ISAAC FOR INTEGRONS AND ATTC

ISAAC: AN IMPROVED STRUCTURAL ANNOTATION OF ATTC AND AN INITIAL APPLICATION THEREOF

BY

JUDITH COWPER SZAMOSI, B.SC.

A THESIS

SUBMITTED TO THE DEPARTMENT OF BIOLOGY

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

© Copyright by Judith Cowper Szamosi, B.Sc., September 2012

Licensed under a Creative Commons Attribution, Non-Commercial, Share-Alike license

Master of Science (2012) McMaster University

(Biology) Hamilton, Ontario, Canada

TITLE: ISAAC: AN IMPROVED STRUCTURAL ANNOTATION OF ATTC AND AN INITIAL APPLICATION THEREOF

AUTHOR: Judith Cowper Szamosi, B.Sc.

B.Sc., (Biology)

University of Toronto, Toronto, Canada

SUPERVISOR: Dr. Jonathan Dushoff

NUMBER OF PAGES: x, 107

Abstract

We introduce new software (ISAAC: **I**mproved **S**tructural **A**nnotation of **a**tt**C**) to annotate cassette arrays in bacterial integrons by finding attI and attC sites, and to provide detailed annotation of the attC sites for analysis. We demonstrate an initial application of ISAAC by annotating the cassette complements of all the integrons we identified in the RefSeq bacterial genome database, and providing an analysis of the patterns of nucleotide frequencies at the structurally important positions in the attCs we've found.

Acknowledgments

I would like to thank my supervisor, Jonathan Dushoff, and my committee members, Paul Higgs and Brian Golding, for their guidance and support. Thanks also to Ben Evans, David Earn, Ben Bolker, and Marie Elliot for their help and advice. Thanks to all the members of the Dushoff, Golding, and Evans labs 2010 - 2012, and especially Lee Worden, Hermina Ghenu, Melanie Lou, Terri Porter, and Yifei Huang for their invaluable assistance, emotional support, and technical advice. Thanks to my partner, David Leaman, and my father, Michael Szamosi, for their assistance with programming in general and Python in particular, and to those two as well as my mother, Elizabeth Cowper, and my sister, Ruthie Cowper Szamosi, for emotional support and encouragement throughout the writing of this document. Many thanks to Michael Joss for providing me with the source code to ACID and allowing me to base ISAAC on it and release it as open source. And finally, thanks to Maydianne Andrade, Nathan Lovejoy, Mauricio Terebiznik, and Mark Fitzpatrick for providing me with the encouragement and opportunities that started me down this road.

This thesis is dedicated to all the wonderful people who helped make it a reality.

Contents

Abstract **iii**

Acknowledgments **iv**

1 Introduction **1**

1.1 The Integron 1

1.2 Gene Cassettes 3

1.3 Attachment Sites 4

2 Data **8**

2.1 Identifying and Annotating Integrons 8

2.2 Integrase Phylogeny 11

2.3 IntI1 Phylogeny and Cassette Clusters 12

3 ISAAC Software **15**

3.1 Finding the Start 16

3.1.1 attI 16

3.1.2 No attI 17

3.2 Finding attCs 17

3.2.1 Score 1: Structural Scoring 18
3.2.2 Score 2: Sequence-Based Scoring 19
3.3 Cassettes 21
3.4 Parameters 21
3.4.1 Required Parameters 21
3.4.2 Optional Parameters 21

4 Base Frequencies in attC Sequences **24**
4.1 R Boxes 26
4.2 R-L Spacer 27
4.2.1 Sp″ 28
4.2.2 Sp′ 28
4.3 L Boxes 29
4.4 I Boxes 29
4.5 Conclusions 31

5 Discussion **33**
5.1 Future Directions 34

References **36**

A ISAAC Source **40**
A.1 ISAAC Program 40
A.2 ISAAC Functions 45

B Base Frequency Tables **92**
B.1 IUPAC Codes 92

B.2 R Boxes 93
B.3 Spacer Boxes 96
B.4 L Boxes 100
B.5 I Boxes 104

List of Figures

1.1 A cartoon diagram of an integron 2

1.2 A cartoon diagram of a gene cassette with attC site. 3

1.3 A typical attC sequence 5

1.4 Base pairing in R boxes of cassette vs. genomic attCs 6

2.1 An alignment of the attI mutations that we have identified. 11

2.3 IntI1 cladogram with cassette clusters 13

2.2 An integrase phylogeny 14

4.1 Simpson's diversity index for important positions in attC 25

4.2 Base frequencies in IntI1's R boxes 27

4.3 Base pairing in the I boxes 31

List of Tables

2.1 Integrase numbers and genera 10

B.1 The IUPAC ambiguity codes 92

B.2 IntI1 R box base frequencies 93

B.3 IntI4 R box base frequencies 93

B.4 Son R box base frequencies 94

B.5 Vfi R box base frequencies 94

B.6 Vpa R box base frequencies 94

B.7 Xca R box base frequencies 95

B.8 All R box base frequencies 95

B.9 IntI1 Sp box base frequencies 96

B.10 IntI4 Sp box base frequencies 96

B.11 Son Sp box base frequencies 97

B.12 Vfi Sp box base frequencies 97

B.13 Vpa Sp box base frequencies 98

B.14 Xca Sp box base frequencies 98

B.15 All Sp box base frequencies 99

B.16 IntI1 L box base frequencies 100

B.17 IntI4 L box base frequencies 100

B.18 Son L box base frequencies 101
B.19 Vfi L box base frequencies 101
B.20 Vpa L box base frequencies 102
B.21 Xca L box base frequencies 102
B.22 All L box base frequencies 103
B.23 IntI1 I box base frequencies 104
B.24 IntI4 I box base frequencies 104
B.25 Son I box base frequencies 105
B.26 Vfi I box base frequencies 105
B.27 Vpa I box base frequencies 106
B.28 Xca I box base frequencies 106
B.29 All I box base frequencies 107

Chapter 1

Introduction

1.1 The Integron

An integron is a genetic element that bacteria use to capture and disseminate mobile genetic elements known as gene cassettes. There are three components to an integron. The first is the integrase gene, which is a member of the tyrosine recombinase family, and which codes for a protein that catalyzes the gene cassette insertion or excision. The integron integrase is distinguished from other integrases by the presence of an additional domain that contributes to DNA binding and recombination activity (Messier & Roy, 2001). Collis *et al.* (2002) identified a 13-amino acid motif from within this additional domain which can be used to distinguish among the various integron integrase families. The 22 integrase families identified by Collis and colleagues have among them 21 unique IntI motifs, with two integron integrase families (IntI4 and IntI5) sharing a motif (Collis, Kim, Stokes, & Hall, 2002).

The second element required for an integron is the promoter. This promoter is generally located immediately upstream of and overlapping with the integrase gene, and it promotes the expression of the genes in the cassette array. The final element required for a functional integron is an attachment site, attI. This site is also generally located upstream of the integrase gene (downstream of the promoter) and provides a recombination site where the gene cassettes are inserted. A fourth common element to an integron is the array of cassettes it is associated with. Although this array is not necessary to the definition of an integron, it is what the integron controls (figure 1.1).

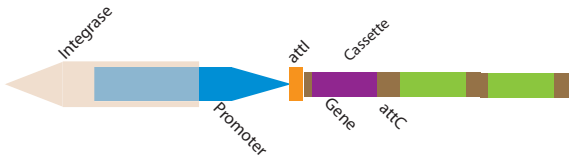

Figure 1.1: A cartoon diagram of an integron located on a bacterial chromosome or plasmid.

Integrons were originally discovered in the context of antibiotic resistance in pathogenic bacteria (Martinez & Cruz, 1990). Frequently associated with mobile elements such as transposons and plasmids, they were believed to be a novel method for accumulating collections of antibiotic-resistance genes in a single bacterial genome, conferring multiple resistance much faster than is possible by mutation alone. Although these mobile integrons were discovered first, it is now understood that these structures are derived from less-mobile chromosomal integrons found in *Vibrio* and *Xanthomonas*, as well as other γ-proteobacteria (Rowe-Magnus et al., 2001). The diverse range of taxa within which chromosomal integrons have been identified, as well as the fact that integrase families tend to cluster within bacterial species, implies that integrons are an ancient mechanism of horizontal gene transfer in bacteria

(Rowe-Magnus & Mazel, 2001). There is also some evidence that even the chromosomal integrons have been gained and lost through transfer of the whole integron multiple times (Nemergut et al., 2008).

Although the distinction between chromosomal and mobile integrons is not always stark (Boucher, Labbate, Koenig, & Stokes, 2007), there is a tendency for integrons that are mobile to have fewer cassettes, and greater diversity within their attC complement (Boucher et al., 2007; Rowe-Magnus & Mazel, 2001). The lower diversity in the attCs of chromosomal integrons suggests to some researchers that these attCs may share a common origin of construction, while those attCs associated with mobile integrons may not. One possible reason for this is that those bacteria with chromosomal integrons are producing their own cassettes, while those with only mobile integrons must acquire cassettes from their environment[1].

1.2 Gene Cassettes

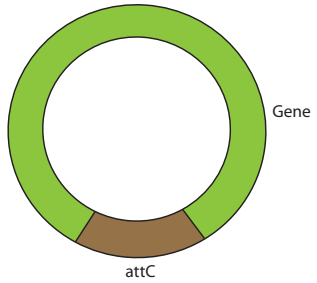

Figure 1.2: A cartoon diagram of a gene cassette with attC site.

Gene cassettes are small pieces of functional DNA that have been packaged so that they can be inserted into an integron. They are comprised of two parts. The first is the information being transferred. This is usually but not always a single gene. The second is the attachment site, attC, which will recombine with attI when the cassette is integrated into an integron's cassette array (Figure 1.2). When a cassette is not integrated, it forms a double stranded covalently closed loop with secondary structure

[1]Rowe-Magnus, p.c.

in the attC. (Collis & Hall, 1992)

A gene cassette is almost always inserted at attI, directly downstream of the promoter, making integrons an especially interesting system because any gene that has been packaged into a cassette can be immediately expressed upon integration, without having to bring its own machinery along.

1.3 Attachment Sites

The attC and attI sites are the attachment sites that recombine when the gene cassettes are spliced into or out of the genome. They consist of a number of structurally important regions, but have very little conserved sequence.

attC

Because of the palindromic nature of the attC, the two strands of the attC each base-pair with themselves to form two hairpin structures. Of these two strands, the strand that the integrase gene does not reside on (conventionally called the "bottom" strand) is overwhelmingly more likely to recombine with attI than the top strand (Bouvier, Demarre, & Mazel, 2005). The attC site is generally between 50 and 150 base pairs long, and is an imperfect inverted palindrome with three structurally defined regions at each end (R, Sp, and L) and a highly variable middle region (figure 1.3). In our analysis, we investigated the composition of these three structural boxes, plus the outermost six bases in the middle region (I).

The R″ and R′ sites consist of the seven outermost bases at each end and have sequences `NNNNAAC` and `GTTNNNN` respectively. They include the core (`GTT`) and inverse

Figure 1.3: A typical attC sequence. The R boxes are outlined in blue (R″ on the left and R′ on the right), the L boxes are outlined in green, and the extrahelical base in L″ is outlined in beige. The Sp boxes are found between R″ and L″ on the left and between L′ and R′ on the right. The I boxes are the two sets of six bases immediately internal to the two L boxes. The extrahelical base in I″ is outlined in purple.

core (`AAC`) recombination sites. These regions are expected to base-pair with each other when the cassette is not integrated, and the two unspecified four-bp sequences have a high probability of being complementary (figure 1.4). The L″ and L′ sites are seven and six base pairs long respectively, start five or six bases internal to the R sites, and consist of two base-pairing triplets with the central (fourth) base in L″ forming an extra-helical bubble that is required for recombination activity. Between the R and L sites are spacer regions (here called Sp″ and Sp′). These regions are structurally conserved as well. The three bases immediately adjacent to R do not pair with each other, and the identities of these bases are important for strand selection during cassette insertion (Bouvier, Ducos-Galand, Loot, Bikard, & Mazel, 2009). The six bases immediately internal to the L boxes at either end (here I″ and I′) appear to be less structurally important than the R, L, and Sp boxes, but they do contain an extrahelical base that contributes to strand selection (Bouvier et al., 2009) and they display some interesting patterns, discussed in chapter 4.

During recombination, the attC site is split in the R′ region immediately after the initial G (Partridge et al., 2000) and most of the R′ site of that cassette ends up at

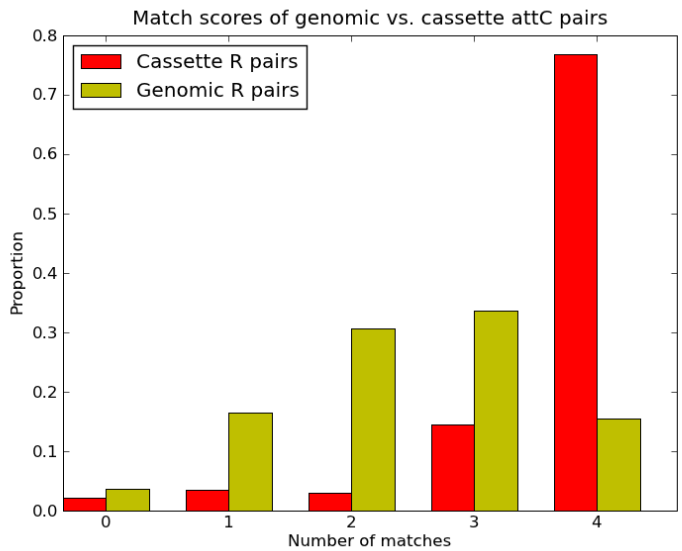

Figure 1.4: The proportion of R″/R′ site pairs in genomic attCs (gold) versus presumptive cassette attCs (red) with 0, 1, 2, 3, or 4 successful base pairings. The attCs were identified using ACID.

the upstream end of the cassette, while the rest of the attC ends up downstream. If two or more cassettes are integrated into the integron then the bulk of the attC of the upstream cassette (R″ - Sp″ - L″ - middle - L′ - Sp′ - `G`) is fused with the post-`G` R′ region of the neighbouring downstream cassette's attC, forming a hybrid attC site which we refer to as a **genomic attC**, to distinguish it from the **cassette attC** described above. Although the bulk of a genomic attC comes from a single cassette, the four unspecified bases in the R sites of a genomic attC come from different cassettes, and are considerably less likely to pair with each other than those of a cassette attC (figure 1.4).

In chapter 3 we introduce new software, ISAAC, that takes advantage of the base pairing patterns between L″ and L′ and between R″ and R′ in cassette attCs to

identify genomic attCs in long DNA sequences.

attI

Much less is known about the attI site, although conserved attI sequences have been identified for the three most common classes of resistance integrons, those associated with IntI1, IntI2, and IntI3 (attI1, attI2, and attI3 respectively) (Hansson, Sköld, & Sundström, 1997). These conserved sequences end in the core site `GTT`, and if a cassette has been integrated at this attI then the split attC of the first cassette contributes the final `TT` and the bases that follow (Partridge et al., 2000). ISAAC is able to identify the upstream boundary of the initial cassette at these attI regions if they conform to any of the three sequences mentioned above, as well as two mutations of attI1 that we have found. Additionally, it is possible for a user to supply their own attI sequence if it is known, provided the sequence ends in the core `GTT`. If the integron's attI site does not conform to one of these, and if the user cannot supply their own attI sequence, then ISAAC can identify the first attC (forming the downstream boundary of the initial cassette), but will not be able to identify the upstream boundary of that cassette.

Chapter 2

Data

The sequences used in these analyses were drawn from the RefSeq other genomic database, which includes the whole genome sequences (including chromosomes and plasmids) of all fully sequenced organisms in the RefSeq databases except mouse and human (NCBI, 2002). All the integrons we identified in those genomes were included in our analysis.

2.1 Identifying and Annotating Integrons

To identify sequences that might contain an integron, we conducted three tBLASTn (Altschul et al., 1997) searches of integrase genes against the database. The three query sequences chosen were IntI1 (the most well-studied integrase) and IntI10 and Gsu (the two most divergent integrase genes) (Collis et al., 2002). The results of the BLAST searches were filtered using custom Python (Python, 2008) scripts and only those that contained one of the 21 IntI-specific motifs (Messier & Roy, 2001; Collis et al., 2002; Nunes-Düby, Kwon, Tirumalai, Ellenberger, & Landy, 1998) in

the appropriate position were retained (Nemergut et al., 2008). The genes were then categorized based on which of the motifs they contained. Those that contained the motif that is common to both IntI4 and IntI5 were aligned with reference sequences of both of those genes (IntI4: AF055586.1, IntI5: AAD55407.2) and were determined to all be IntI4. The search resulted in the identification of 112 potential integrons on 96 plasmids or chromosomes. 13 of the 21 known integrase family motifs were represented and they were distributed among 23 bacterial genera (table 2.1). 79 (70% of the integrons found) were IntI1 (also known as type 1 integrons).

The sequences were initially annotated using the software ACID (Joss et al., 2009). For each integrase gene identified above, a region of sequence was taken from the molecule it occupied that spanned 20kb on either side of the gene, or until the end of the molecule, whichever was shorter. These sequences were uploaded to the ACID website and the attCs and cassettes were annotated separately using the following attC parameters:

- Score 1 Min = 75
- Score 2 used, Min = 0
- Min Length = 50 bp
- Max Length = 150 bp
- Start Point = 0
- Length = total length of sequence

In an initial attempt to identify the upstream boundary of the first cassette in type 1 integrons, we used custom Python scripts to conduct a straight sequence search for attI1 (Hansson et al., 1997) in the regions in between the integrase gene and the first

attC in the type 1 integrons for which attCs had been found, and in the 1000 bases immediately upstream of the integrase gene in the others.

The integrase genes, attC sites, and cassettes (as annotated by ACID) were then stored in an SQLite3 (SQLite3, 2011) database.

Integrase Family	Number	Genera
Gsu	1	*Geobacter*
IntI1	79	*Acinetobacter, Aeromonas, Citrobacter, Comamonas, Corynebacterium, Desulfurispirillum, Edwardsiella, Escherichia, Klebsiella, Pseudomonas, Salmonella, Yersinia*
IntI2	5	*Escherichia, Pseudomonas, Shigella*
IntI3	1	*Escherichia*
IntI4	8	*Vibrio*
IntI9	1	*Vibrio*
IntI10	1	*Pseudoalteromonas*
Neu	4	*Congregibacter, Nitrosomonas*
Son	5	*Shewanella*
Tde	1	*Treponema*
Vfi	2	*Vibrio*
Vpa	1	*Vibrio*
Xca	3	*Xanthamonas*

Table 2.1: The number of representatives of each integrase family, and the genera in which they were found. We use the integrase names from Collis, 2002.

A copy of attI1 was found in all but eight of the 79 IntI1 sequences. We conducted a Clustal W alignment (Larkin et al., 2007) of the attI1 sequence with each of the eight sequences that did not contain an exact match. These aligments yielded six cases where there was no attI1, one instance of attI1 with a single nucleotide insertion, and one instance of attI1 with a single substitution mutation. An alignment of these two mutations with the original attI1 is displayed in figure 2.1.

```
attI1            ACGCAGCAGGGCAGTCGCCCT-AAAACAAAGTT
NC_013446.1      ACGCAGCAGGGCAGTCGCCCTAAAAACAAAGTT
NC_009651.1      ACGCAGCAGGGCAGTCGCCCT-AAAACAAGGTT
                 ******************** ******* ***
```

Figure 2.1: An alignment of the attI mutations that we have identified.

Once the ISAAC software was complete, we used it to annotate regions of the genome on the side of the integrase gene where the cassette array was expected (upstream of the gene for all integrase types except Tde, where the cassette array is normally located downstream of the integrase gene (Colemand, Tetu, Wilson, & Holmes, 2004)), starting at the integrase gene and ending at the end of the molecule being annotated. These new data were also stored in an SQLite3 database.

2.2 Integrase Phylogeny

An amino acid alignment of the integrase genes was produced using MUSCLE (Edgar, 2004), and a nucleotide alignment based on the amino acid alignment was generated with Pal2Nal (Suyama, Torrents, & Bork, 2006). All but one representative of all non-unique sequences were removed from the alignment using custom Python scripts.

We constructed a phylogeny of the genes based on this alignment (figure 2.2) using MrBayes (Huelsenbeck & Ronquist, 2001) version 3.2.1. We checked that the run had converged using awty (Wilgenbusch, Warren, & Swofford, 2004) and visualized the

resulting tree using the Interactive Tree of Life software, iTOL (Letunic & Bork, 2007, 2011). The groupings in our phylogeny are more consistent with those found by Rowe-Magnus and Mazel (Rowe-Magnus & Mazel, 2001) than with those found by Boucher (Boucher et al., 2007).

It is worth noting that two integrase genes classified as Neu by their integron-integrase motif were in fact found in *Congregibacter litoralis* rather than the expected *Nitrosomonas europaea*. The two genes found in *C. litoralis* were identical to each other, and despite sharing their identifying motif with the *N. europaea* integrases, were not monophyletic with them. Also worth mentioning is the fact that the integrase genes matching the Son motif, despite being found on multiple species of *Shewanella*, (one in *S. halifaxensis*, two in *S. baltica*, one in *S. oneidensis*, and one in an unidentified species), and despite showing evidence of existing in both chromosomal (Rowe-Magnus et al., 2001) and mobile (Werbowy, Cieśliński, & Kur, 2009) form, were indeed monophyletic.

2.3 IntI1 Phylogeny and Cassette Clusters

In addition to analysing the relationships between the different types of integrase gene, we endeavoured to determine the rates of cassette gain and loss in type 1 integrons against the benchmark of the rate of evolution of the integrase gene (Hao & Golding, 2006). We clustered all the cassettes found in type 1 integrons transitively into groups of 97% similarity, and attempted to produce a phylogeny of the sequence beginning with the IntI1 gene and extending from the 5' end of the gene to the conserved attI1 sequence. These sequences share an extremely high degree of similarity (often 99 - 100%), so we were unable to produce a well-resolved phylogeny.

Despite this similarity in the integrase gene, there is considerable diversity in the cassette complement, both in the identity of cassettes, as well as their groupings (figure 2.3). It is clear, therefore, that cassette gain and loss is occurring at a higher rate than IntI1 evolution, and that horizontal transfer, rather than descent, is the driving force behind the evolution of the cassette complement.

Complete copies of the scripts used to conduct these analyses, including all MrBayes parameters, can be found online at the following URL:

`http://lalashan.mcmaster.ca/theobio/ISAAC/index.php/JCSz_Masters`

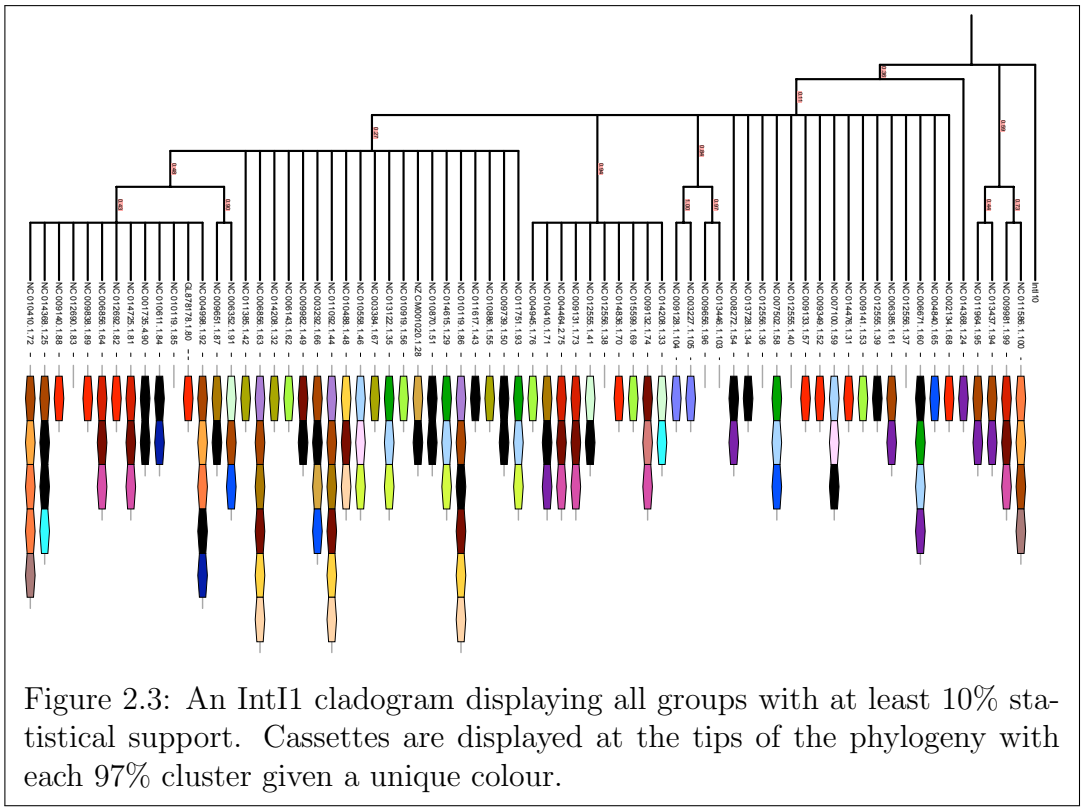

Figure 2.3: An IntI1 cladogram displaying all groups with at least 10% statistical support. Cassettes are displayed at the tips of the phylogeny with each 97% cluster given a unique colour.

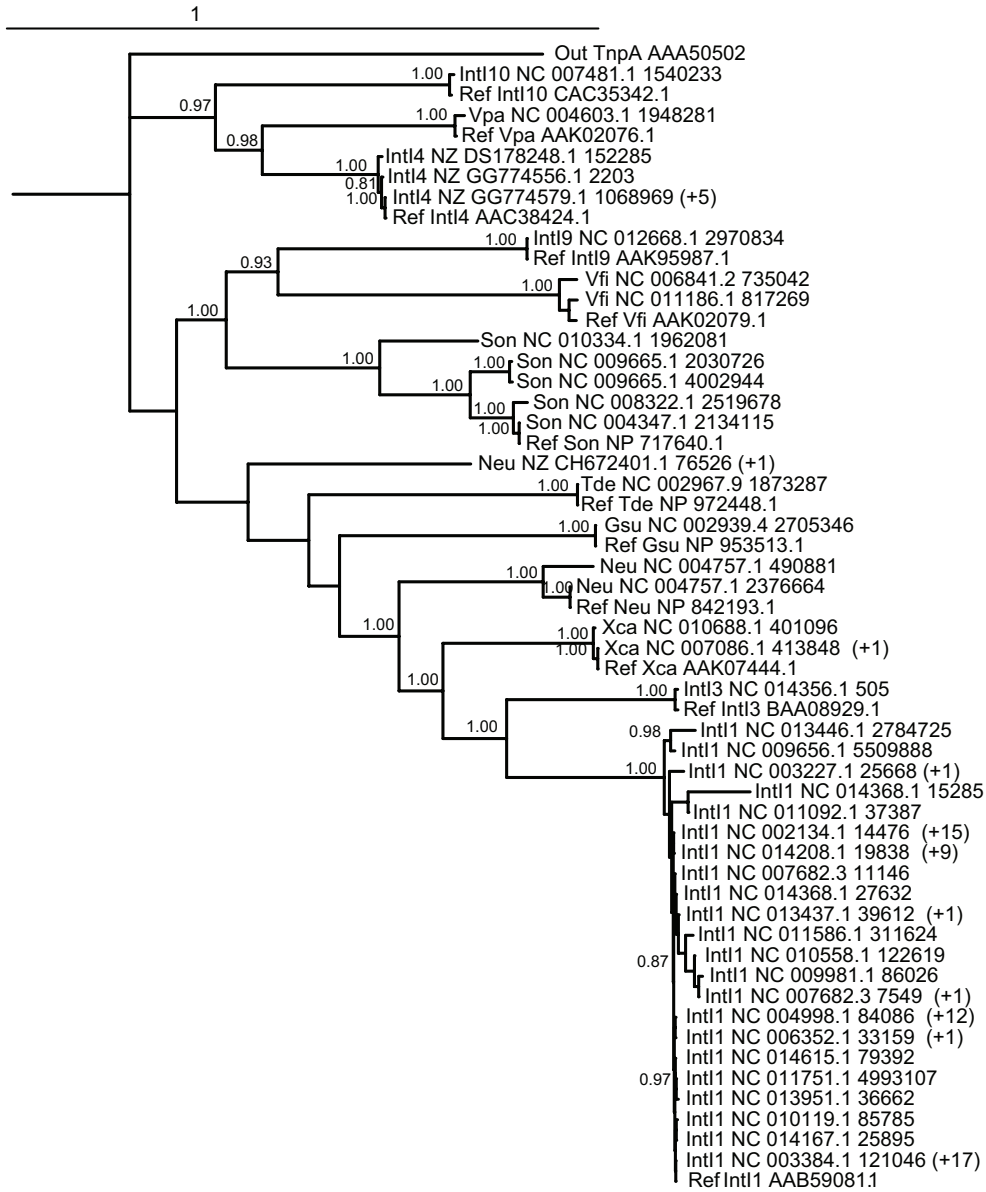

Figure 2.2: An IntI phylogeny. Statistical support is reported for all nodes with support greater than 80%. Counts in parentheses indicate the number of sequences identical to the one indicated. Only one representative of each set of identical sequences was included in the analysis.

Chapter 3

ISAAC Software

We have developed ISAAC, a program which searches for attI and attC sites in DNA sequences and returns detailed annotations of the elements that it finds. The software was developed in Python (versions 2.5 and 2.6) using the Biopython module (Cock et al., 2009) (versions 1.56 and 1.58). ISAAC searches along the length of the sequence in order, identifying putative attI and attC sites, and annotates cassette boundaries thereby. It can also take as input a fasta file of pre-identified attC sequences and return detailed annotations of each one.

Much of ISAAC's function is based on previous software called ACID, developed by Michael Joss (Joss et al., 2009); however, ISAAC improves upon ACID in the following ways: unlike ACID, ISAAC can be run locally on any Unix-based computer instead of through a web browser, and therefore can handle longer sequences without encountering problems such as http request timeouts. ISAAC's scoring of the R boxes is based on the fact that the R″ and R′ boxes of cassette attCs are expected to be complementary (or on the frequencies of specific nucleotides at specific positions when the complementary R box is not available), which will be more accurate than

ACID's simple binary scoring, which assumes (not always correctly) that the four variable bases in the R boxes will be of the form `RRRY` (see table B.1 for the complete listing of the IUPAC ambiguity codes used in this document). The final improvement upon ACID is in the kind of data output and the flexibility of the code. In addition to providing the user with the position and sequence of all attCs found, ISAAC returns detailed annotations of each attC, including the positions and sequences of the structural boxes. ISAAC's modular code also allows the integration of parts of its functionality into other scripts. These advantages will enable easier analysis of the sequences, as demonstrated in chapter 4.

3.1 Finding the Start

ISAAC can find the beginning of the cassette array in two ways, either by identifying an attI sequence, or by finding the first candidate attC.

3.1.1 attI

ISAAC comes with a list of known attI sequences. Additionally, the user can provide an attI sequence of their own if desired. Assuming the `--no-attI` flag is not used[1], ISAAC will begin by searching the subject sequence for attI sequences using simple string matching. If no attI sequence is found, ISAAC will stop and report a failure to find any integrons. If an attI sequence is found, ISAAC will store the attI sequence, as well as the R′ sequence it is adjacent to, and move on to the attC search.

[1]For a full description of the arguments ISAAC takes, see section 3.4.

3.1.2 No attI

If the `--no-attI` flag is used, ISAAC will not search for an attI, but instead search for the first candidate attC as described in section 3.2; however, because it will not have an initial R′ box to match to, the R″ sequence of the first attC will be scored using base-frequency scoring as described in section 3.2.2.

3.2 Finding attCs

Once a starting attI or attC is found, ISAAC crawls along the sequence searching for potential R″ sites, which will be of the form `NNNNAAC` (here called candidate starts). Once a candidate start is found, ISAAC searches forward for a potential R′ site, `GTTNNNN` (candidate end), which will form the other boundary of a possible attC. An `AAC` - `GTT` pair only represents a potential candidate attC if the length of the sequence bounded by `NNNNAAC ... GTTNNNN` inclusive is between the minimum and maximum attC lengths. These values are 50 and 150 respectively by default, but can be changed by the user.

Once an appropriate candidate end site has been found, ISAAC scores the sequence using a structure-based scoring method based on that pioneered in ACID (Joss et al., 2009) (see section 3.2.1) and a novel sequence scoring method (section 3.2.2).

The forward search for candidate ends (`GTT`) is continued until the maximum attC length is reached. Once sequences bounded by all candidate ends have been scored, the best-scoring potential candidate attC for this candidate start is returned and compared against the cutoffs for the two scoring mechanisms. If the cutoffs are met,

that candidate attC is stored, the R′ is updated, and ISAAC begins searching for the next candidate start (`AAC`) after the end of the current candidate attC. If the cutoffs are not met, ISAAC tests the candidate attC's complement in case the attC is simply oriented the wrong way. If the complement also cannot meet the cutoffs, ISAAC gives up on the current candidate start and begins searching for the next one. Unlike ACID, ISAAC's algorithm does not allow us to find overlapping candidate attCs; however, this is necessary in order to compare the R sites of cassette attCs, and since overlapping attC sites are not a biological possibility and attC sequences are short, any real attCs that are masked by a rare false positive would be very close to the attC identified by ISAAC.

ISAAC will continue searching for candidate starts until the maximum cassette length has been exceeded (default is 4000 nucleotides, but this can be changed by the user) or the end of the sequence is reached.

3.2.1 Score 1: Structural Scoring

L Score

The L sites are scored in three steps. First, the two triplets are given one point per match, for a maximum of three points per triplet pair (figure 1.3). Then a point is added to the score of each triplet pair for each triplet in that pair that does not consist of three identical bases. This brings the maximum score per triplet up to five. The two triplet scores are then multiplied (score range: 0 - 25) and doubled (0 - 50). The fourth base in the L″ box is then examined, and one point is added to the score (bringing the maximum up to 51) if this base does not base pair with the fourth-last base in L′, and therefore must be extrahelical.

Because the spacers between the R and L boxes may be either five or six bases in length (Stokes, O'Gorman, Recchia, Parsekhian, & Hall, 1997), there are four possible combinations of L sites in a given potential candidate attC. Unlike ACID, which allows Sp′ to be five or six bases long, but required Sp″ to be five, ISAAC scores all four combinations and the best-scoring annotation is kept.

R Score

The R boxes are then scored in a novel way. The R″ box is compared to the R′ box of the previous attC or attI and given one point per match of the four variable bases, for a maximum score of four. Because the R″ box of the next attC (if it exists) is not available at this point, the R′ box is scored using the base-frequency scoring method described in 3.2.2, but only applied to the four variable bases in R′.

3.2.2 Score 2: Sequence-Based Scoring

ISAAC takes advantage of the fact that attC sequences, while not well conserved, nonetheless have some fairly robust patterns of bases in given positions[2], to eliminate false positives identified by Score 1. Unlike ACID, which scores the outer 25 bases at each end based solely on their absolute position, ISAAC takes into account the lengths of the spacer regions. Because the length of the spacer regions between the R and L boxes are variable, the different structural pieces are scored independently and then the scores are summed to give a final score.

A preliminary database of attC sequences identified by ACID was assembled and the sequences of eight separate structural elements (boxes) were analyzed. The eight

[2]The base frequencies and their implications are discussed in some detail in chapter 4

boxes are R′, R″, L′, L″, the two spacers between the R and L boxes (Sp′, Sp″), and the two sets of six bases immediately inside the two L boxes (I′, I″). For each box, the frequency of each base at each position was ascertained (base frequency). New attCs are scored as follows:

Once Score 1 has been found and the parts of the attC are known, each nucleotide position in each of the eight structural boxes is examined, and the box is scored using equation 3.1 where n is the number of positions in the box, f_i is the frequency with which the base found at position i in this sequence is found at position i in all the sequences, and f_{max_i} is the frequency of the most common base at position i.

$$\sum_{i=1}^{n} f_i / f_{max_i} \tag{3.1}$$

The only exception is when an Sp box is six bases long instead of five. In this case the score is scaled to prevent this sequence having an advantage by virtue of its extra base (equation 3.2).

$$5 * (\frac{\sum_{i=1}^{n} f_i / f_{max_i}}{6}) \tag{3.2}$$

Once each box has been scored, the scores of all eight boxes are added together to give the final values of Score 2. The maximum possible value for Score 2 is 49, and a cutoff of between 20 and 25 is recommended for an optimal tradeoff between eliminating false positives and retaining true positives.

3.3 Cassettes

Once all candidate attCs in a sequence have been found, ISAAC will annotate cassettes. It defines cassettes as starting at the second nucleotide of the R′ box of the preceeding attI/attC (the first `T` in the `GTT`) and ending at the first nucleotide of the R′ box of the next attC (the `G` of the `GTT`). Unlike ACID, which is off by six bases when the attC is oriented in the wrong direction, ISAAC finds the correct cassette boundary regardless of the orientation of the attC. ISAAC can also provide start and end points of the "inner" portion of the cassette, i.e. that part that excludes the attC.

3.4 Parameters

3.4.1 Required Parameters

The following parameters must be specified by the user:

- `--in` or `--in-file`: Takes the name of the fasta file containing the sequence to be annotated. If `--attC` is not used, only the first sequence in the file will be annotated
- `--out` or `--out-file`: Takes the base name of the output files to be written.

3.4.2 Optional Parameters

These parameters have default values, but can be changed by the user:

- `--ci` or `--cass-inner`: A flag. If used, the cassette sequences that are returned

will exclude their attCs. If not used, the cassette sequences will include the pieces of attC on either end that belong to them.

- `--attISeq`: Takes an attI sequence. The sequence must end in `GTT`.
- `--nattI` or `--no-attI`: A flag. If used, ISAAC will not search for an attI but will instead search for an attC to start with. Recommended if the attC sequence of the particular integrase is not known or if an attempted annotation turned up nothing.
- `--co1` or `--score1-cutoff`: Takes an integer between 0 and 59, specifies the minimum acceptable value for score 1. Defaults to 44.
- `--co2` or `--score2-cutoff`: Takes an integer between 0 and 49, specifies the minimum acceptable value for score 2. Defaults to 20.
- `--attC` or `--attC-only`: A flag. If used, the program will assume that each sequence in the input file is a single attC, and will score and annotate each one.
- `--aml` or `--attCminLen`: Takes the minimum allowable length of attC. Must be at least 20. Defaults to 50.
- `--aMl` or `--attCmaxLen`: Takes the maximum allowable length of attC. May not be less than `--aml`. Defaults to 150
- `--cml` or `--cassminLen`: Takes the minimum allowable length of a cassette, excluding the attC. Defaults to 0 (i.e. allows attCs to be immediately adjacent.)
- `--cMl` or `--cassmaxLen`: Takes the maximum allowable length of a cassette, excluding the attCs. This defines how far after the previous attC the program

should look before concluding that there are no more attCs to be found. A value of 0 means there is no maximum and ISAAC will search to the end of the sequence. Defaults to 4000. May not be less than `--cml`.

- `--dir` or `--direction`: Takes the direction in which the program will search. May only take values 1 (forward), -1 (reverse), and 0 (both). Defaults to 1.

- `--sp` or `--start-position`: Takes the position in the given sequence at which you want the program to start looking for attCs. Defaults to 1 (the first base in the sequence). Interacts with the `--dir` flag such that if `--dir` is set to -1 the values of `--sp` will refer to that base in the input sequence's complement. If `--dir` is set to 0, it is best to leave this at 1.

- `--ep` or `--end-position`: Takes the position in the given sequence at which the search for attCs will be stopped even if the maximum cassette length has not been exceeded. Interacts with `--dir` as `--sp` does. Not recommended for use with `--dir 0`. If this option is not used, or if a value is given that is greater than the length of the sequence, then ISAAC will search to the end of the sequence or until `--cMl` is exceeded. If this parameter is used the value of `--ep` must be greater than that of `--sp`.

Chapter 4

Base Frequencies in attC Sequences

Elaborate measures such as those found in ACID and ISAAC are required to identify attCs because their sequences can be highly variable. Aside from the core (`GTT`) and inverse core (`AAC`) sites, attCs are defined primarily by their structure rather than their sequence. There is, however, much interest in determining whether specific bases at specific positions are required for the functioning of the integron. Empirical studies have shown that the presence of certain conserved elements beyond the core sites (all located in the outer sections of the attC sequences) are required for full insertion functionality (Johansson, Kamali-Moghaddam, & Sundström, 2004; Bouvier et al., 2009). Here I investigate the frequencies of specific nucleotides in the outer portions of the attCs, to see if some positions are more well-conserved than others, possibly implying purifying selection.

These analyses were conducted on the attCs identified by ISAAC's `find_all_attCs` function (appendix A) with the default parameter values specified in chapter 3 except

a maximum cassette length of 2000 was used. The attCs were analyzed all together and in groups based on integrase motif type. Only those integrase types for which we had at least 20 associated attCs were included in the separate analysis. These were IntI1 (for which we had 153 attCs), IntI4 (664), Son (22), Vfi (97), Vpa (69), and Xca (57) (to see which genera the different interases are associated with, refer back to table 2.1). We had 1107 attCs in total.

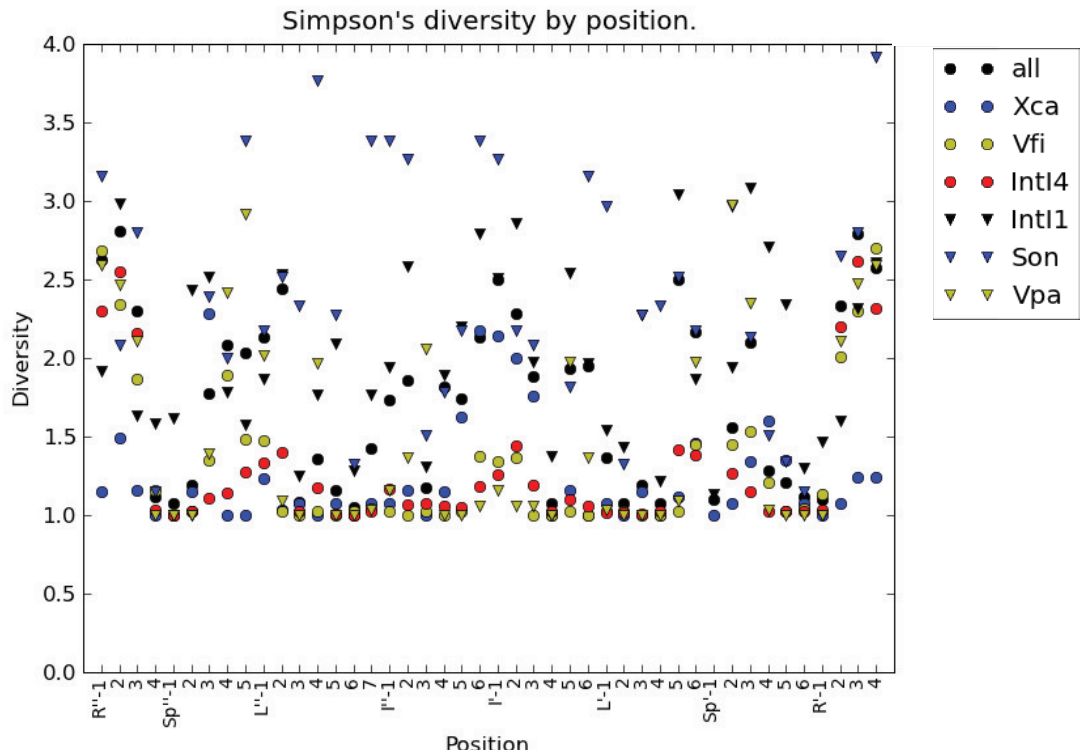

Figure 4.1: The Simpson's diversity index (equation 4.1) of each position in the four outer boxes at each end. Taken for all attCs.

We calculated the Simpson's diversity index at each position (equation 4.1 where

f_x represents the frequency of nucleotide x at a given position), and found that there

$$\frac{1}{\sum_{n=1}^{4}(\frac{f_n}{\sum_{i=1}^{4} f_i})^2} \tag{4.1}$$

were some robust patterns of conserved positions within each integron type. We also found that the different integron types had different degrees of sequence variability in their associated attCs (figure 4.1). Unsurprisingly, IntI1 had quite a high degree of variability, while Vfi, Vpa, Xca, and IntI4 were much more conserved. The attCs associated with Son were also quite variable, which is not too surprising since *Shewanella* is believed to harbour both mobile (Werbowy et al., 2009) and chromosomal (Rowe-Magnus et al., 2001) integrons (figure 4.1). Additionally, we found some systematic differences in conserved attC positions among the different integrase types.

4.1 R Boxes

Since Stokes *et al.*'s seminal paper on the structure of attCs (Stokes et al., 1997), it has been believed that the R boxes were generally of the form `GTTRRRY` and `RYYYAAC`. Our data show that, while this is weakly true for attCs associated with IntI1 (table B.2, figure 4.2), the pattern doesn't hold at all for some integrase types (see tables B.3 - B.8). With the exception of those attCs associated with Xca, the first three of the four variable bases in the R boxes were very poorly conserved (figure 4.1). The only obvious pattern in these boxes was that the bases immediately adjacent to the core site were `T` in R″ and `A` in R′ with very high frequency across all integrase types.

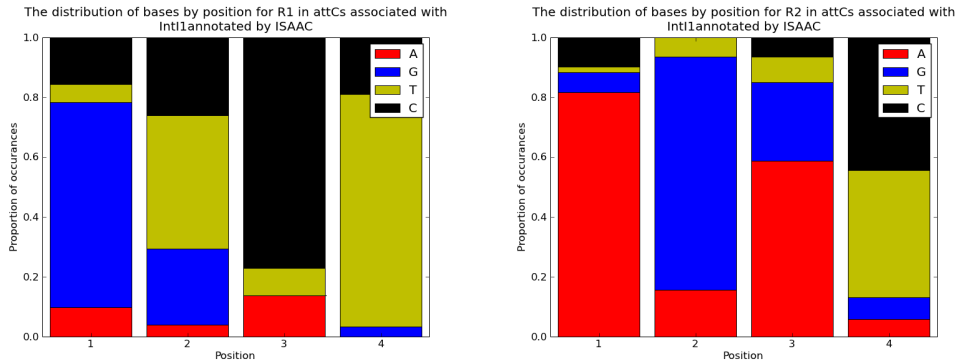

Figure 4.2: The base frequencies at each of the four variable positions in R″ (left) and R′ (right) in attCs associated with IntI1.

4.2 R-L Spacer

There is a spacer between the R″ and L″ boxes (Sp″) and another between the R′ and L′ boxes (Sp′). Lengths of five or six bases have been attested for Sp′ (Stokes et al., 1997), but Sp″ has only been attested as five bases long. Using ISAAC, we investigated the possibility that Sp″ may be six bases long. We found very few attCs that conformed to that structure (31 out of 1107 annotated attCs) and those attCs had low values for Score 2 (maximum 26.9, mean 22.5). We believe that this is consistent with five bases being the only allowable length for Sp″, and will be eliminating the possibility of a six-base Sp″ from future versions of ISAAC. Six-base-long Sp′s were also in the minority (181 out of 1107), but this structure has been attested previously, and appears to be the norm, rather than the exception, in attCs associated with Xca and IntI1. In analyzing the base frequencies of the Sp boxes by integrase type, we only report the results from those that are five bases long for all Sp″s and for Sp′s in all integron types except IntI1 (where 100 out of 153 are six bases long) and Xca (56 out of 57).

4.2.1 Sp″

The three bases closest to the R boxes form an extrahelical bulge and have been shown to be essential for recombination (Bouvier et al., 2009). These three bases are the most well conserved and do seem to conform approximately to the sequence `AAA` as predicted by experiment, although the third base does display some systematic differences depending on the integron type.

Even at these extremely important initial positions, the Sp″ sequences associated with IntI1 are more varied than those associated with other integrases. They do retain the `A` plurality in the first two positions (table B.9), while the third position is highly varied. In contrast to most of the integrase types, `T` has the plurality here.

The first three positions of Sp″ tend to be more well conserved across the other integrase types, with `A` in the majority in the first two positions in all integron types and in the third position in all but two integron types (Son (`G`) and Xca (`T`)) (tables B.10 - B.14).

The last two bases in Sp″ are considerably more variable both within and among integrase types.

4.2.2 Sp′

The three bases closest to the R box in Sp′ are the final three. These three also appear to be more highly conserved than the other bases in this box, and generally conform to the pattern `GGC`. Unlike in Sp″, there do not appear to be systematic differences among integrase types in this region. In Sp′ overall, we observed that the most common base at each position in IntI1's 6-base Sp′s were the same as those in the 6-base Sp′s of Xca.

4.3 L Boxes

The L boxes, especially the regions closest to the Sp boxes, contain some of the least variable bases in the attCs (figure 4.1). Our findings of base frequencies are consistent with Johansson *et al.*'s findings that certain bases in the L boxes are required for binding with IntI. We found high degrees of consistency in the third base of the first triplet (L″-3 and L′-4) as well as the first two bases of the second triplet (L″-5, 6 and L′-3, 2) (see the tables in Appendix B.4). Despite Johansson *et al.*'s conclusion that the identities of the last base in the second triplet (L″-7 and L′-1) and the extra-helical base (L″-4) do not affect integration, we found very low frequencies of `A` and `T` at this position, implying that these bases might inhibit insertion activity. The L boxes also lacked systematic differences among integrase types except for the differences in variability that are consistent across the boxes.

4.4 I Boxes

Consistent with the findings of the other boxes, the positions in these boxes are remarkably well-conserved across all integron types except IntI1 and Son. Unlike the other boxes; however, the I boxes display fairly extreme systematic differences among integron types, not only in sequence, but in the apparent structure likely to be formed by that sequence.

The majority-base at each position in IntI1's I″ and I′ boxes are `CCGACS` and `GSGCGG` respectively. Only the first three bases pair reliably when no bases are assumed to be extrahelical, but base pairing is improved noticeably if the fourth base of I″ is assumed to be extrahelical (figure 4.3 A). This is consistent with previous

findings that the fourth base in many attCs I″ boxes are extrahelical (Bouvier et al., 2009).

IntI4's I boxes tend to be of the form `AGGGAC` / `CAGCCC`, with base pairing occurring only at the second and third positions when no bases are extrahelical. Consistent with previous findings that the first base is extrahelical in those attCs found in *Vibrio cholera* (Bouvier et al., 2009), the base pairing in this attC type improves somewhat if the first base in I″ is removed (figure 4.3 B).

Vfi's I boxes are generally of the form `ACGATT` / `AAACGT`, with base pairing occurring at all but the fourth base. Base pairing is not reduced if the fourth base of I″ is assumed to be extrahelical (but is if the first base is removed), so this attC type is more consistent with those of IntI1 (figure 4.3 C).

Vpa's I boxes are somewhat more variable, and are of the form `AGKGAT` / `TCACMC`. Base pairing only seems to occur at the second and third positions when there is no extrahelical base. As with IntI4, base pairing is improved markedly by the assumption that the first base of I″ is extrahelical (figure 4.3 D).

Xca's I boxes are more variable still, with a general pattern of `CCGAMS` / `RSKCGG`, but appear to undergo a higher rate of base pairing than Vpa, with the first three positions all being likely to base pair. In this attC type, base pairing is again improved by the assumption that the fourth base of I″ is extrahelical (figure 4.3 E).

Son's I boxes are too variable to draw any inferences at all about sequence, pairing, or extrahelical bases (see table B.25). In the five integron types where the extrahelical base can be determined, we find that regardless of the position of that base, the most common nucleotide at that position is always `A`.

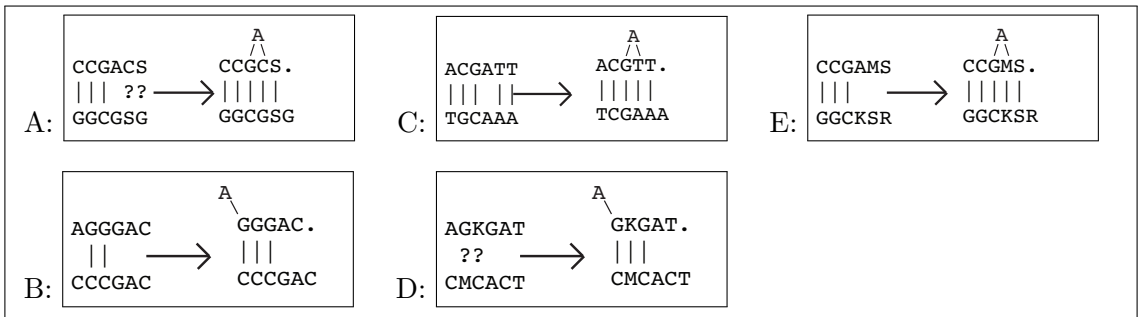

Figure 4.3: Base pairing in the I boxes. The sequences shown employ the majority base at each position. I″ is on the top, and I′ is reversed on the bottom. A: Those attCs associated with IntI1. The fourth base of I″ seems most likely to be extrahelical. B: IntI4. The first base is extrahelical. C: Vfi. The fourth base is extrahelical. D: Vpa. The first base is extrahelical. E: Xca. The fourth base is extrahelical.

4.5 Conclusions

Throughout the various boxes, we observe consistently that when there are systematic differences among integron types, IntI1 tends to cluster with Xca. The other types may cluster together (as with the length of Sp′) or may split so that some cluster with Xca and IntI1 and the others form another cluster (e.g. L′ is most likely to start with `G` in Son and Vfi, `C` otherwise, and the first base of I″ is likely to be extrahelical in IntI4, Son, and Vpa, fourth otherwise). These clusters are broadly consistent with the integrase phylogeny (figure 2.2), which shows that Son is sister to the group containing Vfi, that IntI4 and Vpa are sister to each other, and that Xca is closely related to IntI1. Additionally, we found that those attCs with six bases in the Sp′ box tend to have similar sequences in those boxes, regardless of whether the attC was associated with Xca or IntI1, implying that those attCs with a longer Sp′ may share a common origin.

We found surprisingly high variability in the attC types associated with mobile integrons. Although we expected these integron types to carry more variable attCs

than the chromosomal integrons (since bacteria harbouring chromosomal integrons are believed to produce their own cassettes, while those harbouring only mobile integrons are believed to acquire theirs from the environment) the variation we found was too high to be explained by a simple mixing of attCs from the known chromosomal integrons. For example, despite previous findings of the importance of the first three positions in Sp″ (Bouvier et al., 2009), and despite the strong agreement among the chromosomal integrons that `A` must occupy the first two positions, the attCs associated with IntI1 display considerably more diversity than would be caused by a mixing of the chromosomal attCs, with `A` occurring at the first position only 77% of the time, and only 59% at the second position. This pattern is repeated at the complement of these positions, the final three positions of Sp′. Despite `G` never having a frequency of less than 80% in any of the chromosomal integrons analysed here, and having an overall frequency of 92%, its frequency is only 35% among the IntI1 attCs with 5-base Sp′s, and 74% in those with 6-base Sp′s. In those attCs associated with Son, `T` only occupies the third position in L″ 45% of the time, despite its frequency at this position never dropping below 96% in the chromosomal integrons, and despite it occurring at 87% frequency even with IntI1. Finally, the high variability in IntI1's L boxes also cannot be explained by systematic differences among other known integrase types.

Chapter 5

Discussion

Integrons are a significant contributor to bacterial genome evolution, and a clear understanding of their functioning is important, not only to the study of antibiotic resistance, but in understanding the evolution of bacterial genomes more broadly (Rowe-Magnus & Mazel, 2001). Functions from the ISAAC software were used to annotate these integrons, and further Python scripts were created to analyse the base frequencies in the attC sites that were found. Although ISAAC can certainly be run as a whole piece of software on any UNIX-based machine with Python (version 2.6) and Biopython (version 1.56) installed, we chose an application of ISAAC that uses its functions independently in part to demonstrate how flexible ISAAC is. Less computer-savvy users who simply need a list of attC sites or cassettes may use ISAAC as provided, but for the more technically oriented bioinformaticians, the high degree of modularity in ISAAC's open-source code allows for a lot of flexibility in terms of controlling the output format, and ISAAC's functions may be used on their own when creating new scripts to analyse integron sequence data, as was done here.

Our analysis of the base frequencies at the outer positions of the attC sequences

showed that there is broad consistency between the clustering of integrase genes in their phylogeny, and the clustering of patterns in attC sequences from the different integron types. This may imply that the mechanism by which attC sequences are created is evolving in tandem with the integrase genes, and that IntI1 may retain some of its cassettes as a legacy of its pre-mobile history. We also found more variation in the mobile integrons associated with IntI1 and Son than can be explained by a simple mixing of sequences from the other chromosomal integrons we found. This is true even at positions that have been shown experimentally to be very important to attC functionality. None of the Son integrase genes, and only nine of the 79 IntI1 genes used in our analysis, contain internal stop codons or any other apparent serious mutations, so it seems unlikely that decay caused by lack of use is responsible for the high variability in these attC sequences. It may be that there exist one or more heretofore undiscovered integron types, perhaps with slightly different mechanisms of either attC creation or integration, that are acting as a source for these different cassettes that are picked up by IntI1 and Son. Detailed investigation of the Son integrase gene's functionality also seems indicated.

5.1 Future Directions

Given the high degree of variability in IntI1- and Son-associated attCs, it may be interesting to investigate whether certain unusual features tend to co-occur in the same attCs. This could provide some insight into the mechanism behind these different attC types. Data output from ISAAC could also be used to investigate not only the patterns of which bases tend to occur at which positions, but also which positions in the boxes are more or less likely to base pair or be extra-helical.

ISAAC may also be used even on sequences that are not associated with integrase genes. Running ISAAC on the genomes of bacteria without known integrase genes may turn up stranded cassette arrays or lead to the discovery of novel integrase families.

In terms of improvements to ISAAC, we hope to take advantage of the fact that different integron types have different base frequencies in their attC sequences. In the future, ISAAC may come bundled with multiple sets of base-frequency backgrounds against which to score candidate attCs, based on the integrase type.

References

Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., et al. (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, *25*(17), 3389-3402.

Boucher, Y., Labbate, M., Koenig, J., & Stokes, H. W. (2007, Jan). Integrons: mobilizable platforms that promote genetic diversity in bacteria. *Trends in Microbiology*, *15*(7), 301–309.

Bouvier, M., Demarre, G., & Mazel, D. (2005, Dec). Integron cassette insertion: a recombination process involving a folded single strand substrate. *EMBO Journal*, *24*(24), 4356-4367.

Bouvier, M., Ducos-Galand, M., Loot, C., Bikard, D., & Mazel, D. (2009). Structural features of single-stranded integron cassette attc sites and their role in strand selection. *PLoS Genetics*, *5*(9), e1000632.

Cock, P. J. A., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., et al. (2009). Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, *25*(11), 1422-1423.

Colemand, N., Tetu, S., Wilson, N., & Holmes, A. (2004). An unusual integron in treponema denticola. *Microbiology*, *150*(11), 3524-3526.

Collis, C. M., & Hall, R. M. (1992). Gene cassettes from the insert region of integrons

are excised as covalently closed circles. *Molecular Microbiology*, *6*(6), 2875-2885.

Collis, C. M., Kim, M. J., Stokes, H. W., & Hall, R. M. (2002). Integron-encoded inti integrases preferentially recognize the adjacent cognate atti site in recombination with a 59-be site. *Molecular Microbiology*, *46*(5), 1415-1427.

Edgar, R. C. (2004, Mar). Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, *32*(5), 1792-1797.

Hansson, K., Sköld, O., & Sundström, L. (1997). Non-palindromic atti sites of integrons are capable of site-specific recombination with one another and with secondary targets. *Molecular Microbiology*, *26*(3), 441-453.

Hao, W., & Golding, G. B. (2006, Jan). The fate of laterally transferred genes: life in the fast lane to adaptation or death. *Genome Research*, *16*, 636-643. Available from `http://genome.cshlp.org/content/16/5/636.full`

Huelsenbeck, J. P., & Ronquist, F. (2001). Mrbayes: Bayesian inference of phylogenetic trees. *Bioinformatics*, *17*(8), 754-755.

Johansson, C., Kamali-Moghaddam, M., & Sundström, L. (2004, Jul). Integron integrase binds to bulged hairpin dna. *Nucleic Acids Research*, *32*(13), 4033-4043.

Joss, M. J., Koenig, J. E., Labbate, M., Polz, M. F., Gillings, M. R., Stokes, H. W., et al. (2009, Jan). Acid: annotation of cassette and integron data. *BMC Bioinformatics*, *10*(1), 118.

Larkin, M. A., Blackshields, G., Brown, N. P., Chenna, R., McGettigan, P. A., McWilliam, H., et al. (2007). Clustal w and clustal x version 2.0. *Bioinformatics*, *23*(21), 2947-2948.

Letunic, I., & Bork, P. (2007). Interactive tree of life (itol): an online tool for phylogenetic tree display and annotation. *Bioinformatics*, *23*(1), 127-128.

Letunic, I., & Bork, P. (2011). Interactive tree of life v2: online annotation and display of phylogenetic trees made easy. *Nucleic Acids Research*, *39*, W475-W478.

Martinez, E., & Cruz, F. de la. (1990). Genetic elements involved in tn21 site-specific integration, a novel mechanism for the dissemination of antibiotic resistance genes. *EMBO Journal*, *9*(4), 1275-1281.

Messier, N., & Roy, P. H. (2001). Integron integrases possess a unique additional domain necessary for activity. *Journal of Bacteriology*, *183*(22), 6699-6706.

NCBI. (2002, October). The reference sequence (refseq) project. In (chap. 18). Bethesda, MD: National Library of Medicine. Available from `http://www.ncbi.nlm.nih.gov/books/NBK21091`

Nemergut, D. R., Robeson, M. S., Kysela, R. F., Martin, A. P., Schmidt, S. K., & Knight, R. (2008, Jan). Insights and inferences about integron evolution from genomic data. *BMC Genomics*, *9*(1), 261-272.

Nunes-Düby, S. E., Kwon, H. J., Tirumalai, R. S., Ellenberger, T., & Landy, A. (1998). Similarities and differences among 105 members of the int family of site-specific recombinases. *Nucleic Acids Research*, *26*(2), 391-406.

Partridge, S. R., Recchia, G. D., Scaramuzzi, C., Collis, C. M., Stokes, H. W., & Hall, R. M. (2000). Definition of the atti1 site of class 1 integrons. *Microbiology*, *146*(11), 2855-2864.

Python. (2008). *Python (versions 2.5 and 2.6).* Available from `http://www.python.org`

Rowe-Magnus, D. A., Guerout, A. M., Ploncard, P., Dychinco, B., Davies, J., &

Mazel, D. (2001). The evolutionary history of chromosomal super-integrons provides an ancestry for multiresistant integrons. *Proceedings of the National Academy of Sciences of the United States of America*, *98*(2), 652-657.

Rowe-Magnus, D. A., & Mazel, D. (2001). Integrons: natural tools for bacterial genome evolution. *Current opinion in microbiology*, *4*(5), 565-569.

SQLite3. (2011). *Sqlite3 (versions 3.1.3 and 3.7.7.1).* Available from `http://www.sqlite.org/`

Stokes, H. W., O'Gorman, D. B., Recchia, G. D., Parsekhian, M., & Hall, R. M. (1997). Structure and function of 59-base element recombination sites associated with mobile gene cassettes. *Molecular Microbiology*, *26*(4), 731-745.

Suyama, M., Torrents, D., & Bork, P. (2006, Jul). Pal2nal: robust conversion of protein sequence alignments into the corresponding codon alignments. *Nucleic Acids Research*, *34*, W609-W612.

Werbowy, K., Cieśliński, H., & Kur, J. (2009). Characterization of a cryptic plasmid psfkw33 from shewanella sp. 33b. *Plasmid*, *62*(1), 44-49.

Wilgenbusch, J. C., Warren, D. L., & Swofford, D. L. (2004). *Awty: A system for graphical exploration of mcmc convergence in bayesian phylogenetic inference.* Available from `http://ceb.csit.fsu.edu/awty`

Appendix A

ISAAC Source

A.1 ISAAC Program

ISAAC_prog.py contains the executable program.

```
#!/usr/bin/env python

from optparse import OptionParser
import ISAAC_funcs as apf
import pickle

#function definitions might go here once development is done

##############################################################################
#Define Variables
##############################################################################
attIfnm = 'attIs.txt'
rbd = pickle.load(open('rbd.pickle'))
#lbd1 = pickle.load(open('lbd1.pickle'))
#lbd2 = pickle.load(open('lbd2.pickle'))
bds = pickle.load(open('all_bfds.pickle'))

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

```
#Start
##############################################################################
if __name__ == '__main__':

    ##########################################################################
    #Parse all the options with optparse
    ##########################################################################
    print 'Parsing inputs...'
    parser = OptionParser()

    options,args = apf.do_args(parser)


    ##########################################################################
    #Check all the inputs and set variables
    ##########################################################################
    print 'Checking input values...'
    msg = ''
    #an input fasta file must be provided
    if not options.in_file:
        msg += '\nA fasta file must be provided using the -in or --in-file'+\
                    ' flag.\n'
    else:
        fasta = options.in_file

    if not options.out: #an output file name must be provided
        msg += '\nA name for the output file must be provided using the'+\
                    '-out or --out-file flag.\n'

    else:
        outfnm = options.out

    inner = options.ci #this doesn't get checked

    if options.attISeq: #attI sequences must end in GTT
        if options.attISeq[-3:].upper != 'GTT':
            msg += '\nThe attI sequence that you provide using the -attISeq'+\
```

```
                    ' flag must end in GTT. You may choose not to use '+\
                    'this flag, in which case the default will be used.\n'
        else:
            attIseq = options.attISeq
    else:
        attIseq = False

    nattI = options.nattI #this doesn't get checked

    if (options.co1 < 0) or (options.co1) > 59:
        msg += '\nThe score 1 cutoff provided using the flag -co1 or --score1'+\
                '-cutoff must be an integer between 0 and 59. You may choose'+\
                ' not to use this flag, in which case the default value of 75'+\
                ' will be used.'
    else:
        co1 = options.co1

    if (options.co2 < 0) or (options.co2) > 50:
        msg += '\nThe score 2 cutoff provided using the flag -co2 or --score2'+\
                '-cutoff must be an integer between 0 and 50. You may choose'+\
                ' not to use this flag, in which case the default value of 75'+\
                ' will be used.'
    else:
        co2 = options.co2

    attCo = options.attCo #doesnt get checked

    if options.aml < 20: #attC minimum length can't be less than 20
        msg += '\nYour minimum attC length provided using the flag --aml or '+\
                '--attCminLen must be at least 20. You may choose not to '+\
                'use this flag, in which case the default will be used.\n'
    else:
        aml = options.aml

    if options.aMl < aml: #attC max length can't be less than min
        msg += '\nThe maximum attC length may not be shorter than the '+\
                'minimum.\n'
```

```
    else:
        aMl = options.aMl

    if (options.cml > options.cMl) and (options.cMl != 0):
        msg += '\nYour minimum attC length must be less than the max if the '+\
                'max is not set to 0 (no limit)\n'
    else:
        cml = options.cml
        cMl = options.cMl

    #skipping sp and ep because they require information about the sequence

    if (options.direction != 1) and (options.direction != 0) and \
            (options.direction != -1):
        msg += '\nThe provided sequence direction (using the -dir or '+\
                '--direction flag) must be 1 (for forward), -1 (for '+\
                'reverse), or 0 (for both). You can choose not to use '+\
                'this flag, in which case the default 0 will be used.\n'
    else:
        drn = options.direction

    if msg: #If anything failed, raise this exception
        raise ValueError(msg)

    ##############################################################################
    #Check if running in attC mode or regular mode
    ##############################################################################
    if not attCo: #not running in attC mode

        ##########################################################################
        #Open the fasta file
        ##########################################################################
        print 'Opening fasta file...'
        rec = apf.open_fasta(fasta,aml)

        ##########################################################################
        #Check the input values that couldn't be checked without the file
```

```
130         ##########################################################################
131         print 'Checking fasta-related inputs...'
132         sp = options.sp
133         ep = options.ep
134
135         msg = ''
136         if not ep:
137             ep = len(rec)
138             end_name = 'the sequence length'
139         elif ep > len(rec):
140             ep = len(rec)
141             end_name = 'the sequence length'
142             print 'The specified end point is greater than the sequence lengt'+\
143                     'h. Using the end of the sequence.'
144
145         else:
146             end_name = 'your end position'
147
148         if sp > ep - aml: #must start at least 1 attC length before end
149             msg += 'The start position provided using the -sp or '+\
150             '--start-position flag must be at least one attC length less '+\
151             'than %s.\n' % end_name
152         if msg:
153             raise ValueError(msg)
154
155         #adjust the indexing of sp to 0-indexed now that it's been checked
156         sp = sp-1
157         if sp < 0:
158             sp = 0
159
160         ##########################################################################
161         #generate a list of the sequences to use based on which direction we're
162         #checking
163         ##########################################################################
164         print 'Getting all requested directions...'
165         seqs = apf.get_seqs(drn,rec)
166
```

```
        ##########################################################################
        #Find the features and write the files
        ##########################################################################
        print 'Executing...'
        apf.search_all_directions(seqs,nattI,attIfnm,attIseq,co1,co2,aml,aMl,\
                                  rbd,bds,cml,cMl,sp,ep,outfnm,inner)


    ##############################################################################
    #If running in attC mode
    ##############################################################################
    else:
        #Score all the attCs in the fasta file
        score_lst = apf.score_all_attCs(fasta,rbd,bds)

        #Write the output file
        apf.write_attC_files(score_lst,outfnm)

##############################################################################
#End
##############################################################################
```

A.2 ISAAC Functions

ISAAC_funcs.py contains all the function definitions used in ISAAC.

```
##############################import statements##############################
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO
from optparse import OptionParser

##############################################################################
##############################Function Definitions##############################
##############################################################################

```

```
11 ##################################Program Admin##################################
12
13 ##############################################################################
14 #Option Parsing
15 ##############################################################################
16 def do_args(parser):
17     '''A function that parses the command-line arguments supplied by the user
18     and returns the options object and the leftover args list in a tuple.'''
19
20
21     '''Can't be unit-tested.'''
22
23     #define all the options
24     ##file options
25     parser.add_option('--in','--in-file',type='string',help='A required '+\
26                         'option. Takes the name of the fasta file containing'+\
27                         ' the sequence to annotate. Only the first sequence '+\
28                         'in the file will be used.',dest='in_file')
29     parser.add_option('--out','--out-file',type='string',help='A required '+\
30                         'option. Takes the name of the file to which the '+\
31                         'output annotations should be written. Silently over-'+\
32                         'writes the provided file if it exists, so be careful!'\
33                         ,dest='out')
34     parser.add_option('--frmt','--format',type='string',default='csv',help=\
35                         'Specifies the desired output file type. Currently '+\
36                         'only csv is supported, so this option is not used, '+\
37                         'but we hope to eventually implement genbank.',\
38                         dest='format')
39     parser.add_option('--ci','--cass-inner',action='store_true',default=False,\
40                         help='Use this flag if you want the sequences in the '+\
41                         'cassette fastas to exclude the attC sites and only '+\
42                         'include the non-attC parts of the cassettes. Off by '+\
43                         'default.')
44
45     ##attI options
46     parser.add_option("--attISeq",help='Allows the user to provide their own '+\
47                         'attI sequence if they have one that is not one of '+\
```

```
                        'the ones provided. The attI sequence must end in GTT'+\
                        '. Defaults to None.')
    parser.add_option('--nattI','--no-attI',action='store_true',default=False,\
                        help='Use this flag to instruct the program not to '+\
                        'look for an attI site, but just find attCs. This is '+\
                        'best combined with start position (--sp) and '+\
                        'direction (--dir) arguments (if you know the values '+\
                        'for them) so that the computer doesn\'t waste its '+\
                        'time.',dest='nattI')

    ##attC options
    parser.add_option('--co1','--score1-cutoff',type='int',help='The minimum '+\
                        'allowable score for the structure-based scoring of '+\
                        'attC. Must be an integer between 0 and 59. Defaults'+\
                        ' to 44.',default = 44, dest='co1')
    parser.add_option('--co2','--score2-cutoff',type='int',help='The minimum '+\
                        'allowable score for the sequence-matching scoring of'+\
                        'attC. Must be an integer between 0 and 50. Defaults '+\
                        'to 17.',dest = 'co2',default=17)
    parser.add_option('--attC','--attC-only',action='store_true',default=False,\
                        dest='attCo',help='Use this flag when your input '+\
                        'fasta file only contains attC sequences (one per '+\
                        'entry in the fasta file) and you want each one to be'+\
                        'annotated individually. Defaults to False.')

    ##length options
    parser.add_option('--aml','--attCminLen',type='int',default=50,help=\
                        'The minimum allowable length of attC. Must be at '+\
                        'least 20. Defaults to 50.')
    parser.add_option('--aMl','--attCmaxLen',type='int',default=150,help=\
                        'The maximum allowable length of attC. Defaults to '+\
                        '150.')
    parser.add_option('--cml','--cassminLen',type='int',default=0,help=\
                        'The minimum allowable length of a cassette, '+\
                        'excluding the attCs. Defaults'+\
                        ' to 0.')
    parser.add_option('--cMl','--cassmaxLen',type='int',default=4000,help=\
```

```
                        'The maximum allowable length of a cassette, '+\
                        'excluding the attCs. 0 means no maximum. Defaults'+\
                        ' to 4000.')

    ##searching options
    parser.add_option('--sp','--start-position',type='int',default=1,help=\
                        'The position at which you want the program to start '+\
                        'looking for attCs. Defaults to 1 (the first base in '+\
                        'the sequence). Interacts with dir such that if dir '+\
                        'is set to -1, the value 1 will refer to the first '+\
                        'base in the reverse complement of the provided '+\
                        'sequence (the last base of the provided sequence).'+\
                        ' It is best to leave this at 1 if --dir is going to '+\
                        'be 0. If a value less than 1 is given, 1 will be '+\
                        'used.')
    parser.add_option('--ep','--end-position',type='int',default=None,help=\
                        'The position at which you want to program to stop '+\
                        'looking for attCs. If not provided, or if the value '+\
                        'provided is greater than the length of the sequence '+\
                        'then the program will search to the end '+\
                        'of the sequence. Interacts with --dir like --sp does.')
    parser.add_option('--dir','--direction',type='int',default=1,help=\
                        'The direction in which the program will search.'+\
                        ' By default the '+\
                        'program searches in the given direction only.'+\
                        'If you have no idea which direction you want to '+\
                        'search in then set this to 0 and both directions '
                        'will be searched. If you set this at 0 it is best '+\
                        'not to give values for --sp and --ep. --dir May '+\
                        'only be given the value 1 (forward, default), -1 '+\
                        '(reverse), or 0 (both).',dest='direction')

    ##behaviour options
#   parser.add_option('--q','--quiet',action=store_true,default=False,help=\
#                        'Use of this option is required for batch use of '+\
#                        'this software such as running from a Make rule. It'+\
#                        ' suppresses all prompts to the user.')
```

```
    (options,args) = parser.parse_args()

    return (options,args)

##############################################################################
#File handling
##############################################################################
def open_fasta(fasta,aml):
    '''A function that takes the name of a fasta file, fasta, and returns a
    sequence record object of the first sequence in that fasta file. Raises an
    exception if the sequence is shorter than the minimum attC length.'''

    ''' Final Writing and unit tested. '''

    try:
        rec = SeqIO.read(fasta,'fasta')

    except ValueError,e:
        #if there is more than one record in the file use the first and warn
        if e.args[0] == 'More than one record found in handle':
            recs = SeqIO.parse(fasta,'fasta')
            rec = recs.next()
            print 'The file %s ' % fasta +'contains more than one record. '+\
                    'Using only the first.'

            #other exceptions get raised directly
        else:
            raise e
    if len(rec) < aml:
        ex = 'The sequence must be at least as long as the minimum attC length.'
        raise ValueError(ex)

    return rec

def get_seqs(drn,rec):
    '''A function that takes a direction variable, drn, and a sequence record
```

```
    object, rec, and returns a list of sequences taken from rec in the
    directions specified by drn (1 for forward, -1 for backward, 0 for
    both.)'''


    ''' Final Writing and unit tested. '''

    #Create the output list:
    seqs = []

    if drn == 0:
        seqs.append(str(rec.seq))
        seqs.append(str(rec.seq.reverse_complement()))
    elif drn == 1:
        seqs.append(str(rec.seq))
    elif drn == -1:
        seqs.append(str(rec.seq.reverse_complement()))

    return seqs

def list_attIs(attIfnm,attIseq=''):
    '''A function that takes the file name where the attIs are stored, attIfnm,
    and the attI sequence provided by the user, attISeq, and returns a list of
    properly-formatted and checked attIs that can be used for searching.'''

    ''' Final Writing and unit tested. '''

    #fetch default attIs
    '''Default attIs will be in a text file in the directory with the
    rest of the program. To fetch them we open the file and put all the
    lines into a list. The user can add more attIs to this directory
    directly if they want.'''

    attIf = open(attIfnm)
    attIs = attIf.readlines()
    attIf.close()

```

```
    #attI sequence provided?
    if attIseq:
        #if yes, add to list
        attIs.append(attIseq)

    #make sure everything is uppercase and stripped
    for i in range(len(attIs)):
        attIs[i] = attIs[i].strip()
        attIs[i] = attIs[i].upper()

    #check the sequences for GTT
    msg = ''
    for i in attIs:
        if i[-3:] != 'GTT':
            msg += 'The attI sequence %s doesn\'t end in GTT. Check the ' % i\
            +'default attI list in %s.' % attIfnm

    if msg:
        raise ValueError(msg)

    return attIs

##############################Program Function##############################

#################################################################################
#Initial Search
#################################################################################
def find_all_attIs(seqstr,attIfnm,aml,attIseq=''):
    '''A function that takes a sequence string (seqstr), a name of a file of
    attIs (attIfnm), and an optional string of an attI sequence (attIseq) and
    returns a list of starting dictionaries for integrons within that
    sequence.'''


    '''Written and unit tested.'''

    attIs = list_attIs(attIfnm,attIseq) #get list of attIs
```

```
    this_dir = [] #to store all integrons in this direction

    '''Now that I have a list of all the attIs I'm using, iterate
    through all the attIs in the list and store the position of each one
    in a list of start points.'''

    for attI in attIs: #check each attI sequence
        ##################################################################
        #iterate through attIs and find each instance of each one
        ##################################################################
        combo = [] #create lst of output data for this seq/attI combo

        #prep for the while loop below
        start = 0
        attI_pos = 0
        while (start != -1) and (start < len(seqstr)-len(attI)-aml):

            #search for the next instance of the attI
            this_int = find_attI(start,attI,seqstr)
            if this_int:
                combo.append(this_int) #store this instance of this attI
                start = this_int['start']
            else:
                start = -1

        this_dir += combo #store all instances of a given attI

    return this_dir

def find_first_attC(seq,mco1,co2,aml,aMl,rbd,bds,final=0):
    '''A function that takes a sequence string (seq), a score1 cutoff modified
    because we're not looking for a full match (mco1), a score2 cutoff (co2),
    the minimum and maximum attC lengths (aml and aMl) and base frequency
    dictionaries (rbd,bds) and searches for a single, intact attC. Returns a
    list containing a single starting dictionary with the first attC.'''


```

```
    '''Final Writing and unit tested.'''

    #Look for the first attC
    first_attC = attC_search(seq,0,len(seq),mco1,co2,aml,aMl,rbd,bds,\
                                final=final)

    #if there was no attC, return 0
    if not first_attC:
        return 0

    #Create the "first_attC" dictionary
    start_d = {}
    for k in first_attC:
        key = 'First'+k
        start_d[key] = first_attC[k]

    #add the data needed to find the rest of the attCs if not final
    if not final:
        start_d['start'] = start_d['Firstend']
        start_d['R1'] = start_d['FirstattC'][-4:]

    return start_d


def find_attI(start,attI,seqstr):
    '''A function that takes a start position, start, an attI sequence, attI,
    and a sequence string, seqstr. Searches the seqstr for the first occurence
    of the given attI and returns a dictionary with the keys attIStart,
    attIEnd, attISeq, start, and R1. The value of attISeq will include the
    entire R' it is fused with and R1 will contain the last four bases of that
    R' sequence.'''


    ''' Final Writing and unit tested'''

    #look for the attI
    attI_pos = seqstr.find(attI,start)
```

```
    #if found
    if attI_pos != -1:
        start = attI_pos + len(attI) + 4

        #create the dictionary for this integron
        this_int = {}
        this_int['attISeq'] = seqstr[attI_pos:start] #the whole attI incl R'
        this_int['attIStart'] = attI_pos
        this_int['attIEnd'] = start
        this_int['R1'] = this_int['attISeq'][-4:]
        this_int['start'] = start #the place to start looking for the attC

    #if not found
    else:
        start = -1
        this_int = None

    return this_int


##############################################################################
#attC Searching
##############################################################################
def find_R(seq,se,beg=0,end=0):
    '''A function that takes a sequence string, seq, an integer representing
    the index to start searching at, beg, a string, se, that must be either
    "start" or "end", and an optional argument, end, that gives the index at
    which to stop looking, and returns a tuple whose first element is the
    starting index of the first R site it encounters in the sequence and whose
    second element is that site's sequence. If se is "start" it searches for
    'AAC' and gives an index four bases before that. If se is "end" it searches
    for 'GTT' and gives the index of the G.  Raises an exception if se isn't
    "start", "s", "end", or "e". Returns -1 if no R site was found.'''

    ''' Final Writing and unit tested'''

```

```
    #Check inputs
    msg = ''
    if (se.lower() != "s") and (se.lower() != "start") and (se.lower() != "e") \
        and (se.lower() != "end"):
        msg += '\nThe second argument of find_R() must be one of \'start\', '+\
                '\'s\', \'end\', or \'e\'.\n'

    if seq == '':
        msg += '\nThe sequence may not be null.\n'

    if beg >= len(seq):
        msg += '\nThe beginning search position must be less than the length '+\
                    'of the sequence.\n'

    if msg:
        raise ValueError(msg)

    if end:
        if end <= beg+7:
            return (-1,None)
        if end > len(seq):
            end = len(seq)

    #Start or end?
    if (se.lower() == 's') or (se.lower() == 'start'):
        r = 'AAC'
        #the value to adjust the positioning of the R site
        ap = 4

    elif (se.lower() == 'e') or (se.lower() == 'end'):
        r = 'GTT'
        ap = 0

    #check if we need to include the stop value
    if not end:
        pos = seq.find(r,beg)
    else:
```

```
        pos = seq.find(r,beg,end)

    #is there room for a whole R site in the searched area?
    if pos - ap >= 0:
        pos = pos - ap
        if end:
            if pos + 7 > end:
                #there isn't room for the whole R site
                return (-1,None)
        else:
            if pos + 7 > len(seq):
                #there isn't room for the whole R site
                return (-1,None)
    else:
        #there isn't room for the whole R site
        return (-1,None)

    return (pos,seq[pos:pos+7])

def check_scores(score_dict,co1,co2,bds,attC):
    '''A function that takes a score dictionary, score_dict, an attC sequence,
    attC, the dictionary of base frequency dictionaries for score2 (bds) and
    the two score cutoffs, co1 and co2. Returns the score dictionary with
    score2 added to it if both scores meet the cutorr. Returns a dictionary
    with no score2 value otherwise.'''


    '''Final Writing and not unit tested (score2)'''

    #check first score
    score1 = score_dict['score']
    if score1 >= co1:
        #check second score
        s2 = score2(score_dict,attC,bds)
        if s2 >= co2:
            #add it to the dictionary and return the dictionary
            score_dict['score2'] = s2
```

```
            return score_dict
        else:
            return score_dict
    else:
        return score_dict

def gtt_search(seqstr,aac,min_end,max_end,co1,co2,rbd,bds,R4='',final=0):
    '''A function that takes a sequence string, seqstr, the starting position
    of the current potential attC (aac), a gtt position for minimum length
    (min_end), a gtt position for maximum length (max_end), a score 1 cutoff
    (co1), a score 2 cutoff (co2), base frequency dictionaries (rbd,bds), an R'
    sequence to match to that has the empty string as a default value if there
    is no R' to match to, and a switch, final, that takes the value 0 if we are
    not searching for a final cassette (default) and 1 if we are. Searches for
    an acceptable attC in the sequence.  Returns the score data for the first
    acceptable attC it finds, unless final is 1, in which case it returns all
    potentially-acceptable attCs. If it doesn't find an acceptable attC,
    returns 0.'''


    ''' Final Writing but not unit tested (score2)'''

    #Make list of good enough attCs
    ge = []

    #Set gtt to start
    gtt = 0
    while (gtt != -1) and (min_end < max_end):
        '''This loop searches through gtts until either the score cutoff is met
        or the maximum attC length is reached.'''

        if not final:
            #find the next gtt
            gtt,R2 = find_R(seqstr,'e',min_end,max_end)
        elif final:
            #find the next g
            gtt = seqstr.find('G',min_end,max_end)
```

```
455
456         if gtt != -1:
457             #Score the attC (fn not done)
458             if not final:
459                 pot_attC = seqstr[aac:gtt+7]
460             else:
461                 pot_attC = seqstr[aac:gtt+1]
462             score_dict = score_whole_attC(pot_attC,rbd,final=final,R4=R4)
463             score_dict['dir'] = 1
464
465             #Check the scores against the cutoffs
466             score_dict = check_scores(score_dict,co1,co2,bds,pot_attC)
467             if 'score2' in score_dict: #this indicates it passed
468                 #add the start and end positions
469                 score_dict['start'] = aac
470                 score_dict['attC'] = pot_attC
471                 if not final:
472                     score_dict['end'] = gtt+7
473                 else:
474                     score_dict['end'] = gtt+1
475                 ge.append(score_dict)
476
477             else: #it failed
478                 #check this attC's reverse complement
479                 rev_attC = str(Seq(pot_attC).reverse_complement())
480                 score_dict = score_whole_attC(rev_attC,rbd,final=final,R4=R4,\
481                                               rev=1)
482                 score_dict['dir'] = -1
483
484                 #Check the scores against the cutoffs
485                 score_dict = check_scores(score_dict,co1,co2,bds,rev_attC)
486                 if 'score2' in score_dict: #indicates it passed
487                     #add start and end positions
488                     score_dict['start'] = aac
489                     score_dict['attC'] = rev_attC
490                     if not final:
491                         score_dict['end'] = gtt+7
```

```
                    else:
                        score_dict['end'] = gtt+1
                    ge.append(score_dict)

            #look for the next gtt
            if not final:
                min_end = gtt + 3
            else:
                min_end = gtt+1

    #Once out of the while loop, check if any attCs were found
    if len(ge) == 0: #no attCs found
        return 0

    else: #attCs found
        #get the best one
        if len(ge) == 1:
            best_attC = ge[0]
        else:
            best_attC = get_best(ge)

        return best_attC

def get_best(ge):
    '''A function that takes a list of score dicts, ge, and returns the dict
    with the highest score. If there is more than one then the first is
    taken.'''


    '''under construction and not unit tested'''


    #Get the best scores by score 1
    best = 0
    best_ds = []
    for d in ge:
        if d['score'] == best:
```

```
            best_ds.append(d)
        elif d['score'] > best:
            best_ds = [d]
            best = d['score']

    #If there is more than one, sort by score 2
    if len(best_ds) > 1:
        best = 0
        bestest = []
        for d in best_ds:
            if d['score2'] == best:
                bestest.append(d)
            elif d['score'] > best:
                bestest = [d]
                best = d['score2']
        #If there is still more than one, just take the first
        best_attC = bestest[0]

    else:
        best_attC = best_ds[0]

    return best_attC




def old_gtt_search(seqstr,aac,min_end,max_end,co1,co2,rbd,lbd1,lbd2,R4='',\
                   final=0):
    '''A function that takes a sequence string, seqstr, the starting position
    of the current potential attC (aac), a gtt position for minimum length
    (min_end), a gtt position for maximum length (max_end), a score 1 cutoff
    (co1), a score 2 cutoff (co2), base frequency dictionaries for R, L'' and
    L' (rbd, lbd1, and lbd2 respectively), an R' sequence to match to that has
    the empty string as a default value if there is no R' to match to, and a
    switch, final, that takes the value 0 if we are not searching for a final
    cassette (default) and 1 if we are. Searches for an acceptable attC in the
    sequence.  Returns the score data for the first acceptable attC it finds,
```

```
    unless final is 1, in which case it returns all potentially-acceptable
    attCs. If it doesn't find an acceptable attC, returns 0.'''


    ''' Final Writing and unit tested '''

    gtt = 0
    while gtt != -1:
        '''This loop searches through gtts until either the score cutoff is met
        or the maximum attC length is reached.'''

        if min_end > max_end:
            #we have exceeded the maximum allowable attC len
            #without finding a good attC
            return 0

        if not final:
            #find the next gtt
            gtt,R2 = find_R(seqstr,'e',min_end,max_end)
        elif final:
            #find the next g
            gtt = seqstr.find('G',min_end,max_end)

        if gtt != -1:
            #Score the attC (fn not done)
            if not final:
                pot_attC = seqstr[aac:gtt+7]
            else:
                pot_attC = seqstr[aac:gtt+1]
            score_dict = score_whole_attC(pot_attC,rbd,final=final,R4=R4)
            score_dict['dir'] = 1

            #Check the scores against the cutoffs
            score_dict = check_scores(score_dict,co1,co2,bds,pot_attC)
            if 'score2' in score_dict: #this indicates it passed
                #add the start and end positions
                score_dict['start'] = aac
```

```
                score_dict['attC'] = pot_attC
                if not final:
                    score_dict['end'] = gtt+7
                else:
                    score_dict['end'] = gtt+1
                return score_dict

            else: #it failed
                #check this attC's reverse complement
                rev_attC = str(Seq(pot_attC).reverse_complement())
                score_dict = score_whole_attC(rev_attC,rbd,final=final,R4=R4,\
                                              rev=1)
                score_dict['dir'] = -1

                #Check the scores against the cutoffs
                score_dict = check_scores(score_dict,co1,co2,bds,rev_attC)
                if 'score2' in score_dict: #indicates it passed
                    #add start and end positions
                    score_dict['start'] = aac
                    score_dict['attC'] = rev_attC
                    if not final:
                        score_dict['end'] = gtt+7
                    else:
                        score_dict['end'] = gtt+1
                    return score_dict

                else: #the revcomp also failed
                    #look for the next gtt
                    if not final:
                        min_end = gtt + 3
                    else:
                        min_end = gtt+1

    '''If the function gets out of the while loop, it has reached the end of the
    possible attC lengths without finding an acceptable attC.'''

    return 0
```

```
def attC_search(seqstr,min_start,max_start,co1,co2,aml,aMl,rbd,bds,R4='',\
                final=0):
    '''A function that takes a sequence string, seqstr, an aac position for if
    the cassette has minimum length, min_start, an aac position for if the
    cassette has maximum length, max_start, cutoff values for scores 1 and 2
    (co1 and co2), minimum and maximum attC lengths (aml and aMl,
    respectively), base frequency dictionaries for R, L'', and L' (rbd,bds), a
    four-letter string that is the R' to match to (R1), and a switch, final.
    final will be 1 if we are searching for a final cassette and 0 otherwise.
    R1 will be an empty string if we are searching for the first cassette.
    Searches the entire range of possible cassette lengths for an acceptable
    attC. Returns the first good attC it finds. If no attC can be found within
    the allowable lengths, returns 0.'''


    ''' Final Writing not unit tested (score2)'''

    aac = 0

    while aac != -1:
        '''This loop searches through aacs until either the score cutoff is met
        or the maximum cassette length is reached.'''

        if max_start > len(seqstr):
            max_start = len(seqstr) #dont' go beyond the end of the seq

        if min_start > max_start:
            #we have exceeded the maximum allowable cass len
            #without finding a good attC
            return 0

        #look for the next aac within the allowable cass lens
        aac,R1 = find_R(seqstr,'s',min_start,max_start)

        if aac != -1: #if aac was found
            #the place a gtt will be if the attC is min len
```

```
            min_end = aac + aml - 7
            #the place a gtt will be if the attC is max len
            max_end = aac + aMl - 7
            if max_end > len(seqstr):
                max_end = len(seqstr) #don't go beyond the end of the seq

            this_aac = \
            gtt_search(seqstr,aac,min_end,max_end,co1,co2,rbd,bds,R4=R4,final=final)

            if this_aac:
                '''An attC has been found for this cassette. Return it.'''

                return this_aac
            else:
                min_start = aac+7

        else:
            '''A suitable attC has not yet been found. Increment min_start and
            find the next AAC that could mark the end of this cassette. (i.e.
            continue with this loop).'''

            min_start = aac+7

    '''If the function gets out of the while loop, it has reached the end of the
    possible cassette lengths without finding an attC.'''

    return 0

def add_attC(attCs,this_attC):
    '''A function that takes a dictionary of found attCs (attCs) and a current
    attC dict (this_attC) and adds the current dict to the dict of found attCs.
    Returns the modified dict of found attCs, but also modifies it in place.'''


    '''Written and unit tested'''

    k = attCs.keys()
```

```
    if k == []:
        attCs[0] = this_attC
    else:
        num = max(k) + 1
        attCs[num] = this_attC

    return attCs

def all_intact_attCs(intd,cml,cMl,seqstr,co1,co2,aml,aMl,rbd,bds):
    '''A function that takes a dictionary corresponding to a single integron,
    intd, the minimum and maximum cassette lengths (cml and cMl), the sequence
    string, seqstr, the score 1 and 2 cutoffs (co1 and co2), the minimum and
    maximum attC lengths (aml and aMl), and the base frequency dictionaries
    (rbd,bds) and finds all the intact genomic attCs associated with that
    integron and adds them to the dictionary of attCs.  Returns a tuple whose
    first element is the modified intd and whose second and third elements are
    updated values for min_start and max_start that can be used to find the
    final attC. Also changes it in place.'''


    '''Written and not unit tested (score2)'''

    attCs = intd['attCs']

    #Get the first R' sequence
    R4 = intd['R1']
    start = intd['start']

    #Check bit, will be set to 0 when the last intact attC is found:
    this_attC = True
    while this_attC:

        #the place aac will be if the cass is min len
        min_start = start + cml + 4
        #the place aac will be if the cass is max len
        max_start = start + cMl + 4

```

```
751         #Search for an attC between min_start and max_start
752         this_attC = attC_search(seqstr,min_start,max_start,co1,co2,aml,aMl,\
753                                 rbd,bds,R4=R4)
754         '''If no attC is found, the above will return 0 and we will
755         exit the while loop.'''
756
757         if this_attC: #if an attC is found
758             #increment start and update R4
759             start = this_attC['end']
760             R4 = this_attC['attC'][-4:]
761
762             #add the current attC to the attCs
763             attCs = add_attC(attCs,this_attC)
764
765     '''The above loop will end when the last attC has been found. Once
766     it is found we must search for the final attC.'''
767
768     return (intd,min_start,max_start)
769
770 def find_starts(nattI,seqstr,attIfnm,attIseq,co1,co2,aml,aMl,rbd,bds):
771     '''A function that takes a binary switch, nattI, that is 0 if we are using
772     attI and 1 if we are not, a sequence string (seqstr), the name of the file
773     where the default attIs are stored (attIfnm), a user-provided sequence for
774     a non-default attI sequence (attIseq), score 1 and 2 cutoffs (co1,co2), min
775     and max attC lengths (aml,aMl), frequency dictionaries for R, (rbd,bds) and
776     returns a tuple whose first element is a switch, done, that is False if
777     there are potentially more attCs to find in this direction and True if
778     there are not, and whose second element is a list containing all the
779     integron dictionaries for this direction.'''
780
781
782     '''Written and unit tested.'''
783
784     #create a check bit for identifying when we're done searching for attCs
785     done = False
786
787     #Check if we're using attI
```

```
    if not nattI:
        ##################################################################
        #yes using attI
        ##################################################################
        print 'Searching for attI(s)...'

        '''Find all the attIs associated with this direction & store them
        in integron dictionaries in a list.'''

        this_dir = find_all_attIs(seqstr,attIfnm,aml,attIseq=attIseq)
        if this_dir == []: #if there were no attIs
            done = True

    else:
        ##################################################################
        #not using attI
        ##################################################################
        '''Since we're not using attI we find the first attC after the start
        point and use it as our only start in this sequence.'''

        print 'Searching for first attC...'

        mco1 = co1 -4 +2
        #create list to hold the int dicts
        this_dir = []

        #Get an integron dictionary of the first attC
        first_attC = find_first_attC(seqstr,mco1,co2,aml,aMl,rbd,bds)

        if first_attC:
            #put in in the list for this direction
            this_dir.append(first_attC)

        else:
            #no first attC was found. Search for an incomplete
            #co1 mod'ed for 1st cass & incomplete: co1-(4-mean)-mean
            mco1 = co1 - 4
```

```
825             first_attC = find_first_attC(seqstr,mco1,co2,aml,aMl,rbd,bds,\
826                                          final=1)
827
828             if first_attC:
829                 #Only 1 attC in this direction.
830                 #Put it in the list
831                 this_dir.append(first_attC)
832
833                 #Move on to file writing
834                 done = True
835             else:
836                 #No attCs in this direction
837                 #Move on to file writing
838                 done = True
839     return (done,this_dir)
840
841 def find_all_attCs(this_dir,cml,cMl,seqstr,co1,co2,aml,aMl,rbd,bds):
842     '''A function that takes the list of integron dicts in this direction
843     (this_dir), min and max cassette lengths (cml,cMl), the sequence string
844     (seqstr), score 1 and 2 cutoffs (co1,co2), min and max attC lengths
845     (aml,aMl), base frequency dictionaries (rbd,bds), and iterates through
846     this_dir to find all the attCs associated with each integron in a given
847     direction. Returns the this_dir list, which has been modified so that all
848     the integron dictionaries in it now contain all the attCs associated with
849     that integron.  Also modifies this_dir in place.'''
850
851
852     '''Written and not unit tested (score2)'''
853
854     for i in range(len(this_dir)):
855         print 'Searching for attCs...'
856         intd = this_dir[i]
857
858         #Create a dictionary to store the attCs in
859         intd['attCs'] = {}
860         attCs = intd['attCs']
861
```

```
        #Find all the intact attCs associated with this integron and get start
        #values for finding the final attC.
        intd,min_start,max_start = all_intact_attCs(intd,cml,cMl,seqstr,\
                                                    co1,co2,aml,aMl\
                                                    ,rbd,bds)
    return this_dir

def all_but_write(seq,nattI,attIfnm,attIseq,co1,co2,aml,aMl,rbd,bds,\
                  cml,cMl,sp,ep):
    '''A function that takes a DNA sequence to search (seq), a boolean that will
    be 1 if there is no attI expected, and 0 if attI is expected (nattI), the
    name of the file where allowable attI sequences are stored (attIfnm), a
    user-provided attI sequence (which may be empty) (attIseq), the cutoff
    values for scores 1 and 2 (co1, co2), the minimum and maximum allowable attC
    lengths (aml, aMl), the base frequency dictionary for R (rbd), the base
    frequency dictionary for all the other boxes (bds), the minimum and maximum
    allowable cassette lengths (cml, cMl), and the start and end points for
    searching the sequence (sp, ep). Returns a list of integron annotation
    dictionaries to be written to a file.'''

    if ep == 0:
            ep = len(seq)
    seqstr = seq.upper()
    seqstr = seqstr[sp:ep]

    #Find all the starting integrons in this dir
    done,this_dir = find_starts(nattI,seqstr,attIfnm,attIseq,co1,co2,\
                                aml,aMl,rbd,bds)

    '''Staying within a given sequence direction, we now iterate through all
    the potential integrons identified above and find all the attCs.'''

    if not done: #if it looks like there are more attCs
        this_dir = find_all_attCs(this_dir,cml,cMl,seqstr,co1,co2,aml\
                                  ,aMl,rbd,bds)

    return this_dir,seqstr
```

```
899
900
901 def search_all_directions(seqs,nattI,attIfnm,attIseq,co1,co2,aml,aMl,rbd,bds,\
902                                 cml,cMl,sp,ep,outfnm,inner=0):
903     '''A function that takes a list of sequences (seqs), the nattI switch
904     (nattI), the name of the default attI file (attIfnm), a user-provided attI
905     sequence (attIseq), score 1 and 2 cutoffs (co1,co2), max and min attC
906     lengths (aml, aMl), base freq dicts (rbd,bds), min and max cassette lengths
907     (cml, cMl), start and end points for the sequence to be searched (sp, ep),
908     and the output file name (outfnm). Iterates through all the sequences in
909     seqs and finds all the integrons and attCs associated with them, then writes
910     a csv file for each one. Returns None.'''
911
912
913     '''Written and unit tested.'''
914
915     ##########################################################################
916     #Iterate through the sequences
917     ##########################################################################
918     for seqnum in range(len(seqs)): #iterate through the sequence directions
919         print 'Analyzing direction %i...' % (seqnum+1,)
920
921         #get a plain, upper case string of the sequence
922         seq = seqs[seqnum]
923
924         #Find all attachment sites
925         this_dir,seqstr = all_but_write(seq,nattI,attIfnm,attIseq,co1,co2,aml,\
926                                         aMl,rbd,bds,cml,cMl,sp,ep)
927
928         #Once searching for attC is over, write the files
929         #Write the files
930         print 'Writing attC csv file for direction %i...' % (seqnum+1,)
931         write_files(this_dir,outfnm,seqnum,sp)
932
933         ##########################################################################
934         #Find the cassettes and write the files
935         ##########################################################################
```

```
        print 'Writing cassette fasta file for direction %i...' % (seqnum+1,)
        get_and_write_all_cass(this_dir,seqstr,sp,outfnm,seqnum,inner)

    return

##############################################################################
#attC Scoring
##############################################################################
def seq_match(s1,s2):
    '''A function that takes two dna sequences (one of which has been revcomped
    if necessary), s1 and s1, and assigns points based on the number of matches
    between the two. Returns the integer number of points.'''


    ''' Written and unit tested.'''

    #check the inputs
    if len(s1) != len(s2):
        msg = 'The two sequences are not the same length.'
        raise ValueError(msg)

    #match
    points = 0
    for i in range(len(s1)):
        if s1[i] == s2[i]:
            points += 1
    return points

def base_pat_score(s,bd):
    '''A function that takes a dna sequence, s, and a base
    frequency dictionary, bd, and calculates a score out of 4 for the s. The
    score is calculated by giving a score out of 1 for each base in the
    following way: the score of a given base at a given position is the
    frequency of that base at that position divided by the frequency of the most
    common base at that position. Returns the score as a float.'''


```

```
    ''' Final Writing and unit tested.'''

    #check inputs
    msg = ''
    for i in s:
        if (i != 'A') and (i != 'T') and (i != 'G') and (i != 'C'):
            msg += '\nThe sequence can only have A, T, G, and C.\n'
            break

    k = bd.keys()
    k.sort()
    if 'tot' not in k:
        m = '\nThe keys of the base frequency dictionary must be the '+\
                'numbers from 0 to n-1 where n is the length of sequence to '+\
                'be tested, plus \'tot\'.\n'
        raise ValueError(m)

    else:
        k.remove('tot')
        if k != range(len(s)):
            msg += '\nThe keys of the base frequency dictionary must be the '+\
                    'numbers from 0 to n-1 where n is the length of sequence to '+\
                    'be tested, plus \'tot\'.\n'

    for i in k:
        j = bd[i].keys()
        j.sort()
        if j != ['A','C','G','T']:
            msg += '\nThe keys of the base frequency subdictionary at '+\
                    'position %i' %i +' are not A, C, G, T.\n'

        freqs = bd[i].values()
        if sum(freqs) != bd['tot']:
            msg += '\nThe base frequencies at position %i' %i +' don\'t sum '+\
                    'to 1.\n'
    if msg:
        raise ValueError(msg)
```

```
    #score
    points = 0

    for i in range(len(s)):
        vals = bd[i].values() #get all frequencies at this position
        max_val = max(vals) #get the maximum values

        points += bd[i][s[i]]/float(max_val)

    return points

def trip_not_same(trip):
    '''A function that takes a triplet string, trip. Returns 0 if the three
    letters in that triplet are all the same, 1 if not.'''


    ''' Unit tested.'''

    if len(trip) != 3:
        msg = 'This triplet is not 3 letters long.'
        raise ValueError(msg)

    if (trip[0]==trip[1]==trip[2]):
        return 0
    else:
        return 1

def score_triplet_pair(t1,t2):
    '''A function that takes two potentially-matching triplets from L sites and
    returns their triplet score.'''


    ''' Unit tested. '''


    #each triplet pair starts with a score of 0
```

```
    score = 0

    #add one point for each match between the pairs
    score += seq_match(t1,t2) #written and tested

    #add one point for each of the two triplets that isn't the same base 3 times
    score += trip_not_same(t1) #written and tested
    score += trip_not_same(t2)

    return score

def score_L(L1,L2):
    '''A function that takes two potential L sites (L1 and L2) and finds their
    match score'''


    ''' Unit tested. '''


    #L1 should have an extrahelical base and therefore be longer than L2 by 1
    if len(L1) != len(L2)+1:
        msg = 'L\'\' should be one bp longer than L\'.'
        raise ValueError(msg)

    #score the triplets
    trip1 = score_triplet_pair(L1[:3],L2[:3]) #written and tested
    trip2 = score_triplet_pair(L1[-3:],L2[-3:])

    score = trip1*trip2

    return score


def score_Ls(att):
    '''A function that takes a dna string from an attC that starts immediately
    after R'' and ends immediately before R' (seq) and checks all 4 possible L
    scores.  Returns a score dictionary for the best-scoring L. The dictionary
```

```
    will have keys L1,L2,L1-pos,L2-pos,L-score,ehb.'''


    ''' written and unit tested. '''

    #A dictionary to store all the scores in
    all_scores = {}

    #Take the complement of the sequence
    revatt = str(Seq(att).reverse_complement())

    for e in range(5,7): #iterates through 5 and 6 for L2
        for s in range(5,7): #iterates through 5 and 6 for L1
            pos = '%i-%i' % (s,e) #the key for all_scores for this combo
            all_scores[pos] = {}

            #add the L sequences to the dictionary
            L1 = att[s:s+7]
            all_scores[pos]['L1'] = L1
            L2 = revatt[e:e+6]
            all_scores[pos]['L2'] = str(Seq(L2).reverse_complement())

            all_scores[pos]['L-score'] = score_L(L1,L2) #written and tested

    #Choose the best score (if there is more than one max, takes at random)
    scores = []
    for k in all_scores:
        scores.append(all_scores[k]['L-score'])
    max_score = max(scores)
    for pos in all_scores:
        if all_scores[pos]['L-score'] == max_score:
            best_score = all_scores[pos]
            best_score['L1-pos'] = int(pos[0])
            best_score['L2-pos'] = int(pos[2])
            break

    #Double the L structure score to increase its weighting
```

```
    best_score['L-score'] = best_score['L-score'] * 2
    L1 = best_score['L1']
    L2 = str(Seq(best_score['L2']).reverse_complement())

    #check the extrahelical base
    if L1[3] != L2[3]:
        best_score['L-score'] += 1
        best_score['ehb'] = 1
    else:
        best_score['ehb'] = 0

    return best_score


def score_whole_attC(attC,rbd,final=0,R4='',rev=0):
    '''A function that takes an attC sequence (attC), a switch (final) that
    will be 0 if this is not a final attC and 1 if it is, a switch (rev) that
    takes 0 (default) if the attC being tested is not reversed from the overall
    sequence, and 1 if it is, a four-character string, R4 that is the R' of the
    previous attI/attC to match with, and a base frequency dictionary for R
    (rbd). If this is the first attC with no previous R' then R4 will be an
    empty string. Scores the attC. Returns a dictionary that has all the
    features of the attC so they can be properly annotated.'''


    ''' Final Writing and unit tested '''

    #Check the R sites
    ##Check R''
    if rev: #this will be the last four bases, and is already revcomped
        r1r = attC[-4:]
    else:
        r1 = attC[:4]
        r1r = str(Seq(r1).reverse_complement())
    if R4:
        #match R'' to the old R'
        r1_points = seq_match(r1r,R4) #written and tested
```

```
    else:
        #use R base pattern matching
        r1_points = base_pat_score(r1r,rbd) #written, not tested

    ##Check R'
    if not final: #there will be an R'
        if rev: #this will be the first four bases and needs to be revcomped
            r2r = attC[:4]
            r2 = str(Seq(r2r).reverse_complement())
        else:
            r2 = attC[-4:]
        r2_points = base_pat_score(r2,rbd)

    else: #there is no R'
        r2_points = 0


    #Check the L sites
    if not final:
        Ls = attC[7:-7]
    else:
        if not rev:
            Ls = attC[7:-1]
        else:
            Ls = attC[1:-7]

    score_d = score_Ls(Ls) #written, not tested
    score_d['score'] = score_d['L-score']+r1_points+r2_points
    score_d['r1_score'] = r1_points
    if final:
        score_d['r2_score'] = 'final'
    else:
        score_d['r2_score'] = r2_points

    return score_d
```

```
def old_score2(score_dict,lbd1,lbd2):
    '''A function that takes an attC score dictionary, score_dict, an L1 base
    frequency dictionary that has the base frequencies for L'' (lbd1) and an L2
    base frequency dictionary that has the base frequencies for L' (lbd2) and
    scores the base frequencies in the L.  Returns a score.'''


    '''Final Writing and unit tested.'''

    score = 0

    #score L1
    L1 = score_dict['L1']
    score += base_pat_score(L1,lbd1)

    #score L2
    L2 = score_dict['L2']
    score += base_pat_score(L2,lbd2)

    return score

def score2(score_dict,attC,bds,final=0):
    '''A function that takes an attC score dictionary, score_dict, the attC
    sequence, attC, and a dictionary of base dicts, bds. Returns a score based
    on nucleotide frequencies at a number of positions.'''


    '''Written and unit tested'''

    score = 0

    #score Ls
    L1 = score_dict['L1']
    score += base_pat_score(L1,bds['L1'])
    L2 = score_dict['L2']
    score += base_pat_score(L2,bds['L2'])

```

```
    #score first gap and first internal
    if score_dict['L1-pos'] == 5:
        G1 = attC[7:12]
        I1 = attC[19:25]
        score += base_pat_score(G1,bds['G1-5'])
        score += base_pat_score(I1,bds['I1'])
    elif score_dict['L1-pos'] == 6:
        G1 = attC[7:13]
        I1 = attC[20:26]
        #normalize this score so it doesn't have an advantage
        g1s = base_pat_score(G1,bds['G1-6'])
        g1s = (g1s/6)*5
        score += g1s
        score += base_pat_score(I1,bds['I1'])
    else:
        msg = '\nL1 must be either 5 or 6 bases from R1'
        raise ValueError(msg)

    #score second gap and internal
    if not final:
        if score_dict['L2-pos'] == 5:
            G2 = attC[-12:-7]
            I2 = attC[-24:-18]
            score += base_pat_score(G2,bds['G2-5'])
            score += base_pat_score(I2,bds['I2'])
        elif score_dict['L2-pos'] == 6:
            G2 = attC[-13:-7]
            I2 = attC[-25:-19]
            #normalize this score so it doesn't have an advantage
            g2s = base_pat_score(G2,bds['G2-6'])
            g2s = (g2s/6)*5
            score += g2s
            score += base_pat_score(I2,bds['I2'])
        else:
            msg = '\nL2 must be either 5 or 6 bases from R2'
            raise ValueError(msg)
    else:
```

```
        if score_dict['L2-pos'] == 5:
            G2 = attC[-6:-1]
            I2 = attC[-18:-12]
            score += base_pat_score(G2,bds['G2-5'])
            score += base_pat_score(I2,bds['I2'])
        elif score_dict['L2-pos'] == 6:
            G2 = attC[-7:-1]
            I2 = attC[-19:-13]
            #normalize this score so it doesn't have an advantage
            g2s = base_pat_score(G2,bds['G2-6'])
            g2s = (g2s/6)*5
            score += g2s
            score += base_pat_score(I2,bds['I2'])
        else:
            msg = '\nL2 must be either 5 or 6 bases from R2'
            raise ValueError(msg)


    #score Rs
    R1 = attC[:4]
    score += base_pat_score(R1,bds['R1'])
    if not final:
        R2 = attC[-4:]
        score += base_pat_score(R2,bds['R2'])

    return score

##############################################################################
#attC Mode
##############################################################################
def check_seq_pats(seq):
    '''A function that takes a sequence string (seq) and checks that it follows
    the attC pattern (i.e. that it starts with NNNNAAC, ends with GTTNNNN, and
    is at least 31 bases long. Returns 1 if the conditions are met, 0 if not.'''

    '''Written and unit tested.'''

```

```
    if seq[-7:-4].upper() != 'GTT':
        return 0
    if seq[4:7].upper() != 'AAC':
        return 0
    if len(seq) < 31:
        return 0

    return 1

def score_all_attCs(in_file,rbd,bds):
    '''A function that takes the name of a fasta file of attCs (in_file) and
    the base dictionaries (rbd,bds) and returns a list of tuples whose first
    element is an annotated attC and whose second element is either an attC
    score dictionary or, if the sequence didn't appear to be an attC, 0.'''


    '''Written and unit tested.'''

    #Open the fasta file
    recs = SeqIO.parse(in_file,'fasta')

    #Create a list to store the results
    score_lst = []

    #Iterate through the records
    for rec in recs:
        seq = str(rec.seq)
        #check the sequence
        ok_seq = check_seq_pats(seq)

        #if the sequence looks like an attC, score it
        if ok_seq:
            score_dict = score_whole_attC(seq,rbd)

            #Add score 2
            score_dict['score2'] = score2(score_dict,seq,bds)

```

```
            #Add the sequence to the dictionary
            score_dict['attC'] = seq

            #Add this dictionary to the list
            score_lst.append((rec.id,score_dict))

        else:
            score_lst.append((rec.id,0))

    return score_lst

##############################################################################
#Writing Output
##############################################################################
def write_attC_files(score_lst,fnm):
    '''A function that takes a list of attC score dictionary tuples and a file
    name and writes a csv annotating each attC.'''


    '''Written and unit tested'''

    #open the file for writing
    fnm = fnm+'_attC_scores.csv'
    outf = open(fnm,'w')

    #Create the header line
    L = 'ID,R\'\',R\'\'-score,5/6,L\'\',ehb,L\''+\
                ',L-score,5/6,R\',R\'-score,final_score,score2\n'
    outf.write(L)
    #Write all the scores
    for stpl in score_lst:
        #id
        L = str(stpl[0])+','
        #attC
        attCd = stpl[1]
        if attCd: #if this passed the attC pattern test
            L += str(attCd['attC'][:4])+','
```

```
            L += str(attCd['r1_score'])+','
            #for 5/6:
            pos = attCd['L1-pos']
            L += str(attCd['attC'][7:7+pos])+','
            L += str(attCd['L1'])+','
            if attCd['ehb']:
                L += str(attCd['L1'][3])+','
            else:
                L += 'X,'
            L += str(attCd['L2'])+','
            L += str(attCd['L-score'])+','
            #for next 5/6
            pos = attCd['L2-pos']
            if attCd['r2_score'] == 'final':
                L += str(attCd['attC'][-1-pos:-1])+','
            else:
                L += str(attCd['attC'][-7-pos:-7])+','
            if attCd['r2_score'] == 'final':
                L+= 'final,'
            else:
                L += str(attCd['attC'][-4:])+','
            L += str(attCd['r2_score'])+','
            L += str(attCd['score'])+','
            L += str(attCd['score2']) +'\n'
        else: #if this didn't pass the pattern test
            L += 'This doesn\'t appear to be an attC.\n'
        outf.write(L)

    outf.close()

    return


def write_files(int_lst,fnm,seqnum,sp):
    '''A function that takes a list of integron dictionaries (int_lst), a file
    name (fnm), a number indicating whether this is the first (0) or second (1)
    direction (seqnum), and the start position of the sequence (sp) and writes
```

```
    the attC annotations to a file.'''


    '''Written and unit tested.'''

    #Create a name for the file based on the sequence number
    fnm = fnm+str(seqnum+1)+'.csv'

    #Open the file for writing
    outf = open(fnm,'w')

    #Iterate through the integrons
    for intnum in range(len(int_lst)):
        int_dict = int_lst[intnum]
        L = 'Integron %i:\n' % (intnum+1,)
        outf.write(L)

        #Check if there's an attI
        if 'attISeq' in int_dict:
            #Write the attI lines
            L = 'attI:\n'
            L += 'attIStart,attIEnd,R\',attISeq\n'
            L += str(int_dict['attIStart']+sp+1)+','
            L += str(int_dict['attIEnd']+sp)+','
            L += str(int_dict['R1'])+','
            L += str(int_dict['attISeq'])+'\n'
            outf.write(L)
            #Create the header for the attCs
            L = 'attCs:\n'
            L += 'StartPos,EndPos,dir,R\'\',R\'\'-score,5/6,L\'\',ehb,L\''+\
                    ',L-score,5/6,R\',R\'-score,final_score,attCSeq\n'
            outf.write(L)

        elif 'FirstattC' in int_dict:
            #Write the attC headers
            L = 'No attI. attCs:\n'
            L += 'StartPos,EndPos,dir,R\'\',R\'\'-score,5/6,L\'\',ehb,L\''+\
```

```
                ',L-score,5/6,R\',R\'-score,final_score,attCSeq\n'
        outf.write(L)
        #Write the line for the first attC
        if int_dict['Firstdir'] == -1:
            int_dict['FirstattC'] = \
                    str(Seq(int_dict['FirstattC']).reverse_complement())
        L = ''
        L += str(int_dict['Firststart']+sp+1)+','
        L += str(int_dict['Firstend']+sp)+','
        L += str(int_dict['Firstdir'])+','
        L += str(int_dict['FirstattC'][:4])+','
        L += str(int_dict['Firstr1_score'])+','
        #for 5/6:
        pos = int_dict['FirstL1-pos']
        L += str(int_dict['FirstattC'][7:7+pos])+','
        L += str(int_dict['FirstL1'])+','
        if int_dict['Firstehb']:
            L += str(int_dict['FirstL1'][3])+','
        else:
            L += 'X,'
        L += str(int_dict['FirstL2'])+','
        L += str(int_dict['FirstL-score'])+','
        #for next 5/6
        pos = int_dict['FirstL2-pos']
        if int_dict['Firstr2_score'] == 'final':
            L += str(int_dict['FirstattC'][-1-pos:-1])+','
        else:
            L += str(int_dict['FirstattC'][-7-pos:-7])+','
        if int_dict['Firstr2_score'] == 'final':
            L += 'final,'
        else:
            L += str(int_dict['FirstattC'][-4:])+','
        L += str(int_dict['Firstr2_score'])+','
        L += str(int_dict['Firstscore'])+','
        L += str(int_dict['FirstattC'])+'\n'
        outf.write(L)

```

```
        #Write the rest of the attCs
        attCs = int_dict['attCs']
        if len(attCs) > 0:
            keys = attCs.keys()
            keys.sort()
            for k in keys:
                d = attCs[k]
                if d['dir'] == -1:
                    d['attC'] = str(Seq(d['attC']).reverse_complement())
                L = ''
                L += str(d['start']+sp+1)+','
                L += str(d['end']+sp)+','
                L += str(d['dir'])+','
                L += str(d['attC'][:4])+','
                L += str(d['r1_score'])+','
                #for 5/6:
                pos = d['L1-pos']
                L += str(d['attC'][7:7+pos])+','
                L += str(d['L1'])+','
                if d['ehb']:
                    L += str(d['L1'][3])+','
                else:
                    L += 'X,'
                L += str(d['L2'])+','
                L += str(d['L-score'])+','
                #for next 5/6
                pos = d['L2-pos']
                if d['r2_score'] == 'final':
                    L += str(d['attC'][-1-pos:-1])+','
                else:
                    L += str(d['attC'][-7-pos:-7])+','
                if d['r2_score'] == 'final':
                    L+= 'final,'
                else:
                    L += str(d['attC'][-4:])+','
                L += str(d['r2_score'])+','
                L += str(d['score'])+','
```

```
                L += str(d['attC'])+'\n'
                outf.write(L)
        else:
            L = 'No (more) attCs.\n'
            outf.write(L)

    if int_lst == []:
        L = 'No integrons found in this direction.\n'
        outf.write(L)

    outf.close()
    return

def write_cass_files(outfnm,seqnum,intnum,cass_lst,sp):
    '''A function that takes the output file name base (outfnm), the sequence
    number (seqnum), the integron number (intnum), a list of cassettes
    (cass_lst), and the start position of the sequence, sp, and writes a fasta
    file for all the cassettes in that integron in order.'''


    '''under construction and not unit tested'''

    #Open the file for writing
    outfnm = outfnm+'_'+str(seqnum+1)+'_integron_'+str(intnum+1)+'_cassettes.fna'
    outf = open(outfnm,'w')

    #Iterate through the cassettes
    for i in range(len(cass_lst)):
        tpl = cass_lst[i]

        #Create the sequence record object
        seq = Seq(tpl[2])
        rec = SeqRecord(seq)
        rec.id = str(i)
        rec.description = str(tpl[0]+sp+1)+' - '+str(tpl[1]+sp)

        #Write to the file
```

```
        L = rec.format('fasta')
        outf.write(L)

    outf.close()
    return

##############################Cassette Functions##############################

def get_whole_cass(attC1,attC2,seqstr,final=0):
    '''A function that takes the end points of two attCs (or one attI and one
    attC) in order (attC1, attC2), a sequence string (seqstr), and a switch that
    is 1 if this is the final cassette and 0 if it is not and returns a tuple
    whose first two elements are the start and end positions of the cassette and
    whose third element is the cassette sequence.'''

    #Get the cassette boundaries
    cass_start = attC1 - 6
    if not final:
        cass_end = attC2 - 6
    else:
        cass_end = attC2

    #Get the sequence
    cass_seq = seqstr[cass_start:cass_end]

    return (cass_start,cass_end,cass_seq)

def get_inner_cass(attC1,attC2,seqstr):
    '''A function that takes the end point of an attI or attC (attC1) and the
    start point of the next attC (attC2), and a sequence string (seqstr).
    Returns a tuple whose first two elements are the start and end points of the
    inner section of the cassette bounded by these attCs (inner section means
    the cassette excluding the attCs) and whose final element is the sequence of
    that inner section.'''

    cass = seqstr[attC1:attC2]

```

```
    return (attC1,attC2,cass)

def get_cass_from_csv(csv):
    '''A function that takes a csv of an annotated integron and the associated
    sequence (seqstr) and extracts all but the last cassette.'''

    #Open the file
    pass

def get_all_cass(intd,seqstr,inner=0):
    '''A function that takes an integron dictionary that has all the annotated
    attI/attCs of a given integron in it (intd), the sequence this was taken from
    (seqstr), and a switch that is 1 if we are looking for inner cassettes and 0
    otherwise (inner) and returns a list of tuples in order where each tuple
    contains the start position, end position, and sequence of a cassette.'''


    '''Written and unit tested'''

    attCs = intd['attCs']

    #Create the list
    cass_lst = []

    #Check if there are any cassettes
    if len(attCs) == 0:
        #there are no cassettes
        return []

    #Get the first cassette
    #Check if there is more than one cassette (i.e. if the first is also last):
    if len(attCs) == 1:
        #there is only one cassette
        #inner or outer?
        if inner:
            #get cassette boundaries
            s = intd['start']
```

```
            e = attCs[0]['start']
            cass = get_inner_cass(s,e,seqstr)
        else:
            s = intd['start']
            e = attCs[0]['end']
            cass = get_whole_cass(s,e,seqstr)
        cass_lst.append(cass)
        return cass_lst
    else:
        #there is more than one cassette
        #Get the first cassette
        #inner or outer?
        if inner:
            #get cassette boundaries
            s = intd['start']
            e = attCs[0]['start']
            cass = get_inner_cass(s,e,seqstr)
        else:
            s = intd['start']
            e = attCs[0]['end']
            cass = get_whole_cass(s,e,seqstr)
        cass_lst.append(cass)
        #iterate through all the attCs
        for i in range(len(attCs)-1):
            #inner or outer
            if inner:
                s = attCs[i]['end']
                e = attCs[i+1]['start']
                cass = get_inner_cass(s,e,seqstr)
            else:
                s = attCs[i]['end']
                e = attCs[i+1]['end']
                cass = get_whole_cass(s,e,seqstr)

            cass_lst.append(cass)

    return cass_lst
```

```
def get_and_write_all_cass(this_dir,seqstr,sp,outfnm,seqnum,inner=0):
    '''A function that takes a list of integrons in this direction (this_dir),
    a string of the search sequence (seqstr), the start position (sp), the base
    name of the output files, outfnm, the sequence number, seqnum, and a
    switch, inner, that is 1 if the cassettes should exclude the attC and 0 if
    not. Writes annotated cassettes to fasta files.'''


    '''Written but not unit tested.'''

    #Iterate through the list of integrons in this direction
    for i in range(len(this_dir)):
        int_dict = this_dir[i]

        #Get a list of all the cassettes
        cass_lst = get_all_cass(int_dict,seqstr,inner)

        #Write the cassettes to a file
        write_cass_files(outfnm,seqnum,i,cass_lst,sp)

    return
```

Appendix B

Base Frequency Tables

B.1 IUPAC Codes

Ambiguity Code	Base(s)
A	A
C	C
G	G
T/U	T
M	A / C
R	A / G
W	A / T
S	C / G
Y	C / T
K	G / T
V	A / C / G
H	A / C / T
D	A / G / T
B	C / G / T
N	A / C / G / T

Table B.1: The IUPAC ambiguity codes, taken from `http://droog.gs.washington.edu/parc/images/iupac.html`

B.2 R Boxes

Element	Position	A	T	G	C	80	90	95
R″	1	9.8	5.9	68.6	15.7	S		
	2	3.9	44.4	25.5	26.1			
	3	13.7	9.2	0.0	77.1	M	M	
	4	0.0	77.8	3.3	19.0	Y	Y	Y
R′	1	81.7	2.0	6.5	9.8	A	M	
	2	15.7	6.5	77.8	0.0	R	R	
	3	58.8	8.5	26.1	6.5	R		
	4	5.9	42.5	7.2	44.4	Y		

Table B.2: The frequency of each nucleotide at each position in the R boxes of attCs associated with IntI1

Element	Position	A	T	G	C	80	90	95
R″	1	57.7	6.9	28.8	6.6	R		
	2	11.6	14.6	16.9	56.9			
	3	51.5	4.4	0.2	44.0	M	M	M
	4	1.4	96.7	1.2	0.8	T	T	T
R′	1	98.2	0.9	0.9	0.0	A	A	A
	2	5.3	50.5	44.3	0.0	K	K	
	3	13.0	13.0	56.0	18.1			
	4	7.5	58.9	5.6	28.0	Y		

Table B.3: The frequency of each nucleotide at each position in the R boxes of attCs associated with IntI4

Element	Position	A	T	G	C	80	90	95
R″	1	36.4	22.7	22.7	18.2			
	2	9.1	27.3	22.7	40.9			
	3	36.4	22.7	4.5	36.4			
	4	13.6	77.3	0.0	9.1	W	W	
R′	1	95.5	4.5	0.0	0.0	A	A	A
	2	9.1	27.3	54.5	9.1	K		
	3	27.3	4.5	45.5	22.7			
	4	22.7	18.2	27.3	31.8			

Table B.4: The frequency of each nucleotide at each position in the R boxes of attCs associated with Son

Element	Position	A	T	G	C	80	90	95
R″	1	44.3	13.4	40.2	2.1	R		
	2	2.1	51.5	6.2	40.2	Y	Y	
	3	69.1	11.3	10.3	9.3	W		
	4	0.0	92.8	0.0	7.2	T	T	Y
R′	1	90.7	2.1	6.2	1.0	A	A	R
	2	11.3	68.0	9.3	11.3			
	3	51.5	1.0	41.2	6.2	R	R	
	4	13.4	41.2	3.1	42.3	Y		

Table B.5: The frequency of each nucleotide at each position in the R boxes of attCs associated with Vfi

Element	Position	A	T	G	C	80	90	95
R″	1	49.3	10.1	36.2	4.3	R		
	2	13.0	21.7	7.2	58.0			
	3	56.5	4.3	0.0	39.1	M	M	M
	4	0.0	100.0	0.0	0.0	T	T	T
R′	1	100.0	0.0	0.0	0.0	A	A	A
	2	4.3	56.5	39.1	0.0	K	K	K
	3	20.3	14.5	58.0	7.2			
	4	10.1	49.3	4.3	36.2	Y		

Table B.6: The frequency of each nucleotide at each position in the R boxes of attCs associated with Vpa

Element	Position	A	T	G	C	80	90	95
R″	1	0.0	0.0	93.0	7.0	G	G	S
	2	0.0	14.0	5.3	80.7	C	Y	
	3	3.5	0.0	3.5	93.0	C	C	V
	4	0.0	100.0	0.0	0.0	T	T	T
R′	1	100.0	0.0	0.0	0.0	A	A	A
	2	0.0	3.5	96.5	0.0	G	G	G
	3	8.8	0.0	87.7	3.5	G	R	R
	4	1.8	5.3	5.3	87.7	C	B	

Table B.7: The frequency of each nucleotide at each position in the R boxes of attCs associated with Xca

Element	Position	A	T	G	C	80	90	95
R″	1	44.8	7.5	40.0	7.7	R		
	2	9.0	23.7	16.0	51.3			
	3	44.0	6.2	1.3	48.5	M	M	
	4	1.3	93.5	1.3	4.0	T	T	Y
R′	1	95.1	1.1	2.1	1.7	A	A	A
	2	7.3	42.1	49.4	1.2	K	K	
	3	25.1	10.1	51.3	13.5			
	4	8.0	49.7	6.1	36.2	Y		

Table B.8: The frequency of each nucleotide at each position in the R boxes of all the attCs identified by ISAAC

B.3 Spacer Boxes

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	77.0	6.1	2.7	14.2	M	M	
	2	58.8	6.8	20.3	14.2			
	3	15.5	57.4	19.6	7.4			
	4	1.4	71.6	20.9	6.1	K	K	
	5	6.1	2.7	12.8	78.4	S	S	
Sp′ (5)	1	3.8	34.0	24.5	37.7			
	2	49.1	13.2	17.0	20.8			
	3	39.6	30.2	30.2	0.0			
	4	32.1	15.1	35.8	17.0			
	5	0.0	3.8	18.9	77.4	S	S	S
Sp′ (6)	1	0.0	92.0	1.0	7.0	T	T	Y
	2	4.0	10.0	3.0	83.0	C	Y	
	3	46.0	13.0	15.0	26.0			
	4	31.0	3.0	55.0	11.0	R		
	5	15.0	1.0	74.0	10.0	R		
	6	1.0	7.0	0.0	92.0	C	C	Y

Table B.9: The frequency of each nucleotide at each position in the Sp boxes of attCs associated with IntI1

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	99.8	0.0	0.2	0.0	A	A	A
	2	98.8	0.5	0.5	0.3	A	A	A
	3	94.6	1.8	2.6	0.9	A	A	R
	4	0.6	0.9	5.1	93.4	C	C	S
	5	3.7	3.5	88.2	4.6	G	S	
Sp′ (5)	1	4.2	3.5	3.8	88.5	C	M	
	2	2.9	0.5	92.9	3.8	G	G	S
	3	0.8	1.8	97.3	0.2	G	G	G
	4	0.0	2.0	97.1	0.9	G	G	G
	5	0.0	2.7	0.0	97.3	C	C	C

Table B.10: The frequency of each nucleotide at each position in the Sp′ and Sp″ boxes of attCs associated with IntI4

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	88.2	0.0	5.9	5.9	A	V	
	2	88.2	5.9	0.0	5.9	A	H	
	3	17.6	23.5	52.9	5.9			
	4	5.9	29.4	11.8	52.9	Y		
	5	29.4	5.9	35.3	29.4			
Sp′ (5)	1	6.3	12.5	43.8	37.5	S		
	2	18.8	6.3	62.5	12.5	R		
	3	18.8	6.3	75.0	0.0	R	R	
	4	12.5	6.3	81.3	0.0	G	R	
	5	6.3	0.0	6.3	87.5	C	V	

Table B.11: The frequency of each nucleotide at each position in the Sp″ and Sp′ boxes of attCs associated with Son

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	100.0	0.0	0.0	0.0	A	A	A
	2	98.9	0.0	0.0	1.1	A	A	A
	3	85.3	12.6	1.1	1.1	A	W	W
	4	1.1	69.5	20.0	9.5	K		
	5	0.0	1.1	80.0	18.9	G	S	S
Sp′ (5)	1	0.0	1.1	18.9	80.0	C	S	S
	2	77.9	4.2	0.0	17.9	M	M	M
	3	9.5	0.0	90.5	0.0	G	G	R
	4	1.1	1.1	97.9	0.0	G	G	G
	5	0.0	1.1	1.1	97.9	C	C	C

Table B.12: The frequency of each nucleotide at each position in the Sp″ and Sp′ boxes of attCs associated with Vfi

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	100.0	0.0	0.0	0.0	A	A	A
	2	100.0	0.0	0.0	0.0	A	A	A
	3	84.1	5.8	10.1	0.0	A	R	
	4	0.0	10.1	43.5	46.4	S		
	5	13.0	36.2	7.2	43.5			
Sp′ (5)	1	34.8	14.5	43.5	7.2			
	2	8.7	0.0	47.8	43.5	S	S	
	3	1.4	0.0	98.6	0.0	G	G	G
	4	0.0	0.0	100.0	0.0	G	G	G
	5	0.0	0.0	0.0	100.0	C	C	C

Table B.13: The frequency of each nucleotide at each position in the Sp″ and Sp′ boxes of attCs associated with Vpa

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	100.0	0.0	0.0	0.0	A	A	A
	2	93.0	0.0	7.0	0.0	A	A	R
	3	8.8	59.6	3.5	28.1	Y		
	4	0.0	100.0	0.0	0.0	T	T	T
	5	0.0	0.0	0.0	100.0	C	C	C
Sp′ (6)	1	0.0	100.0	0.0	0.0	T	T	T
	2	0.0	3.6	0.0	96.4	C	C	C
	3	85.7	7.1	0.0	7.1	A	H	
	4	25.0	0.0	75.0	0.0	R	R	R
	5	7.1	3.6	85.7	3.6	G	R	
	6	0.0	3.6	0.0	96.4	C	C	C

Table B.14: The frequency of each nucleotide at each position in the Sp″ and Sp′ boxes of attCs associated with Xca

Box	Position	A	T	G	C	80	90	95
Sp″ (5)	1	96.6	0.8	0.6	2.0	A	A	A
	2	91.4	1.3	3.4	3.9	A	A	M
	3	73.0	15.0	7.2	4.7	W		
	4	0.7	24.6	10.8	63.8	Y		
	5	4.7	5.2	65.1	24.9	S	S	
Sp′ (5)	1	6.2	6.3	10.5	77.1	S		
	2	15.0	1.7	73.9	9.4	R		
	3	5.5	3.2	90.9	0.3	G	G	R
	4	3.8	2.8	91.8	1.6	G	G	R
	5	0.6	3.7	1.4	94.3	C	C	Y
Sp′ (6)	1	0.0	92.3	2.8	5.0	T	T	Y
	2	2.8	8.8	5.0	83.4	C	Y	
	3	60.2	9.9	11.0	18.8			
	4	30.4	3.9	59.1	6.6	R		
	5	12.2	3.9	75.7	8.3	R		
	6	6.1	7.2	0.6	86.2	C	Y	

Table B.15: The frequency of each nucleotide at each position in the Sp boxes of all the attCs identified by ISAAC

B.4 L Boxes

Box	Position	A	T	G	C	80	90	95
L″	1	11.1	5.2	70.6	13.1	S		
	2	9.8	58.8	13.1	18.3			
	3	5.9	86.9	5.2	2.0	T	W	
	4	3.3	7.2	19.0	70.6	S		
	5	64.7	0.0	11.8	23.5	M		
	6	86.9	3.3	2.0	7.8	A	M	
	7	9.2	0.7	69.9	20.3	S	S	
L′	1	0.7	3.3	20.3	75.8	S	S	S
	2	6.5	81.0	11.1	1.3	T	K	
	3	0.0	60.8	23.5	15.7	K		
	4	88.2	6.5	0.0	5.2	A	W	
	5	49.7	9.8	18.3	22.2			
	6	5.2	11.1	13.1	70.6	S		

Table B.16: The frequency of each nucleotide at each position in the L boxes of attCs associated with IntI1

Box	Position	A	T	G	C	80	90	95
L″	1	4.8	0.6	10.1	84.5	C	S	
	2	0.5	16.9	0.2	82.5	C	Y	Y
	3	1.2	98.2	0.0	0.6	T	T	T
	4	3.2	4.5	1.7	90.7	C	C	Y
	5	97.9	0.2	0.0	2.0	A	A	A
	6	99.7	0.0	0.0	0.3	A	A	A
	7	1.7	0.9	97.0	0.5	G	G	G
L′	1	0.6	1.7	0.5	97.3	C	C	C
	2	0.0	98.8	0.8	0.5	T	T	T
	3	0.2	97.9	1.7	0.3	T	T	T
	4	98.3	1.4	0.3	0.0	A	A	A
	5	17.3	0.6	82.1	0.0	G	R	R
	6	0.8	4.1	84.0	11.1	G	S	S

Table B.17: The frequency of each nucleotide at each position in the L boxes of attCs associated with IntI4

Box	Position	A	T	G	C	80	90	95
L″	1	45.5	0.0	9.1	45.5	M	M	
	2	40.9	27.3	13.6	18.2			
	3	31.8	45.5	18.2	4.5			
	4	18.2	27.3	22.7	31.8			
	5	63.6	4.5	0.0	31.8	M	M	M
	6	81.8	9.1	4.5	4.5	A	W	
	7	18.2	13.6	27.3	40.9			
L′	1	9.1	13.6	45.5	31.8			
	2	9.1	86.4	0.0	4.5	T	W	W
	3	9.1	59.1	31.8	0.0	K	K	
	4	36.4	36.4	9.1	18.2			
	5	27.3	40.9	18.2	13.6			
	6	0.0	50.0	40.9	9.1	K	K	

Table B.18: The frequency of each nucleotide at each position in the L boxes of attCs associated with Son

Box	Position	A	T	G	C	80	90	95
L″	1	19.6	1.0	1.0	78.4	M	M	M
	2	97.9	1.0	0.0	1.0	A	A	A
	3	2.1	97.9	0.0	0.0	T	T	T
	4	1.0	1.0	1.0	96.9	C	C	C
	5	99.0	1.0	0.0	0.0	A	A	A
	6	97.9	1.0	0.0	1.0	A	A	A
	7	1.0	2.1	1.0	95.9	C	C	C
L′	1	1.0	1.0	97.9	0.0	G	G	G
	2	1.0	97.9	1.0	0.0	T	T	T
	3	1.0	99.0	0.0	0.0	T	T	T
	4	97.9	2.1	0.0	0.0	A	A	A
	5	1.0	96.9	2.1	0.0	T	T	T
	6	1.0	18.6	79.4	1.0	K	K	K

Table B.19: The frequency of each nucleotide at each position in the L boxes of attCs associated with Vfi

Box	Position	A	T	G	C	80	90	95
L″	1	29.0	0.0	63.8	7.2	R	R	
	2	0.0	95.7	0.0	4.3	T	T	T
	3	0.0	100.0	0.0	0.0	T	T	T
	4	0.0	56.5	0.0	43.5	Y	Y	Y
	5	100.0	0.0	0.0	0.0	A	A	A
	6	100.0	0.0	0.0	0.0	A	A	A
	7	0.0	0.0	98.6	1.4	G	G	G
L′	1	0.0	0.0	1.4	98.6	C	C	C
	2	0.0	100.0	0.0	0.0	T	T	T
	3	0.0	100.0	0.0	0.0	T	T	T
	4	100.0	0.0	0.0	0.0	A	A	A
	5	95.7	0.0	4.3	0.0	A	A	A
	6	0.0	27.5	7.2	65.2	Y	Y	

Table B.20: The frequency of each nucleotide at each position in the L boxes of attCs associated with Vpa

Box	Position	A	T	G	C	80	90	95
L″	1	89.5	0.0	10.5	0.0	A	R	R
	2	1.8	98.2	0.0	0.0	T	T	T
	3	0.0	96.5	0.0	3.5	T	T	T
	4	0.0	0.0	0.0	100.0	C	C	C
	5	96.5	0.0	3.5	0.0	A	A	A
	6	100.0	0.0	0.0	0.0	A	A	A
	7	0.0	0.0	96.5	3.5	G	G	G
L′	1	0.0	0.0	3.5	96.5	C	C	C
	2	0.0	100.0	0.0	0.0	T	T	T
	3	0.0	93.0	0.0	7.0	T	T	Y
	4	100.0	0.0	0.0	0.0	A	A	A
	5	94.7	1.8	3.5	0.0	A	A	R
	6	0.0	80.7	0.0	19.3	T	Y	Y

Table B.21: The frequency of each nucleotide at each position in the L boxes of attCs associated with Xca

Box	Position	A	T	G	C	80	90	95
L″	1	14.5	1.3	21.9	62.3	S		
	2	11.9	32.3	2.3	53.5	Y		
	3	3.2	94.6	1.4	0.9	T	T	W
	4	3.1	8.1	5.2	83.6	C	Y	
	5	91.7	0.5	1.9	6.0	A	A	M
	6	97.1	1.0	0.4	1.5	A	A	A
	7	3.3	1.5	80.8	14.3	G	S	S
L′	1	1.3	1.9	14.5	82.4	C	S	S
	2	1.4	95.9	2.1	0.5	T	T	T
	3	0.5	90.5	5.7	3.3	T	T	K
	4	94.9	3.5	0.4	1.3	A	A	W
	5	31.2	11.8	53.6	3.4	R		
	6	1.4	13.6	62.1	22.9	S		

Table B.22: The frequency of each nucleotide at each position in the L boxes of all the attCs identified by ISAAC

B.5 I Boxes

Box	Position	A	T	G	C	80	90	95
I″	1	9.8	9.2	13.7	67.3	S		
	2	7.8	8.5	31.4	52.3	S		
	3	4.6	6.5	84.3	4.6	G	K	
	4	68.0	15.7	10.5	5.9	W		
	5	18.3	9.2	11.1	61.4			
	6	5.2	12.4	42.5	39.9	S		
I′	1	10.5	5.2	54.2	30.1	S		
	2	13.1	8.5	30.1	48.4			
	3	4.6	22.2	64.7	8.5	K		
	4	8.5	7.2	2.6	81.7	C	M	
	5	11.8	5.2	52.3	30.7	S		
	6	11.1	0.0	64.7	24.2	S		

Table B.23: The frequency of each nucleotide at each position in the I boxes of attCs associated with IntI1

Box	Position	A	T	G	C	80	90	95
I″	1	90.4	0.5	0.9	8.3	A	A	M
	2	2.4	0.8	94.9	2.0	G	G	R
	3	2.3	2.3	94.6	0.9	G	G	D
	4	2.0	1.5	94.9	1.7	G	G	R
	5	95.6	0.3	0.2	3.9	A	A	A
	6	1.2	7.4	1.4	90.1	C	C	Y
I′	1	0.2	8.9	3.9	87.0	C	Y	Y
	2	80.4	8.1	0.6	10.8	A	M	
	3	7.8	1.4	89.5	1.4	G	R	R
	4	1.4	0.8	0.2	97.7	C	C	C
	5	2.7	2.9	0.8	93.7	C	C	Y
	6	1.1	1.8	1.2	95.9	C	C	C

Table B.24: The frequency of each nucleotide at each position in the I boxes of attCs associated with IntI4

Box	Position	A	T	G	C	80	90	95
I″	1	40.9	22.7	13.6	22.7			
	2	18.2	22.7	27.3	31.8			
	3	13.6	0.0	50.0	36.4	S		
	4	18.2	4.5	63.6	13.6	R		
	5	45.5	13.6	0.0	40.9	M		
	6	9.1	31.8	36.4	22.7			
I′	1	4.5	22.7	31.8	40.9			
	2	27.3	13.6	13.6	45.5			
	3	18.2	0.0	54.5	27.3	S		
	4	4.5	0.0	27.3	68.2	S	S	S
	5	13.6	0.0	18.2	68.2	S		
	6	9.1	22.7	27.3	40.9			

Table B.25: The frequency of each nucleotide at each position in the I boxes of attCs associated with Son

Box	Position	A	T	G	C	80	90	95
I″	1	97.9	1.0	1.0	0.0	A	A	A
	2	0.0	2.1	0.0	97.9	C	C	C
	3	0.0	2.1	97.9	0.0	G	G	G
	4	97.9	0.0	2.1	0.0	A	A	A
	5	3.1	96.9	0.0	0.0	T	T	T
	6	2.1	82.5	14.4	1.0	T	K	K
I′	1	84.5	0.0	15.5	0.0	A	R	R
	2	82.5	0.0	0.0	17.5	A	M	M
	3	97.9	1.0	1.0	0.0	A	A	A
	4	0.0	0.0	2.1	97.9	C	C	C
	5	0.0	3.1	96.9	0.0	G	G	G
	6	0.0	99.0	0.0	1.0	T	T	T

Table B.26: The frequency of each nucleotide at each position in the I boxes of attCs associated with Vfi

Box	Position	A	T	G	C	80	90	95
I″	1	92.8	1.4	5.8	0.0	A	A	R
	2	0.0	0.0	84.1	15.9	G	S	S
	3	1.4	36.2	59.4	2.9	K	K	K
	4	0.0	0.0	100.0	0.0	G	G	G
	5	100.0	0.0	0.0	0.0	A	A	A
	6	0.0	97.1	0.0	2.9	T	T	T
I′	1	0.0	92.8	2.9	4.3	T	T	Y
	2	1.4	1.4	0.0	97.1	C	C	C
	3	97.1	0.0	2.9	0.0	A	A	A
	4	0.0	0.0	0.0	100.0	C	C	C
	5	39.1	1.4	0.0	59.4	M	M	M
	6	15.9	0.0	0.0	84.1	C	M	M

Table B.27: The frequency of each nucleotide at each position in the I boxes of attCs associated with Vpa

Box	Position	A	T	G	C	80	90	95
I″	1	3.5	0.0	0.0	96.5	C	C	C
	2	3.5	3.5	0.0	93.0	C	C	H
	3	0.0	0.0	100.0	0.0	G	G	G
	4	93.0	0.0	7.0	0.0	A	A	R
	5	77.2	3.5	5.3	14.0	M	M	
	6	7.0	0.0	56.1	36.8	S	S	
I′	1	49.1	3.5	47.4	0.0	R	R	R
	2	0.0	8.8	28.1	63.2	S	S	
	3	3.5	73.7	15.8	7.0	K		
	4	0.0	0.0	0.0	100.0	C	C	C
	5	3.5	0.0	93.0	3.5	G	G	V
	6	0.0	0.0	100.0	0.0	G	G	G

Table B.28: The frequency of each nucleotide at each position in the I boxes of attCs associated with Xca

Box	Position	A	T	G	C	80	90	95
I″	1	71.2	2.7	3.4	22.7	M	M	
	2	3.3	4.1	67.8	24.9	S	S	
	3	2.4	5.0	90.1	2.5	G	G	K
	4	27.2	3.2	67.4	2.3	R	R	
	5	72.2	11.0	2.0	14.8	M		
	6	4.2	20.8	12.1	62.9	Y		
I′	1	13.0	13.3	16.0	57.7			
	2	58.9	8.6	6.3	26.2	M		
	3	20.9	8.3	67.8	3.0	R		
	4	2.2	1.6	1.3	94.9	C	C	M
	5	7.7	2.9	22.5	66.9	S		
	6	3.4	10.3	18.7	67.6	S		

Table B.29: The frequency of each nucleotide at each position in the I boxes of all the attCs identified by ISAAC