {
      my $class = $query_seqs{$qkey}[0];
      my @seq = @{$query_seqs{$qkey}[1]};
      my $score = 0;
      ############################## SCORING ALGORITHM
####################################
      for(my $i = 0; $i < $alnlen; $i++){
            my $nt = $seq[$i];
            $nt =~ tr/a-z/A-Z/;
            unless (($nt eq '-') && !$opt_g) {$score += ($pim{$i}{$nt} - $pim_inv{$i}
{$nt});}

            if ($opt_s) {
                  unless (($nt eq '-') && !$opt_g) {
                        no warnings;
                        $qscores{$qkey}{'function'} += $pim{$i}{$nt};
                        $qscores{$qkey}{'inverse'} += $pim_inv{$i}{$nt};
                        if ($savexc) {
                              $qscores{$qkey}{'excluded'} += $pim_exc{$i}{$nt};
                        }

                        if ($opt_l) {
                              my $fmax = 0;
                              my $imax = 0;
                              foreach (@aa_letters) {
                                    # Select the largest information value at the
position in logo
```

```
                                        if ($fldata{$i}{$nt}{$_} > $fmax) {$fmax =
$fldata{$i}{$nt}{$_};}
                                        # To take sum info in inverse
                                        $imax += $ildata{$i}{$nt}{$_};
                                }
                                $qscores{$qkey}{'function_max'} += $fmax;
                                $qscores{$qkey}{'inverse_max'} += $imax;
                        }
                }
        }
}

#################################################################################
        print SCOUT "$qkey\t$class\t$score\n";
        if (scalar(@seq) != $alnlen) {print "WARNING: Query has alignment length
".scalar(@seq)."!\n";}
}
close SCOUT;

# If saving scores, score target seqs
if ($opt_s) {
        my %tscores;
        my @tkeys = sort(keys(%target_seqs));
        foreach my $tkey (@tkeys) {
                my $class = $target_seqs{$tkey}[0];
                my @seq = @{$target_seqs{$tkey}[1]};
                my $score = 0;
                for(my $i = 0; $i < $alnlen; $i++){
                        my $nt = $seq[$i];
                        $nt =~ tr/a-z/A-Z/;
                        unless (($nt eq '-') && !$opt_g) {
                                no warnings;
                                #Score target seqs like query seqs
                                $tscores{$tkey}{'function'} += $pim{$i}{$nt};
                                $tscores{$tkey}{'inverse'} += $pim_inv{$i}{$nt};
                                if ($savexc) {$tscores{$tkey}{'excluded'} += $pim_exc{$i}
{$nt};}

                                #Score target seqs only by their predicted class:
                                $tscores{$tkey}{'function_byclass'} += $fldata{$i}{$nt}
{$class};
                                foreach my $unclass (@aa_letters) {
                                        # Antideterminant information content in a target tRNA
might be better represented by the sum of
                                        # inverse info for all OTHER classes
                                        unless ($unclass eq $class) {$tscores{$tkey}
{'inverse_byclass'} += $ildata{$i}{$nt}{$unclass};}
                                }
                        }
                }
        }

        open ALLSCOUT, ">$opt_s";
        print ALLSCOUT "Sequence\tTFAM_class\tFunction_score\tInverse_score\tFunction_byAA
\tInverse_byAA".($savexc ? "\tExcluded": '')."\n";
        foreach (keys(%qscores)) {
                if ($opt_l) {
                        print ALLSCOUT $_."\t".$query_seqs{$_}[0]."\t".$qscores{$_}
{'function'}."\t".$qscores{$_}{'inverse'}."\t".$qscores{$_}{'function_max'}."\t".
$qscores{$_}{'inverse_max'}.($savexc ? "\t".$qscores{$_}{'excluded'} : '')."\n";
                        push @qry_ftotals, $qscores{$_}{'function_max'};
                        push @qry_itotals, $qscores{$_}{'inverse_max'};
                } else {
                        print ALLSCOUT $_."\t".$query_seqs{$_}[0]."\t".$qscores{$_}
{'function'}."\t".$qscores{$_}{'inverse'}."\t0\t0".($savexc ? "\t".$qscores{$_}
{'excluded'} : '')."\n";
```

```
                    push @qry_ftotals, $qscores{$_}{'function'};
                    push @qry_itotals, $qscores{$_}{'inverse'};
                }
        }
        foreach (keys(%tscores)) {
                print ALLSCOUT $_."\t".$target_seqs{$_}[0]."\t".$tscores{$_}
{'function'}."\t".$tscores{$_}{'inverse'}."\t".$tscores{$_}{'function_byclass'}."\t".
$tscores{$_}{'inverse_byclass'}.($savexc ? "\t".$tscores{$_}{'excluded'} : '')."\n";
                    push @tgt_ftotals, $tscores{$_}{'function'};
                    push @tgt_itotals, $tscores{$_}{'inverse'};
        }
        close ALLSCOUT;
}
#####################################################################
# Gnuplot input tRNAs' total logo info vs total inverse logo info #
#####################################################################

# Global options for gnuplots
my %gpl_opts =      (
                                'title' => 'Total Inverse Function Logo Information\nvs.
Total Function Logo Information',
                                'output type' => 'png',
                                'output file' => $opt_t."infoplot.png",
                                'x-axis label' => 'Total Function Logo Information',
                                'y-axis label' => 'Total Inverse Function Logo
Information',
                        );

my %tgt_opts =      (
                                #'color' => '#000000',
                                'style' => 'points',
                                'type' => 'columns',
                                'title' => 'Target tRNAs',
                        );

my %qry_opts =      (
                                #'color' => '#FF0000',
                                'style' => 'points',
                                'type' => 'columns',
                                'title' => 'Query tRNAs',
                        );

print "Gnuplot generation started at ".timestamp.".\n";
gnuplot(\%gpl_opts, [\%tgt_opts, \@tgt_ftotals, \@tgt_itotals],
                                [\%qry_opts, \@qry_ftotals, \@qry_itotals]);
print "Gnuplot generation finished at ".timestamp.".\n";

print "Done.\n";


###############
# Subroutines #################################################################
###############

# ZEROES
sub zeroes {
        my $length = shift;
        my @result;
        for (my $i = 0; $i < $length; $i++) {
                push (@result, 0);
        }
        return \@result;
}

# TIMESTAMP
# Return a nicely readable string for the current time
```

```
sub timestamp {
       my @abbr = qw( Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec );
       my @days = qw( Sun Mon Tue Wed Thu Fri Sat);
       # Get the time
       my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = @{localtime(time)};
       $year += 1900;
       return sprintf "%02d:%02d:%02d $days[$wday] $abbr[$mon] $mday $year",$hour,$min,
$sec;
}

# LOG2
# Return 2-logarithm of input
sub log2 {
       my $num = shift;
       return log($num)/log(2);
}
```

Appendix 4: tRNAscan-SE Output Processing Script (tse2fa.pl)

```
#!/usr/local/bin/perl -w

use strict;
use Getopt::Std;

die ("Usage: tse2fa.pl <tSE structure file> <FASTA output filename>\n") unless $ARGV[0];

#### Handle options
my %Opts;
&getopts('hst:',\%Opts);
my $opt_h = $Opts{'h'};          # Print help and die
my $opt_t = $Opts{'t'};          # Generate TFAM-compatible "tSE" header
my $opt_s = $Opts{'s'};          # Generate TFAM-compatible "simple" header


my $helpline = "Usage: tse2fa.pl <tSE structure file> <FASTA output filename>\n";
$helpline .= "Designed to be used on tRNAscan-SE secondary structure output!\n";
$helpline .= "options:\n";
$helpline .= "\t-h\t\tprint this help and exit\n";
$helpline .= "\t-s\t\tgenerate headers in TFAM-compatible \"simple\" format\n";
$helpline .= "\t-t <string>\tgenerate tRNA headers in TFAM-compatible tSE format, with
given string as taxonomic ID\n";
die $helpline if ($opt_h);

#### Get arguments
my @args = @ARGV;
my $tsefile = $args[0];
my $fafile = ($args[1] ? $args[1] : "$tsefile.fa");
my $count = 0;
my ($header, $seq);

#### Check I/O
open IN, $tsefile or die("Could not open $tsefile for reading!\n");
open OUT, ">$fafile" or die("Could not open $fafile for writing!\n");


#### Store AA names & codes and generate lookup tables; might need it when dealing with
TFAM output
my $aa_string = 'ACDEFGHIjKLMNPQRSTVWXYZ$?';
my @aa_letters = split(//,$aa_string);
my @aa_abbrevs =
split(/,/,'Ala,Cys,Asp,Glu,Phe,Gly,His,Ile,kIle,Lys,Leu,Met,Asn,Pro,Gln,Arg,Ser,Thr,Val,T
rp,iMet,Tyr,SeC,Pseudo,Undet');
my %aa_let2int;
my %aa_abb2int;
my $i = 0;
foreach (@aa_letters) {
       $aa_let2int{$_} = $i;
       $i++;
}
$i = 0;
foreach (@aa_abbrevs) {
       $aa_abb2int{$_} = $i;
       $i++;
}

#### Loop through input
# Assumes input
while (<IN>) {
   if ($_ =~ /\.trna\d+.+Length:/) { # Match tRNA header

      $header = $_;           # Save header
      chomp $header;
```

```
     #### Format header to Yyyy[class character]xxx######### with Yyyy = opt_t, xxx =
anticodon, ##### = location
     if ($opt_t) {
        $header =~ /\((\d+)-(\d+)\)/;          # Catch location info
        my $location = ($1 < $2 ? $1 : "-$2");    # Save the gene location (with a
negative if it happens to be reversed)
        <IN> =~ /Type:\ (\S+)\s+Anticodon:\s(\S+)/; # Grab type & anticodon
        my $type;
        if (exists $aa_abb2int{$1}) {
           $type = $aa_letters[$aa_abb2int{$1}];  # If the detected type string matches a
registered class, save the appropriate class identifier
        } else {
           $type = "?";                # else, mark as undetermined
        }
        $header = $opt_t.$type.$2.$location;   # TODO: possibly add sequential
identifier and/or genome identifier after location? Can TFAM take the header with those
modifications?

     }
     #### Or, format header to "simple" TFAM-readable format: XxxxxYYY-Zzz-#####
     elsif ($opt_s) {
        my $num = $header =~ /\.trna(\d+)/; # Capture sequential id
        $header =~ s/[-\|:\.,;\s+]/_/g; # Dashes are delimiters in simple format -
replace! Also convert other delimiters
        $header =~ s/[^\w]//g; # Remove characters that are not alphanumeric or
underscores
        <IN> =~ /Type:\ (\S+)\s+Anticodon:\s(\S+)/; # Grab type & anticodon
        $header .= "_$2-$1-$num"; # Format: ConcentratedOldHeader<Anticodon>-<Class>-
<SequentialID>
     }
     #### Or, contract existing info into one header line
     else {
        $header =~ s/\s+/\ /g;       # Contract spaces
        $header =~ s/Length/L/g;        # Contract Length->L
        <IN> =~ /Type:\ (\S+)\s+Anticodon:\s(\S+)/; # Grab type & anticodon
        $header = $header." T:$1 A:$2"; # Attach to header
     }

     $seq = <IN>;                 # Skip ruler line
     if ($seq =~ /pseudogene|intron/) {
        $seq = <IN>;
     }
     $seq = <IN>;            # Get seq line
     chomp $seq;
     $seq =~ s/Seq:\ //;          # Remove leading "Seq: "
     print OUT ">$header\n$seq\n";  # Write to output
     $count ++;
   } else {
     next;                # Skip non-header lines
   }
}

close IN;
close OUT;

print STDOUT "$count tRNAs parsed.\n";
```