Automated identification of antibiotic resistance mutations in bacterial genomes

Creation of the ARM-find pipeline

Master’s thesis in Bioinformatics

MARTIN BOSTRÖM
Automated identification of antibiotic resistance mutations in bacterial genomes

Creation of the ARM-find pipeline

Martin Boström

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Systems Biology and Bioinformatics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017
Automated identification of antibiotic resistance mutations in bacterial genomes
Creation of the ARM-find pipeline
Martin Boström

© Martin Boström, 2017.

Supervisors: Dr. Anna Johnning and Associate Professor Erik Kristiansson,
Department of Mathematical Sciences, Chalmers University of Technology

Examiner: Associate Professor Erik Kristiansson, Department of Mathematical
Sciences, Chalmers University of Technology

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Systems Biology and Bioinformatics
Chalmers University of Technology
SE-412 96 Gothenburg

Cover: Logo for the ARM-find pipeline.

Printed and bound at
Department of Mathematical Sciences
Chalmers University of Technology and University of Gothenburg
2017
Automated identification of antibiotic resistance mutations in bacterial genomes
Creation of the ARM-find pipeline
Martin Boström
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Antibiotic resistant bacteria are a fast-growing problem, worsened by the overuse of antibiotics. In treatment of infections, it is often necessary to determine a resistance profile for the infecting bacteria in order to establish the correct treatment, and the time required to accomplish that through cultivation is sometimes long. Recent advances in next-generation sequencing techniques have decreased the cost and time requirements of whole-genome sequencing to the point where a bioinformatical approach to resistance profile determination may prove faster than the traditional one. Software tools are already available for the detection of mobile resistance genes in bacterial genomes, but to my knowledge, no open-source tools exist that detect resistance mutations. This thesis describes the creation of the ARM-find pipeline, which can find such resistance mutations in assembled bacterial genomes, including draft genomes. It comes with a resistance mutation database that currently contains fluoroquinolone resistance mutations in \( E. coli \), but is easily extensible to cover additional antibiotics and species. In addition to describing the pipeline, this thesis covers the prevalence of fluoroquinolone resistance mutations in \( E. coli \) and \( Shigella \). The pipeline was used to catalogue substitutions (in comparison to \( E. coli \) K-12 MG1655) in the genes encoding DNA gyrase and topoisomerase IV – the targets of fluoroquinolones – in all RefSeq genomes for both \( E. coli \) and \( Shigella \). Fluoroquinolone resistance mutations were found to be common, and the relative frequencies of the mutations matched what has been reported in previous studies on the subject.

Keywords: Antibiotic resistance, pipeline, fluoroquinolones, bioinformatics, \( E. coli \), \( Shigella \), mutations
Acknowledgements

First and foremost, I would like to thank my supervisors, Anna Johnning and Erik Kristiansson. Thank you both for always helping me when I needed it, and for providing me with this master thesis project; it has been the most rewarding work I have done during my time at university. Anna, I knew this was going to work out when I first walked into your office and you had the same beautiful green Zelda poster as I do - you have good taste. I would also like to thank everyone I’ve met at the Mathematical Sciences department for the lovely fika sessions, and everyone who has baked something for the weekly proper fikas. I love Thursdays now. Also, thank you Christoffer for being such an excellent cubicle mate, and for watering Jonte when I’m not around.

Finally, I would like to thank the people behind www.armfinder.com for having already taken the name I had initially wanted for my pipeline (notice the lack of -er in ARM-find). I wish you all the best in your mission to “Register, Find and Contact Armwrestlers in the World”.

Martin Boström, Gothenburg, February 2017
1

Introduction

Antibiotics are molecules that either kill or inhibit the growth of bacteria. Most antibiotics that are in use as drugs today were originally found in nature, where they are used by bacteria or fungi against other microbes [1]. They are an important defence against bacterial infections, and are essential in healthcare. However, when antibiotics are given as treatment, antibiotic resistant bacteria are given an increased chance of achieving dominance through the elimination of the non-resistant competition. Through this mechanism, overuse of antibiotics has resulted in increasing resistance among bacteria, and the trend is worsening [2]. Without effective antibiotics, we could face a world where infections that have previously been easily treatable become a death sentence.

Antibiotics work by affecting certain targets – the main ones are bacterial cell-wall biosynthesis, protein synthesis, and DNA replication and repair. There are different strategies that are used by bacteria to survive such attacks. They may alter cell permeability, e.g. by changing the amounts of efflux pumps and porins or by altering the cell wall, so that the concentration of the drug at the site of its target remains too low to cause severe harm. An alternative strategy is to modify the antibiotic so that it no longer binds to its target, or at least does so with lower affinity. The bacteria may also modify the drug’s target, either through mutations or later modifications (such as the addition of molecules at binding sites) to reduce its binding affinity to the drug [1]. Finally, metabolic pathways can be altered to no longer rely on the targeted enzyme [3].

Acquiring or improving the properties described above can be done through acquisition of novel DNA or through alterations in pre-existing DNA. In the first case, resistance genes can be transferred horizontally between bacteria by plasmids, bacteriophages, naked DNA, or transposons. These genes could for instance code for enzymes that modify the antibiotic, or replace antibiotic-targeted enzymes in metabolic pathways. In the second case, antibiotic resistance arises through step-wise mutations, with each mutation resulting in less susceptibility to the antibiotic in question. The typical example of this is a reduced binding affinity between a drug and its target, caused by mutations to that target.

Whether resistance genes or resistance mutations are the most important varies between different antibiotics. In the case of fluoroquinolones, a broad-spectrum class of antibiotics that are highly effective for treating a variety of infections, chromosomal mutations have the highest impact on resistance. Since all fluoroquinolones
have the same antibiotic mechanism, any mutation in the target genes that results in resistance to one fluoroquinolone will also yield resistance to all the others [4]. Fluoroquinolone resistance has been extensively studied in *Escherichia coli* [4], a common source of urinary tract infections. Studies have found that an alarmingly large portion of *E. coli* have achieved some degree of resistance to fluoroquinolones [2].

Because of the spread of antibiotic resistance, it is important to be able to quickly characterise resistance in an infection, so that the correct treatment may be provided. Currently, the antibiotic to give may be determined based on the symptoms of the patient, or ideally by isolating the bacteria responsible for the infection and evaluating their resistance profile. The latter is done by growing them in the presence of different antibiotics in the lab [5], which can be a slow process, especially when the infecting strain grows slowly, is difficult to cultivate, or is multi-resistant. However, we may now have another option at hand. Recent developments in next-generation sequencing (NGS) techniques have seen both the time requirement and the cost of sequencing entire genomes decrease exponentially [6]. If the genetic alterations that lead to antibiotic resistance are known, we could sequence the genomes of bacteria and use that information to infer what antibiotics are suitable for treatment. This has already been shown to be faster than cultivation-based resistance determination in the UK for the slow-growing *M. Tuberculosis* [7].

If we are to improve the speed of resistance profile determination for bacteria through whole-genome sequencing, we must have bioinformatical software tools that can find resistance-related genetic elements quickly and efficiently. We will need tools both for finding mobile resistance genes, and for finding resistance mutations, like the ones that are important for fluoroquinolone resistance. There are several software tools in existence for identifying mobile resistance genes, such as ResFinder [8], the Comprehensive Antibiotic Resistance Database (CARD) [9], and the Antibiotic Resistance Genes Database (ARDB) [10]. However, to my knowledge there are no open-source tools that are designed to identify resistance mutations in bacteria, which leads us to the aims of this project.

### 1.1 Aims

In this thesis, I have developed a pipeline for identifying antibiotic resistance mutations in bacterial genomes; it is called ARM-find, short for Antibiotic Resistance Mutation finding pipeline. To its database of resistance mutations, I have added mutations that confer resistance to fluoroquinolones in *E. coli*. To test the performance of the pipeline, I have searched for fluoroquinolone resistance mutations in all *E. coli* genomes available from NCBI’s database RefSeq, as well as in the closely related genus *Shigella*. 

The aim of the following sections is to provide a brief explanation of how fluoroquinolones work, and what their targets are, as well as to explain the different types of sequence alignments used in the pipeline, and some relevant information regarding genome assembly.

### 2.1 Fluoroquinolones and Their Targets

Fluoroquinolones are a class of synthetic antibiotics that inhibit the replication and transcription of bacterial DNA, by acting against DNA gyrase and topoisomerase IV. At high concentrations, this leads to cell death. They are broad-spectrum, and important in health-care. Fluoroquinolones are categorised into generations based on the improvements that have been made to, among other things, their half life and range of different kinds of bacteria that they are active against. What they have in common is that they are synthetic fluorinated analogues of nalidixic acid, and tend to contain a 4-pyridone-3-carboxylic acid with a ring connecting to positions 5 and 6, as shown in figure 2.1 [11].

![Figure 2.1](image_url)

**Figure 2.1:** The required pharmacophore of fluoroquinolones. Image modelled after figure from [11].

The targets of fluoroquinolones, DNA gyrase and topoisomerase IV, are both topoisomerases, meaning they participate in the supercoiling of DNA. DNA gyrase is a tetrameric enzyme that is composed of two GyrA and two GyrB subunits. It is responsible for introducing negative supercoils into DNA, which is necessary for DNA
2. Theory

replication [12]. Topoisomerase IV is homologous to DNA gyrase, and is composed of two ParC and two ParE subunits. It is involved in the separation of chromosomes during DNA replication [4]. Both enzymes cut DNA molecules in order to be able to change their coiling, and ligate them back together after having done so. The antibiotic mechanism of fluoroquinolones is that they bind to the DNA gyrase/topoisomerase IV-DNA complex, thereby stabilising it and preventing ligation [4]. This stops DNA synthesis, causing the cells to stop growing. There are several hypotheses for what causes the bactericidal effect that fluoroquinolones have. A common and likely one states that the release of DNA ends from the enzyme-DNA-drug complex triggers apoptosis [12].

The most clinically relevant cause of fluoroquinolone resistance is chromosomal mutations to the genes encoding DNA gyrase and topoisomerase IV. These mutations lower susceptibility to fluoroquinolones by reducing the binding affinity of the drug to the enzyme-DNA complex. The current knowledge of such mutations has been reviewed by Hopkins et al. [4], and that review is the basis for the resistance mutations in ARM-find’s database. Though all those mutations have been associated with increased fluoroquinolone resistance, having one does not necessarily decrease a bacterium’s susceptibility to the drug. As an example, all resistance mutations that have been found in topoisomerase IV have been accompanied by gyrA mutations. The reason for this is believed to be that the susceptibility of GyrA to fluoroquinolones must be decreased for mutations to the topoisomerase IV genes to even make a difference to survivability. Fluoroquinolone resistance can also be achieved through means other than mutations. One alternative is decreased uptake of the drug through increased amounts of efflux pumps or decreased amounts of porins, and another is mobile genes such as the plasmid-mediated qnr [4].

2.2 DNA and Protein Sequence Alignments

Both local and global alignments of nucleotide and amino acid sequences are used in this pipeline. Global alignments are only viable when both sequences are of roughly equal length, which of course is not the case when aligning a gene against a genome. A local alignment is suitable for aligning a shorter sequence against a longer one, as it does not attempt to match the entire sequences against each other in the way that a global alignment does. Instead, it finds where in the longer sequence the shorter sequence, or pieces of it, will fit. The choice between global and local alignments is also affected by sequence similarity, but for the purposes of the pipeline, sequence length differences are the deciding factor, as the sequences to be aligned are assumed to be highly similar. ARM-find uses BLAST (Basic Local Alignment Search Tool) [13] for local alignments, and MAFFT (Multiple Alignment using Fast Fourier Transform) [14] for global alignments.

BLAST is a heuristic local alignment tool. It splits the query sequence into short “words” (sequence snippets), and creates permutations of these words that are sufficiently similar to the original word, according to a scoring matrix. These words are then matched against the target sequence, and the matching positions are used
as seeds for alignment. Every such seeded alignment is extended in both directions until its alignment score decreases beyond a certain threshold compared to the maximum value during extension. After termination, the alignment is rolled back to its maximum score. The resulting alignments’ scores are then evaluated for statistical significance by comparison to random sequences. Alignments scoring above a certain threshold are kept and presented to the user [15].

MAFFT is a multiple sequence alignment tool that is capable of both local and global alignments. As the name implies, it is based on the fast Fourier transform, which is used to rapidly detect homologous segments [14]. It comes with options for several alignment strategies, including progressive methods, structural alignment methods for RNAs, and iterative refinement methods [16]. One such alignment option, the iterative refinement method G-INS-i, is used in this pipeline for global alignment. G-INS-i requires that the entire region can be aligned and attempts global alignment using the Needleman-Wunsch algorithm [17].

2.3 Genome Sequencing and Assembly

Since DNA sequencing can only generate reads of limited length, sequencing genomes necessitates the generation of many, many reads. To generate a full genomic DNA sequence, the genome has to be assembled using these reads. Assembled genomes can be in different states of completeness; genomes that have been fully assembled are called complete assemblies, whereas partially assembled genomes are called draft assemblies. In a draft assembly, the genetic sequence is typically split over many contigs, and the concept of scaffolding is important. A contig is a contiguous sequence of nucleotides known with a high degree of certainty. A scaffold can contain several contigs, between which the distance is known (or estimated), but the DNA sequence is not. The distance between contigs can be found through methods such as mate pair sequencing. The result is sequences that can include large stretches of N’s, designating unknown bases. The number of N’s between two contigs may not match the actual number of bases separating them exactly.
2. Theory
3

Methods

The pipeline was coded in Python (version 3), and is built to run on a Linux system with the BLAST, MAFFT, and EMBOSS command line interface programs installed. It was designed to find resistance mutations in a FASTA format input file containing the genomic DNA sequence of an organism. The DNA sequence may be split across many FASTA sequences in the input file. The pipeline is meant to be used with assembled genomes, and has therefore not been tested with raw sequence data. However, it was written to work with genomes in various stages of assembly, and can thus handle draft genomes. The workflow of the pipeline is explained in section 3.1, and illustrated by means of a flowchart in figure 3.1. The code is presented in appendix B.

ARM-find was built to find antibiotic resistance mutations, but the word mutation implies change, which can be a bit misleading. What the pipeline actually does is identify nucleotides and amino acids that are different in the input genome compared to a reference sequence. When mutations or substitutions are mentioned in this report, it would be more appropriate to say “difference between input genome and reference sequence”. Working this distinction into every sentence would make for cumbersome reading, however, which is why I am clarifying the nomenclature now.

3.1 Pipeline Workflow

When searching for antibiotic resistance mutations, the pipeline considers one target sequence (e.g. the DNA sequence encoding an enzyme subunit) at a time. For each target sequence, the pipeline goes through the following main steps:

1. Extracting the appropriate region of DNA, corresponding to a reference DNA sequence (most likely a gene) with known resistance mutations, using BLAST.
2. Global alignment of the extracted sequence (and its translated amino acid sequence if the target is a protein) against the reference sequence, using MAFFT.
3. Mutation calling, by comparing the extracted sequence to the reference sequence at each position.
4. Comparison of found mutations to a database of known resistance mutations.
3. Methods

Figure 3.1: Flowchart showing the workflow of the pipeline. Processes in green and blue are part of a loop that is repeated for each target sequence associated with resistance mutations for the chosen species.
3.1.1 Sequence Extraction

To identify antibiotic resistance mutations in a specific DNA sequence in a genome, that sequence must first be found in the genome, which can be accomplished through sequence alignment against a reference sequence. The reference sequences for *gyrA*, *gyrB*, *parC*, and *parE* are from *E. coli* K-12 MG1655, and were downloaded from the PATRIC database [18]. Since genes and genomes are obviously not similar in length, global alignment is not an option. For that reason, a local alignment tool is used to find the correct sequence in the genome in this pipeline - more specifically, BLAST [13], as it is the most widely used tool for local alignments, and contains all the functionality needed for this part of the pipeline. The *blastn* command is used in order to run Megablast, which is suitable for highly similar sequences (approximately 95 % sequence identity) [19]. All parameters are left at default values. The output format is set to XML (−outfmt 11) to facilitate parsing.

This first local alignment would in many cases be enough for mutation calling, as most nucleotides would be comparable between the sequences directly from the alignment. However, there are some cases which demand sequence extraction and global alignment before mutation calling. For instance, if there are mutations at the end of the sequence being aligned, those might end up not being included in the BLAST alignment. Similarly, if there is a large mismatching region, or large insertions or deletions, in the aligned sequences, BLAST will tend to split the alignment into two hits, one on each side of the mismatching region. Both of these cases would result in mutations being missed during mutation calling, and possibly even in missed frameshifts that result in a completely different amino acid sequence after translation. For these reasons, the BLAST alignment is used only to find the correct region in the genome, which is then extracted for later global alignment.

It is quite possible for the pipeline to extract more than one sequence from a BLAST alignment, for instance when a gene is split over two or more FASTA sequences in the genome. These sequences are treated separately, rather than merging them, as the final report generated by the pipeline includes what FASTA sequence in the genome any given mutation was found on. Naturally, this would not be practical if hits from different FASTA sequences were merged at this stage. Additionally, multiple sequences in the extraction can be a result of multiple copies of a gene, or of several hits in one gene, separated by a mismatching region. In the latter case, the hits are normally merged, as discussed in section 3.1.1.2, but there are scenarios where this is not desirable, as well as those where it simply fails.

3.1.1.1 BLAST Hit Extension

As the alignments made by BLAST may not extend all the way to the end of the reference sequence, extension is often necessary before extraction of the sequence. This is done simply by checking the first and last base number of the reference sequence in a BLAST hit, and whether those numbers correspond to the first and last base numbers of the entire reference sequence. If they do not, then the BLAST
hit does not cover the entire reference sequence, and may need to be extended, as shown in figure 3.2. The number of bases not covered on either side of the BLAST hit is calculated, and the pipeline attempts to retrieve those bases from the genome one by one, taking into account what strand the match was found on to determine direction in the genome and whether the base at a certain position should be taken, or its complement. BLAST hit extension will stop if the pipeline tries to access bases outside of the range of the FASTA sequence the hit was found on in the genome, or if a base in the genome is already part of another BLAST hit. The latter restraint is in place to avoid overlap between the BLAST hits with respect to the bases in the genome. This only applies to overlap in the genome, though. Overlap in the reference sequence between hits can be caused by multiple copies of a gene, for instance, and should therefore be included, so that the pipeline can report whether mutations to a given reference sequence position in the genome is the same in each instance of reference sequence overlap.

![Figure 3.2:](image)

Sometimes, BLAST will find erroneous hits that tend to match only a short part of the reference sequence. When these are extended to cover as much of it as possible, the result can be that thousands of incorrect bases are added. To combat this, the pipeline will discard all hits that are extended more than 20% of the reference sequence length, unless they meet one of the following criteria:

1. The hit neighbours another hit in the genomic sequence, after all hits have been extended. This preserves hits that would otherwise be deleted for extending over large insertions, for example in scaffolds.

2. The hit is the result of a merge of several hits, as detailed below in section 3.1.1.2. In this case, a hit could seem to be extending too far, whereas it in reality is just extending to cover the hits it was merged with.

### 3.1.1.2 Merging BLAST Hits from the Same FASTA Sequence in the Genome

There are two cases that will make BLAST find two or more correct hits in one FASTA sequence from the genome. The first is if there are multiple copies of a gene, in which case the hits should remain separate. The second case is when hits are separated by a mismatching region. In the latter case, it is useful to merge the hits.


To illustrate the problems that can be caused by not merging such hits, imagine an alignment with two hits separated by a large insertion. During hit extension, the number of bases to extend is decided based on where on the reference sequence the hit is, and does not take into account the size of the insertion. In most cases, the hits will simply be extended until they come to a base covered by the other hit, but it is also possible to miss part of the insertion during extension, depending on its size and the positions of the hits on the reference sequence. A second problem arises when the insertion size is not evenly divisible by three, thereby causing frameshift. If the hits were merged, translation of the resulting sequence would lead to a drastically different amino acid sequence, as it should. If the hits aren’t merged, the frameshift would only show in one of the hits, or it might be missed completely.

In order to tell the difference between hits that should be merged and hits that should not, there are a number of checks in place. For this explanation, let us call the distance between two adjacent hits in the genome genome distance, and the distance between the same hits in the reference sequence reference distance (illustrated in figure 3.3). If reference distance < 100, the hits will be merged. This allows for some leeway for deletions and substitutions, both of which cause increased reference distance. It also allows any insertion size, as insertions do not change the reference distance. In addition to the above criterion, any pair of adjacent hits that satisfy the condition \(|\text{genome distance} - \text{reference distance}| < 100\) will be merged. This allows for combinations of substitutions and insertions to pass more freely.

![Figure 3.3](image)

**Figure 3.3:** Two BLAST hits mapping to different parts of the reference sequence, with reference distance and genome distance explained.

Even if the criteria above are met, there are a few more checks that must be cleared for hit merging to occur. In order to avoid merging hits from different copies of a gene, as well as to avoid other complications, the hits must:

- be on the same DNA strand in the genome.
- satisfy the condition genome distance ≤ reference sequence length.
- not overlap in the genome.
- come in the expected order in both the genome and in the reference sequence, with respect to which strand they are on. Out of order hits are likely either bad hits or parts of different gene copies.
3. Methods

- not be separated by a region with a high percentage (>30%) of N’s. This stops merging of hits separated by regions of uncertain lengths in scaffolds, where merging is likely to cause erroneous frameshift.

If there are more than two hits in a FASTA sequence in the genome, each adjacent pair of hits will be considered for merging separately. Then, unbroken chains of pairs accepted for merging will be marked for merging into one hit, as shown in figure 3.4.

![Figure 3.4](image)

**Figure 3.4:** Example of merging multiple hits from one FASTA sequence. Each pair is evaluated for merging separately. The pairs are then added together until a pair that was not accepted for merging is encountered. In this case, all pairs were accepted for merging except 3-4, which results in two final hits, one with hits 1-3, and one with hits 4 and 5.

Finally, when all merging decisions are finished, all hits except the last one in each merge set are deleted, and the value of genome distance – reference distance is saved for each merge pair. The last hit can then be extended as explained in section 3.1.1.1, with the only change being that extension distance is increased by the saved distance values for every involved merge pair, as shown in figure 3.5. This compensates for the change in length due to insertions or deletions between the merge pairs. If the saved distance value is negative, the extension will be shortened instead of lengthened, which is necessary for hit merging when the hits are separated because of a deletion, rather than an insertion or substitutions.

![Figure 3.5](image)

**Figure 3.5:** Once a group of hits to be merged has been decided, only the last hit (here, hit 2) is kept. Extension length is modified by genome distance – reference distance. This difference is indicated in black. Regular extension would only be based on the distance to the end of the reference sequence, which for hit 2 is as long as the green and white segments of the reference sequence. Extra extension (black) is necessary when there is an insertion between the hits, as in this case.
3. Methods

3.1.1.3 Post-Extraction Sequence Modifications

Once the appropriate DNA sequences have been extracted from the genome, some processing is necessary in preparation for global sequence alignment. If the sequence extraction yielded hits that do not cover the entire reference sequence, then the length of those hits will be shorter than the reference, which is suboptimal for a global alignment. Apart from making mutation calling more arduous, it could also result in faulty alignments, with the hit being stretched to cover more of the sequence than it should. To solve this, all hit sequences that do not cover the entire reference sequence are padded with N’s, signifying unknown bases. To calculate how many N’s should be added to each side of the sequence, the number of bases added during hit extension is subtracted from the number of bases that the original BLAST hit missed from the reference sequence, for each side. With this information, the hit can be padded with the appropriate amount of N’s on either side. It is still possible for the hit and the reference sequence to be of different lengths, but only because of indels, which is as it should be.

Another benefit of N padding is avoiding frameshift during translation. If a hit does not cover the beginning of a gene and is not padded, the likelihood of an erroneous frameshift is high. Padding with the correct amount of N’s solves that problem. Hence, translation of hit sequences is done after padding, if they code for proteins.

Translation is performed with the transeq command of the EMBOSS package [20]. All reference sequences that are to be translated start with a start codon and end with a stop codon in the pipeline’s sequence database. The BLAST alignment, hit extension, and sequence padding discussed above make sure that the hits are correctly lined up against the reference sequence, starting with the start codon, and ending with the stop codon, which means that erroneous frameshift is unlikely to cause problems. Hence, transeq is run with the argument -snucleotide1 to limit translation to the first reading frame. Furthermore, the argument table=11 is used to set the translation table to that of bacteria.

While the pipeline database currently only contains protein-coding sequences, it is possible that future additions might include DNA sequences that do not code for proteins, such as promoters or sequences coding for rRNA. To distinguish between sequences that should be translated and sequences that should not, the pipeline checks the mutations database, to see if there are any resistance mutations for amino acids for a given target. If there are, then the sequence must code for a protein, and should be translated. If there are not, then it should not be translated. This holds true as long as no protein-coding sequences without resistance mutations for amino acids are added to the mutations database, but since there would be no point in adding such a sequence, this is not a problem.

Finally, the range of any overlaps between the BLAST hits are saved for later reporting. At this stage, only nucleotide sequence overlap is saved, as the BLAST alignment files provide sufficient information for finding said overlap, whereas the amino acid sequences need to be aligned before overlap can be calculated.
3. Methods

3.1.2 Global Alignment

For the purposes of this pipeline, there were only a few specific requirements for a global alignment tool. Since the sequence extraction can yield multiple hits, it would be much more practical to work with multiple alignment, rather than aligning every hit sequence against the reference sequence one by one. Additionally, to be able to call mutations that add or remove stop codons, it was necessary for the alignment tool to not only be able to accept nucleotide and amino acid codes, but special characters as well, such as the asterisk (*) used to demark a stop codon. MAFFT [14] meets these requirements, and was thus chosen for the role. MAFFT is run and adapted for global alignment through use of the `ginsi` command, with all parameters at default values, apart from the optional argument `--anysymbol`. This is necessary in order to include stop codon asterisks in the alignment. Global alignment is done for both the nucleotide sequence and, in case the target is a protein, the amino acid sequence, to enable mutation calling. After global alignment of the amino acid sequence, the ranges of potential overlaps between translated BLAST hits are saved for later reporting.

3.1.3 Mutation Calling and Identification of Resistance Mutations

During mutation calling, DNA and protein sequences are handled separately, and one hit at a time. For each hit, every position in the sequence is compared one by one. For each position in the reference sequence that is not a hyphen, signifying a gap, a counter is incremented. This is done to keep track of which number in the reference sequence a certain position in the alignment corresponds to, so that the correct position of mutations can be reported. During mutation calling, information on substitutions, insertions, and deletions is saved, but also information on what positions in the reference sequence are covered by the hits. This information is later used to generate reports on mutation coverage.

Once all mutations have been called for a given target sequence, they are compared with all resistance mutations in the pipeline’s database for that target. If there are matches, the relevant antibiotic (or antibiotics) is associated with that mutation in the report. Since the `mutations` database could contain resistance mutations that the reference sequence itself has, all resistance-associated positions are saved during mutation calling. When comparing called mutations to the database, the resulting “non-mutations” (e.g. S83S) are skipped, unless they confer antibiotic resistance.

The database is split into two files, `targets` and `mutations`, in order to minimise repetition of data that could increase the risk of mistakes during data entry. The `targets` file simply associates every target DNA sequence with a unique identifier (target ID), and lists what species the sequence is from (see table 3.1).
### 3. Methods

#### Table 3.1: The targets database lists all target sequences, along with with the associated species.

<table>
<thead>
<tr>
<th>Target ID</th>
<th>Species</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>target0000001</td>
<td><em>E. coli</em></td>
<td>DNA gyrase - subunit GyrA</td>
</tr>
<tr>
<td>target0000002</td>
<td><em>E. coli</em></td>
<td>DNA gyrase - subunit GyrB</td>
</tr>
<tr>
<td>target0000003</td>
<td><em>E. coli</em></td>
<td>Topoisomerase IV - subunit ParC</td>
</tr>
<tr>
<td>target0000004</td>
<td><em>E. coli</em></td>
<td>Topoisomerase IV - subunit ParE</td>
</tr>
</tbody>
</table>

The mutations database contains the resistance mutations, where every entry is linked to the appropriate row in the targets database by its target ID. For every mutation in the database, there is information on what kind of antibiotics the conferred resistance applies to, its position in the target sequence, what nucleotide or amino acid occupies it in the reference sequence, and what the substituted nucleotide or amino acid is. A few example lines from the mutations database are shown in table 3.2.

#### Table 3.2: The mutations database lists all antibiotics resistance mutations, and is linked to the targets database by target ID.

<table>
<thead>
<tr>
<th>Target ID</th>
<th>Antibiotics</th>
<th>Nucleotide/protein</th>
<th>Position</th>
<th>Reference nucleotide/amino acid</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target0000001</td>
<td>Quinolones</td>
<td>prot</td>
<td>84</td>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>target0000001</td>
<td>Quinolones</td>
<td>prot</td>
<td>84</td>
<td>A</td>
<td>V</td>
</tr>
<tr>
<td>target0000003</td>
<td>Quinolones</td>
<td>prot</td>
<td>57</td>
<td>S</td>
<td>T</td>
</tr>
</tbody>
</table>

#### 3.1.4 Reporting

Three kinds of reports are created. The first one lists the mutations that were found, and whether or not these are associated with antibiotic resistance. The second report lists all known resistance mutation positions that the pipeline was unable to evaluate in the input genome. This can either be because the position was not covered by the BLAST hits, or because of an ambiguous base code. Both cases mean it is unknown whether a resistance mutation is present or not. The second report is included so that the user will know if the absence of a resistance mutation in the first report means that the mutation is not present, or if it just means that no information on it is available. Finally, the third report lists all target sequences that are not covered, in order to clearly show when a target sequence is not covered by any BLAST hits.
3. Methods

3.2 Pipeline Arguments

The behaviour of the pipeline can be controlled with several arguments, which are listed in table 3.3. The only required arguments are \texttt{--infile} and \texttt{--species}.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Short form</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{--infile}</td>
<td>\texttt{-i}</td>
<td>Path of the genome file to be analysed.</td>
</tr>
<tr>
<td>\texttt{--out}</td>
<td>\texttt{-o}</td>
<td>Takes path to output directory. Default: create output directory in current working directory. If existing path is specified, files in that directory may be removed if names conflict.</td>
</tr>
<tr>
<td>\texttt{--species}</td>
<td>\texttt{-s}</td>
<td>Takes abbreviated name of the species of the input genome (e.g. “E. coli” - quotation marks are necessary due to the space in the name).</td>
</tr>
<tr>
<td>\texttt{--list_species}</td>
<td></td>
<td>Lists all currently supported species and exits.</td>
</tr>
<tr>
<td>\texttt{--verbose}</td>
<td>\texttt{-v}</td>
<td>Sets output level to \texttt{verbose}.</td>
</tr>
<tr>
<td>\texttt{--quiet}</td>
<td>\texttt{-q}</td>
<td>Suppresses all printed output.</td>
</tr>
<tr>
<td>\texttt{--logfile}</td>
<td>\texttt{-l}</td>
<td>Redirects output (\texttt{verbose}) to log file. If no path is specified, it is placed in the output directory.</td>
</tr>
<tr>
<td>\texttt{--ext_program_output}</td>
<td></td>
<td>Includes external programs’ output in the pipeline’s output.</td>
</tr>
<tr>
<td>\texttt{--keep_tmp_files}</td>
<td></td>
<td>Prevents deletion of temporary files (alignments, translations etc.).</td>
</tr>
<tr>
<td>\texttt{--BLAST_perc_identity}</td>
<td></td>
<td>Takes a float ($0 \leq x \leq 100$) for “percent identity cut-off” in BLAST. Default is the same as Megablast default (95 %)</td>
</tr>
<tr>
<td>\texttt{--report_all_coverage}</td>
<td></td>
<td>Includes all missing positions in coverage report, not just those related to antibiotic resistance.</td>
</tr>
</tbody>
</table>

3.3 Substitution Study in \textit{E. coli} and \textit{Shigella}

All RefSeq genomes for \textit{E. coli} and every species of the \textit{Shigella} genus were downloaded from NCBI [21]. All RefSeq genomes are complete, but can be in varying stages of assembly, making them suitable for testing the pipeline. Knowledge of fluoroquinolone resistance mutations in \textit{E. coli} most likely applies to the genus \textit{Shigella} as well, due to its close relationship to \textit{E. coli}. Before pathogenic forms of \textit{E. coli} were discovered, \textit{Shigella} was classified as a separate genus due to its clinical significance. Later research has clearly shown that they are one species [22]. Therefore, \textit{Shigella} was included in this study.

The downloaded genomes were all analysed in the pipeline for fluoroquinolone resistance mutations, generating data on mutations in the DNA gyrase subunit genes \textit{gyrA} and \textit{gyrB}, as well as the topoisomerase IV subunit genes \textit{parC} and \textit{parE}. All subsequent analysis was done for \textit{E. coli} and \textit{Shigella} separately. The number of substitutions in every position in both the nucleotide sequence and the amino acid sequence for all four subunits was summed up across the genomes. All substitutions found after an indel in a genome were discarded in order to avoid counting substitutions caused by frameshift, as global alignment of a frameshifted amino acid sequence produces very erratic results. Additionally, since overlapping BLAST hits
can result in multiple substitutions in the same reference position in one genome, the extra substitutions in every reference position with more than one substitution were counted. This was done in order to check if the percentage of genomes with substitutions would be overestimated, and if so, by how much. For the amino acid sequences, the most frequent substitution positions (>3 % of genomes) were also checked for which amino acids the substitutions were to. For the nucleotide sequences, analysis of the ratio of synonymous versus non-synonymous mutations was performed. This was done by checking whether an amino acid substitution resulted for each single nucleotide substitution. Analysis and graph creation was done in R and Python.
3. Methods
In this section, two different kinds of results are presented. The first is an example of the output of the pipeline, and the second is the results of the substitution study.

4.1 Pipeline Output

A few example lines from a report file that could be generated by the pipeline are shown in table 4.1. Target names have been abbreviated to fit the page, and would normally read “DNA gyrase - subunit GyrA” etc. “Involved antibiotics” shows which classes of antibiotics are present among the resistance mutations for a given target in the database. Called mutations are listed under “Mutation”. The first letter indicates the base in the reference sequence, the following number its position in the reference sequence, and the remainder the mutation. Hyphens represent deletions, and multiple letters indicate insertions. For instance, “C231CT” means a T was inserted after the C at position 231. “Nucl/prot” simply indicates whether a given mutation is in the DNA or the amino acid sequence. Should a mutation (e.g. S83L in GyrA) confer resistance to an antibiotic, or antibiotics, they will be listed under “Resistance”.

Table 4.1: Example lines from a report file that could be generated by the pipeline.

<table>
<thead>
<tr>
<th>Target name</th>
<th>Involved antibiotics</th>
<th>Mutation</th>
<th>Nucl/prot</th>
<th>Resistance</th>
<th>FASTA sequence(s)</th>
<th>Mutation prevalence</th>
<th>Sequence coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GyrA</td>
<td>Quinolones</td>
<td>C248T</td>
<td>nucl</td>
<td>'1'</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GyrA</td>
<td>Quinolones</td>
<td>S83L</td>
<td>prot</td>
<td>Quinolones</td>
<td>'1'</td>
<td></td>
<td>Full</td>
</tr>
<tr>
<td>GyrB</td>
<td>Quinolones</td>
<td>C231CT</td>
<td>nucl</td>
<td>'1'</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParC</td>
<td>Quinolones</td>
<td>G239T</td>
<td>nucl</td>
<td>'2', '3', '3', '4'</td>
<td>0.25</td>
<td>Partial, w/ start</td>
<td></td>
</tr>
<tr>
<td>ParC</td>
<td>Quinolones</td>
<td>G239C</td>
<td>nucl</td>
<td>'3', '3', '2', '4'</td>
<td>0.5</td>
<td>Partial, w/o start</td>
<td></td>
</tr>
<tr>
<td>ParC</td>
<td>Quinolones</td>
<td>G239C</td>
<td>nucl</td>
<td>'3', '3', '2', '4'</td>
<td>0.5</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>ParE</td>
<td>Quinolones</td>
<td>T1372</td>
<td>nucl</td>
<td>'5'</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“FASTA sequence(s)” lists the FASTA sequence in the input genome that the mutation was found on. If a reference sequence position is covered by two or more BLAST hits, the FASTA sequences they were found on will all be listed. For multi-copy genes within one FASTA sequence, the same sequence can be listed more than once. For every mutation whose position is found on several FASTA sequences,
the first sequence listed under “FASTA sequence(s)” will be the sequence where the mutation was found. This is illustrated by the three mutations in position 239 in ParC in table 4.1. Note that two were found on the same FASTA sequence, and that the position was covered four times. Since only three mutations are reported, there must be one place where the base is not mutated.

Whenever a reference sequence position is covered by more than one BLAST hit, the mutations at that position, or lack thereof, can be different from each other. The degree to which the mutations are the same can be read from the “Mutation prevalence” column, where

\[
\text{Mutation prevalence} = \frac{\text{Number of BLAST hits with same mutation}}{\text{Number of BLAST hits covering given position}}
\]

Taking the example of position 239 in ParC again (table 4.1), G239C has a mutation prevalence of 0.5, since it occurs twice among four hits (including the non-mutated base), while the same value for G239T is 0.25 (one mutation in four hits).

Finally, “Sequence coverage” provides information on how much of the reference sequence was covered by the extended BLAST hit. The possible levels are “Full”, “Partial w/ start”, and “Partial w/o start”. Whether the start of the sequence is included or not in a partial hit is of interest, as the reading frame cannot be guaranteed to be correct if the start of the sequence is missing in a hit. There may be indels that change it before the start of the hit.

Positions that are not covered by the extended BLAST hits are listed in the mutations not covered report file (see table 4.2). By default, only positions where mutations can confer antibiotic resistance are listed, but all non-covered positions can be included through the use of an optional argument (\texttt{--report_all_coverage}) to the pipeline. In case no positions in a target sequence are covered, that sequence’s name will be listed in the targets not covered report (see table 4.3).

\textbf{Table 4.2:} Excerpt from a mutations not covered report file, with optional argument given to include positions not related to antibiotics resistance mutations. Target names have been abbreviated.

<table>
<thead>
<tr>
<th>Target name</th>
<th>Involved antibiotics</th>
<th>Not covered</th>
<th>Nucl/prot</th>
<th>Resistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParE</td>
<td>Quinolones</td>
<td>458 prot</td>
<td></td>
<td>Quinolones</td>
</tr>
<tr>
<td>ParE</td>
<td>Quinolones</td>
<td>459 prot</td>
<td></td>
<td>Quinolones</td>
</tr>
<tr>
<td>ParE</td>
<td>Quinolones</td>
<td>460 prot</td>
<td></td>
<td>Quinolones</td>
</tr>
</tbody>
</table>

\textbf{Table 4.3:} Example of a targets not covered report file.

<table>
<thead>
<tr>
<th>Target name</th>
<th>Involved antibiotics</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA gyrase - subunit GyrB</td>
<td>Quinolones</td>
</tr>
<tr>
<td>Topoisomerase IV - subunit ParC</td>
<td>Quinolones</td>
</tr>
<tr>
<td>Topoisomerase IV - subunit ParE</td>
<td>Quinolones</td>
</tr>
</tbody>
</table>
4. Results

4.2 Substitution Study Results

Due to the similarity of the results between *E. coli* and *Shigella*, and the large number of graphs, the results for *E. coli* will be presented first, and in greater detail. A comparison between *E. coli* and *Shigella* may be found in section 4.2.3.

Running all 4660 *E. coli* genomes through the pipeline sequentially took 5 hours, 41 minutes, and 21 seconds, meaning the average analysis time was 4.4 seconds per genome. This is by no means a controlled measurement, as no care was taken to limit other processes running on the system, and all the genome files were copied and unzipped during the run, but it serves as an indication of approximately how long it takes the pipeline to search a genome for fluoroquinolone resistance mutations.

4.2.1 Substitution Analysis

There was considerable variability in the nucleotide sequence of the analysed sub-units, as shown in figures 4.1 (*gyrA*), 4.3 (*gyrB*), 4.5 (*parC*), and 4.7 (*parE*), where the substitution percentage is shown in grey. As an overlay on the graphs, the percentage of genomes with single nucleotide substitutions that cause amino acid substitutions is shown in red. These results clearly show that most substitutions are synonymous. Comparing the non-synonymous nucleotide substitutions to the amino acid substitutions - see figures 4.2 (GyrA), 4.4 (GyrB), 4.6 (ParC), and 4.8 (ParE) - it is clear that they match well.

Of all the *E. coli* genomes, 70 contained insertions in the topoisomerase genes. Many of these were checked to see if the insertion identification was correct, which resulted in two common insertion types being identified. The first was the insertion of a large amount of N’s, typical of a scaffold where the sequence of bases between two contigs is unknown, and the distance is estimated. These cases did not cause frameshift, as they resulted in one BLAST hit on either side of the insertion. Those hits were protected from merging due to the high N content of the mismatching region. This is exactly the case that feature was designed for. In the second type of insertion, which caused frameshift, the insertion was part of a repeat sequence of one base, such as an extra A in a sequence of several A’s, constituting a homopolymer.

Because some genomes (90 in the case of *E. coli*) contained overlapping BLAST hits, there were cases of multiple nucleotide substitutions in one reference position in the same genome. For the vast majority of these cases, there were only one or two extra substitutions in a reference position across all genomes. Ten reference positions had more than ten extra substitutions, with the highest amount being 35. That means that among all the nucleotide substitution graphs for *E. coli* (figures 4.1, 4.3, 4.5, and 4.7), there is one bar that is 0.75 percentage units (35/4660) too high due to overcounting of substitutions, and that some other bars are affected, but to a much lesser extent. Most of these extra substitutions (~89% of affected positions) were synonymous mutations. For the amino acid sequences, only eight reference positions had extra substitutions, with the highest amount being two.
4. Results

**Figure 4.1:** Substitution percentage for the DNA sequence of *gyrA* in *E. coli*. Red bars denote non-synonymous mutations.

**Figure 4.2:** Substitution percentage for the amino acid sequence of subunit GyrA in DNA gyrase in *E. coli*. 
4. Results

**Figure 4.3:** Substitution percentage for the DNA sequence of *gyrB* in *E. coli*. Red bars denote non-synonymous mutations.

**Figure 4.4:** Substitution percentage for the amino acid sequence of subunit GyrB in DNA gyrase in *E. coli*.
4. Results

Figure 4.5: Substitution percentage for the DNA sequence of parC in *E. coli*. Red bars denote non-synonymous mutations.

Figure 4.6: Substitution percentage for the amino acid sequence of subunit ParC in topoisomerase IV in *E. coli*. 
Figure 4.7: Substitution percentage for the DNA sequence of *parE* in *E. coli*. Red bars denote non-synonymous mutations.

Figure 4.8: Substitution percentage for the amino acid sequence of subunit ParE in topoisomerase IV in *E. coli*. 
4.2.2 Amino Acid Breakdown of Substitutions

For most amino acid positions where the substitution percentage was above 3 % (lower percentage positions were not considered), there was only one amino acid that the substitutions were to, or one that composed the vast majority of the substitutions, as shown in figures 4.9 (GyrA), 4.10 (GyrB), 4.11 (ParC), and 4.12 (ParE). The amount of genomes with missing data was low for every position. Missing data can be a result of lack of coverage by BLAST hits, or ambiguous base codes in the triplet, both resulting in the unknown amino acid X during translation. It can also be caused by indels prior to the position in question, as all positions after an indel are disregarded to avoid counting substitutions caused by frameshift and the resulting bad global alignment.

In terms of fluoroquinolone resistance, many of the genomes had resistance mutations in positions 83 and 87 in GyrA (figure 4.9), 80 and 84 in ParC (figure 4.11), and 529 in ParE (figure 4.12). No resistance mutations were found in GyrB in any genome.

Figure 4.9: Substitution percentage for all amino acid positions with substitutions in >3 % of genomes for GyrA in E. coli, with amino acid breakdown. White diagonal lines indicate fluoroquinolone resistance. Right-hand bars show percentage of genomes with unknown amino acids.
4. Results

**Figure 4.10:** Substitution percentage for all amino acid positions with substitutions in >3% of genomes for GyrB in *E. coli*, with amino acid breakdown. None of the detected substitutions were associated with resistance. Right-hand bars show percentage of genomes with unknown amino acids.

**Figure 4.11:** Substitution percentage for all amino acid positions with substitutions in >3% of genomes for ParC in *E. coli*, with amino acid breakdown. White diagonal lines indicate fluoroquinolone resistance. Right-hand bars show percentage of genomes with unknown amino acids.
4. Results

Figure 4.12: Substitution percentage for all amino acid positions with substitutions in >3 % of genomes for ParE in *E. coli*, with amino acid breakdown. White diagonal lines indicate fluoroquinolone resistance. Right-hand bars show percentage of genomes with unknown amino acids.

4.2.3 Comparison between *E. coli* and *Shigella*

The same types of graphs were generated for *Shigella* as for *E. coli*. In the interest of readability, they may be found in appendix A.

In the *Shigella* genomes, fewer nucleotide positions had substitutions than in the *E. coli* genomes, as evidenced by comparing figures 4.1, 4.3, 4.5, and 4.7 (*E. coli*) to figures A.1, A.3, A.5, and A.7 (*Shigella*). However, the substituted positions had a higher substitution percentage in general. Just like for *E. coli*, synonymous substitutions were much more common than non-synonymous ones. The profiles of non-synonymous mutations in figures A.1, A.3, A.5, and A.7 also matched the amino acid substitutions in figures A.2, A.4, A.6, and A.8 well, apart from one position. In *gyrA*, position 1270 has a non-synonymous mutation (C1270T) in approximately 15 % of the analysed genomes (figure A.1), whereas the corresponding amino acid (L423) is not substituted in any genomes (figure A.2). However, reviewing the raw mutation data (not shown), it became clear that every genome with a mutation in nucleotide 1270 also had a C1272G substitution, in the same codon. Together, these two mutations result in an unchanged amino acid, even though the C1270T mutation alone would have resulted in an amino acid substitution from leucine to phenylalanine, as shown in table 4.4.

The amino acid substitutions that are not related to fluoroquinolone resistance were mostly different between *E. coli* (figures 4.9, 4.10, 4.11, and 4.12) and *Shigella*
Table 4.4: The resulting codons and amino acids for all combinations of C1270T and C1272G mutations in *gyrA*.

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Codon</th>
<th>Amino acid</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (reference)</td>
<td>CTC</td>
<td>Leucine</td>
</tr>
<tr>
<td>C1270T</td>
<td>TTC</td>
<td>Phenylalanine</td>
</tr>
<tr>
<td>C1272G</td>
<td>CTG</td>
<td>Leucine</td>
</tr>
<tr>
<td>C1270T and C1272G</td>
<td>TTG</td>
<td>Leucine</td>
</tr>
</tbody>
</table>

(figures A.9, A.10, A.11, and A.12). In terms of resistance mutations, the same positions had substitutions in GyrA in *Shigella* as in *E. coli*, although with slightly lower substitution percentages in *Shigella*. Neither *E. coli* nor *Shigella* genomes had any resistance mutations in GyrB. For ParC, *Shigella* genomes had resistance mutations in position 84, but not in 80, unlike *E. coli*. Finally, there were no resistance mutations in ParE in *Shigella* genomes, whereas *E. coli* genomes had resistance-associated substitutions in position 529.
4. Results
In this thesis, I have constructed a pipeline that automatically identifies resistance mutations in complete genomes, including draft genomes. I have created a database for resistance mutations that currently contains chromosomal mutations conferring resistance to fluoroquinolones in *E. coli*, but is easily extensible to cover more classes of antibiotics and other organisms. In order to test the pipeline, as well as to put it to good use, I have analysed every RefSeq genome available from NCBI for both *E. coli* and *Shigella*, and shown the prevalence of fluoroquinolone resistance mutations in those genomes. In the sections below, I will discuss pipeline design choices and their implications, as well as the results of the RefSeq genome analysis study.

### 5.1 Pipeline Design

Much work was put into making the pipeline able to handle difficult situations, including genes split over multiple FASTA sequences, multiple copies of genes, similar but unrelated sequence snippets, and intra-gene regions of sizeable dissimilarity. For many of these situations, test cases had to be constructed artificially, as they were not encountered at all during the large-scale genomic analysis of this thesis. While all tested scenarios worked well, real data is preferable for finding oversights. Additionally, since only four genes have been used during testing, it is possible that future additions of antibiotics could unearth new challenges. That being said, I have made an effort to keep the process of the pipeline as general as possible, so that it works for all manner of target sequences.

There are many cases of cut-off values in the pipeline that are used to distinguish between different situations, such as when BLAST hits should or should not be merged, or which hits should be kept and which should be discarded. These values were set to handle situations that were either thought likely to occur, or that were encountered during the large-scale genomic analysis. The values are ad hoc, in that they yield the correct results for the data analysed in this thesis, but have in no way been optimised. It is also important to point out that although the cut-off values work well for the current state of the pipeline, there is no guarantee that they will also work for any target sequences and species that might be added in the future.
5. Discussion

It is important to note that while the pipeline can find resistance mutations, it provides no information on what level of resistance can be expected from a certain mutation, or combination of mutations. That is left to the user to evaluate. Additionally, one should not assume that the list of resistance mutations for any antibiotic in the database is exhaustive. There may be other factors that contribute to antibiotic resistance as well, such as increased expression of efflux pumps, decreased expression of outer membrane porins, or mobile resistance genes.

The fact that this pipeline uses external programs for different tasks makes it susceptible to future parsing problems. If BLAST or MAFFT were to change the formats of their reports, for instance, there could be compatibility issues. The likelihood of that happening might be low, but it is an unavoidable risk when depending on other programs.

5.1.1 Sequence Extraction

Using Megablast for local alignment relies on the aligned sequences being highly similar (approximately 95% sequence identity) [19]. This is a reasonable expectation, as sequences are unlikely to differ more in the same species. Additionally, local regions of high dissimilarity do not cause problems here, as BLAST will simply find hits that meet the sequence identity requirement on either side of such regions, which can then be analysed separately. Using the database entries for one species to analyse another could be problematic, but in those cases it would not be safe to assume that the resistance mutations are the same anyway, so the 95% sequence identity issue should not be a matter of great concern. If the pipeline were to be used on genomes that were not yet assembled, for instance directly on raw sequencing data, the shorter sequence lengths might make the 95% cut-off more limiting. However, that use case would most likely require extensive reworking of the pipeline, and the BLAST sequence identity cut-off would not be the biggest issue by far.

Another potential problem of the sequence extraction part of the pipeline concerns indels in the beginning and ends of genes. If BLAST does not match the ends of a target sequence, hit extension will retrieve the missed bases. However, if there is an insertion or a deletion in the region outside of the BLAST hit, then the number of retrieved bases will be off. This would result in not covering the first base in case of an insertion, or retrieving bases outside of the gene for deletions. The erroneous bases at the ends would then likely be misidentified as substitutions, rather than indels. However, the likelihood of indels in the beginning and end of genes was deemed to be low. Even so, it might be advisable to double-check mutations in the first and last bases of a sequence, should any be found.

Another challenge posed by indels comes from cases where two BLAST hits are separated by a region that contains an indel. If a hit extends to cover the indel, but does not reach the end of the gene, the number of N’s added for padding will be off. Naturally, the worst possible outcome of this is frameshift. In most cases, hit merging will avert the issue completely through the extension distance compensation involved in the process. However, it is possible that the hits will not be merged, most
5. Discussion

notably if they are separated by a large amount of N’s, which can occur in scaffold genomes. For these situations, the “sequence coverage” column of the main report file comes in handy, since the values “Full” and “Partial w/ start” will indicate that the reading frame is correct, as the start of the sequence is covered.

Deletion of bad BLAST hits is based on deleting hits that extend too far. This works well for short sequence matches outside the target sequence, as hit extension will be much longer than what is deemed acceptable by the pipeline, resulting in the deletion of the erroneous hit. However, if BLAST finds a hit just outside of a target sequence’s position in the genome, it is theoretically possible for it to be protected from deletion by extending up to a legitimate hit in the correct place in the genome. In the worst case scenario, it might even claim bases in the end of the target sequence, if its hit extension is handled before the legitimate hit. A potential solution to this could be to check that all BLAST hits that are in close proximity to each other are arranged in a logical sequence with respect to what part of the reference sequence they match, and delete any that do not fit. However, as this problem was not encountered during this thesis, it was deemed to be too unlikely to merit the amount of work required to solve it.

5.1.2 Global Alignment

To get the correct reading frame during translation, the pipeline relies on alignment to the start codon, which will always be at the start of the stored target sequences. The problem with this is that the reading frame could be changed by the gene end indels described above. A possible solution to this could be alternative methods of identifying the reading frame, but the low likelihood of gene end indels makes this a most likely unnecessary addition.

When the reading frame is changed due to indels, the global alignment of amino acid sequences runs into problems, as the sequences to be aligned will tend to be very dissimilar. The report will then include a very large of amount of insertions, deletions, and substitutions in the amino acid sequence. Diagnosing this situation is easy when it occurs, though, as a nucleotide insertion or deletion will be immediately followed by all the mismatches in the amino acid sequence. However, any automated use of the pipeline will require checks for indels in the nucleotide sequence, so that frameshifted global alignment results may be handled, just as was done for the substitution study in this thesis.

5.2 Substitution Study

When analysing the substitution percentages in the four subunits, it is important to remember that “substitution percentage” refers to the percentage of genomes that have a different base or amino acid in any given position compared to the reference sequence. The reference is not the origin of all other strains, and therefore no conclusions can be drawn about which genomes are mutated; they can simply be
compared. One cannot infer mutation likelihood from the graphs presented in this thesis. It should also be noted that the analysed genomes do not necessarily represent all *E. coli* and *Shigella* genomes well. It would not be unreasonable to suspect that clinical isolates are overrepresented among the RefSeq genomes, thereby possibly resulting in an overestimation of the prevalence of resistance mutations across all genomes. After all, the entire reason that *Shigella* is categorised as a genus rather than a subgroup of *E. coli* is its pathogenicity [22], highlighting our focus on effects on humans in our way of thinking of these organisms.

When the insertions that were found in some genomes were investigated, two common insertion types were identified. One of these was the frameshift-causing insertion of one base into a repeat sequence of the same base, such as an extra A inserted into a sequence of several A’s. As repeating bases are more difficult to sequence and assemble correctly, these insertions are probably caused by sequencing or assembly problems. Since both DNA gyrase and topoisomerase IV are essential, any gene with frameshift would cause death, if it were the only copy of the gene. Though the genomes with this kind of insertion were not thoroughly checked for extra gene copies, sequencing or assembly problems seem much more likely than multiple gene copies, with one of them being completely useless because of frameshift.

For some reference positions, overlapping BLAST hits resulted in there being more than one substitution to a reference position in the same genome. However, the relatively low amount of extra substitutions means the resulting overestimation of substitution percentages is negligible. This is especially true for the amino acid sequences, where only very few reference positions had extra substitutions, and at most two per position. That is certainly few enough not to be visible in the presented graphs.

The non-synonymous nucleotide substitutions shown in figures 4.1, 4.3, 4.5, and 4.7, would not necessarily have had to match the amino acid substitutions in figures 4.2, 4.4, 4.6, and 4.8 (and likewise for the corresponding *Shigella* figures), as multiple mutations in a triplet could result in an exchanged amino acid where only one mutation would not have. However, as the profiles of the non-synonymous substitutions and the amino acid substitutions are so similar, one may conclude that the vast majority of amino acid substitutions were caused by single nucleotide mutations. The exception to this is the mutations in nucleotide 1270 in *gyrA* in *Shigella*, which by itself is a non-synonymous mutation, as indicated in figure A.1. However, the mutations in nucleotide 1272 negated the amino acid change, as shown by the lack of a substitution in amino acid 423 in figure A.2. The fact that synonymous mutations vastly outnumbered non-synonymous mutations was expected, as most non-synonymous mutations would likely reduce fitness. A synonymous mutation, on the other hand, can be expected to have much less effect on fitness, resulting in more mutations that remain.

The top substitution percentages were noticeably higher among the *Shigella* genomes than among the *E. coli* genomes. Since the *Shigella* genus is generally accepted to be more appropriately characterised as a part of the *E. coli* species [22], it seems reasonable that there would be more similarity among *Shigella* than the larger en-
compassing species of *E. coli*, resulting in some very high substitution percentages. If similar subgroups of *E. coli* were to be analysed separately from the species, similar results would likely emerge. However, it might also be possible that the apparent greater diversity of *E. coli* could be caused by the larger amount of genomes analysed - 4660 compared to *Shigella’s* 812.

### 5.2.1 Fluoroquinolone Resistance Mutations

Previous studies have found that the most common resistance mutation in GyrA in *E. coli* is in position S83 (especially the substitution S83L), followed by D87 [4]. This matches the results of this thesis exactly. For strains with single mutations in GyrA, S83 substitutions have been found to confer significantly higher fluoroquinolone resistance than D87 substitutions, providing a likely reason for the higher prevalence of the former [4]. Studies have also found that S83L results in higher resistance than S83A, and that D87N yields higher resistance than D87G and D87Y [4]. This provides a plausible explanation for the fact that S83L and D87N were much more common than the other mutations in the same positions in this study.

In spontaneous *in vitro* *E. coli* mutants, gyrB nucleotide substitutions have been shown to be approximately as common as gyrA nucleotide substitutions, while the latter dominated clinical isolates [4]. The substitution study showed no resistance mutations in gyrB at all, while gyrA resistance mutations were common. This could possibly be an indication of clinical isolates being overrepresented among RefSeq genomes.

In ParC, the most commonly reported resistance mutations are in position S80 (especially the substitution S80I), followed by E84 [4]. Again, this matches the results of the substitution study. Since ParC is homologous to GyrA, with S83 and D87 in GyrA corresponding to S80 and E84 in ParC [23], this similarity to the GyrA resistance mutations was expected. In previous studies, ParC and ParE mutations have not been found without GyrA mutations in *E. coli*, probably because the fluoroquinolone susceptibility of DNA gyrase needs to be reduced before topoisomerase IV mutations can affect resistance [4]. This could explain why there are fewer S80 and E84 substitutions in ParC than S83 and D87 substitutions in GyrA. It would have been interesting to check whether the GyrA mutation requirement holds true for all the *E. coli* RefSeq genomes, but time constraints prohibited it.
5. Discussion
Conclusion

I have constructed a pipeline (ARM-find) that works well for finding fluoroquinolone resistance mutations in both *E. coli* and *Shigella* genomes, including draft genomes. Its resistance mutation database is easily extensible, allowing for the identification of resistance mutations for any antibiotic in any species, provided that it is known which mutations are relevant. The analysis time required to analyse a genome is short, making ARM-find suitable to run on consumer-grade computers. Through scripted use of the pipeline, I have been able to discover that a large portion of *E. coli* and *Shigella* RefSeq genomes contain fluoroquinolone resistance mutations. The relative frequencies of those resistance mutations matched was has been previously reported, and the most common resistance mutations were the ones that lower susceptibility to fluoroquinolones the most.
7

Future Work

ARM-find functions as was intended from the start, but there are improvements that could be made. The most obvious improvement is of course the addition of more species and classes of antibiotics, to increase the usefulness of the pipeline for resistance mutation identification. This would provide the added benefit of allowing testing of situations that do not occur when just looking for fluoroquinolone resistance mutations in \textit{E. coli}. For instance, future reference sequences might be more difficult to align, due to things like segments that are also found elsewhere in the genome. Testing situations like these would allow for the optimisation of all parameter values in the pipeline, which are currently chosen in an ad hoc manner that works for fluoroquinolone in \textit{E. coli}.

Another possible feature that comes to mind is automatic species identification. Currently, the species of the input genome has to be set with the \texttt{--species} argument. The rationale for this was that the pipeline is meant to be used with assembled genomes, and it seems unlikely that the species would remain unknown after assembly. If the pipeline were to be adapted for use with raw sequencing data, automatic species identification would make much more sense, as it would be more plausible that the species would be unknown in that scenario. Allowing for input of raw sequencing data would be a big improvement for the pipeline, for several reasons. It would reduce the level of expertise required to use it, and it would probably reduce the amount of time from sequencing to resistance mutation identification, allowing for speedier determination of suitable treatment in hospitals. Reducing the time required for resistance identification is of course a big part of why this pipeline was made in the first place. However, this feature would probably require extensive reworking of the workflow. If the reads from the sequencing data are not sufficiently long, which they most likely would not be, some assembly might be required in the pipeline. That would not be an easy feature to implement, but it would certainly be useful. Another option could be to align the reads against the reference sequence, and modify reporting so that it would not produce one line for every mutation.

One feature that already exists, but could be improved upon, is handling of overlapping sequences. The most important addition would be identification of multiple gene copies. Currently, whether a mutation is covered elsewhere in the genome or not is made clear by the report through the FASTA sequence(s) and mutation prevalence columns. However, no information is provided on why the position is covered multiple times. Identification of multiple gene copies would be very useful, as a
resistance mutation in one gene copy might not mean the same thing if there are non-mutated copies, as it would have if there were only one copy of the gene. That sort of information would aid in determination of whether the analysed bacteria are clinically resistant to a certain antibiotic or not.

I would like to slightly rework the mutations database to include information for every mutation on the references that support its inclusion. This would simplify future maintenance of the database, as well as lend it more credibility. The easiest way to implement this would probably be the addition of a column of comma-separated Digital Object Identifiers (DOIs) to the articles that the resistance mutations are referenced in.

These are just some of the features that would be useful in ARM-find. Other examples include making it work on non-Linux platforms, or even creating a website where analysis could be run on servers, instead of the user’s computer. While certainly useful, those features are less likely as future additions than the ones discussed above.
Bibliography


A

*Shigella* Results
A. *Shigella* Results

**Figure A.1:** Substitution percentage for the DNA sequence of *gyrA* in *Shigella*. Red bars denote non-synonymous mutations.

**Figure A.2:** Substitution percentage for the amino acid sequence of subunit GyrA in DNA gyrase in *Shigella*. 
Figure A.3: Substitution percentage for the DNA sequence of gyrB in *Shigella*. Red bars denote non-synonymous mutations.

Figure A.4: Substitution percentage for the amino acid sequence of subunit GyrB in DNA gyrase in *Shigella*. 

III
A. Shigella Results

Figure A.5: Substitution percentage for the DNA sequence of parC in Shigella. Red bars denote non-synonymous mutations.

Figure A.6: Substitution percentage for the amino acid sequence of subunit ParC in topoisomerase IV in Shigella.
Figure A.7: Substitution percentage for the DNA sequence of parE in Shigella. Red bars denote non-synonymous mutations.

Figure A.8: Substitution percentage for the amino acid sequence of subunit ParE in topoisomerase IV in Shigella.
Figure A.9: Substitution percentage for all amino acid positions with substitutions in >3 % of genomes for GyrA in *Shigella*, with amino acid breakdown. White diagonal lines indicate fluoroquinolone resistance. Right-hand bars show percentage of genomes with unknown amino acids.

Figure A.10: Substitution percentage for all amino acid positions with substitutions in >3 % of genomes for GyrB in *Shigella*, with amino acid breakdown. No resistance mutations were detected. Right-hand bars show percentage of genomes with unknown amino acids.
Figure A.11: Substitution percentage for all amino acid positions with substitutions in >3% of genomes for ParC in *Shigella*, with amino acid breakdown. White diagonal lines indicate fluoroquinolone resistance. Right-hand bars show percentage of genomes with unknown amino acids.

Figure A.12: Substitution percentage for all amino acid positions with substitutions in >3% of genomes for ParE in *Shigella*, with amino acid breakdown. No resistance mutations were detected. Right-hand bars show percentage of genomes with unknown amino acids.
A. *Shigella* Results
Pipeline Code

# ARM-find pipeline

Overview

This pipeline takes a genomic DNA sequence in FASTA format and looks for antibiotic resistance mutations saved in its database.

Main Steps:

Iteration through every target sequence in the pipeline's database:

1. Reading reference mutations and checking whether sequence codes for a protein.
2. Sequence extraction from genome FASTA file of gene that can contain antibiotics resistance mutations.
   - BLAST alignment of reference sequence against genome.
   - Merging of BLAST hits from same FASTA sequence and same gene copy.
   - Extension of BLAST hits to cover reference sequence fully.
   - Deletion of bad BLAST hits.
   - N padding of BLAST hits that don't cover reference sequence fully.
3. Global alignment of nucleotide sequence (as well as translated sequence if it is a protein) against the reference sequence.
4. Mutation calling
5. Comparison with antibiotics resistance mutations in database.

Report creation

Dependencies:

1. NCBI BLAST
2. MAFFT
3. EMBOSS

# IMPORTS

import sys
import os.path
import shutil
import subprocess
import argparse
import random
import datetime
import errno
import re
import collections

# Main data flow

def main():
B. Pipeline Code

```python
# find directory of this script
global pipeline_dir
pipeline_dir = os.path.dirname(os.path.realpath(__file__))
# Parse arguments
parse_parameters()
# Initialise output/working directory
create_dir(args.out)

# Saving report path variables
mutations_report_path = args.out + '/report.txt'
mutations_not_covered_path = args.out + '/mutations_not_covered.txt'
targets_not_covered_path = args.out + '/targets_not_covered.txt'

# Remove old output files in output directory
remove_old_out_files([mutations_report_path, mutations_not_covered_path, targets_not_covered_path])

# Reading input file to variable
global genome_contents
with open(args.infile.name, 'r') as myfile:
    genome_contents = myfile.read()

# Read 'mutations' and 'proteins' databases
mutations_db = read_database('mutations')
targets_db = read_database('targets')

# Check if species argument is supported. Exit if not.
supported_species = get_supported_species(targets_db)
if args.species not in supported_species:
    print('ERROR: "' + args.species + '" was not recognised as a supported species. Supported species are:')
    for species in supported_species:
        print(species)
    print()
sys.exit()

# Keep only targets for the correct species
targets_db = get_targets_for_species(targets_db)

# Initiate report lists
mutation_report = []
mutation_coverage_report = []
target_ID_coverage_report = []

# Make BLAST database from genome file
BLAST_database_path = make_BLAST_db(args.infile.name, args.out + '/tmp')
L_print('Iterating through target sequences in targets database.')
# Iterate through each target ID to handle one sequence at a time
for target_ID in targets_db:
    L_print('Working with target ' + target_ID + '.format(targets_db[target_ID][1]))
    refseq_path = pipeline_dir + '/reference_sequences/' + target_ID + '.fasta'
    out_dir = args.out + '/tmp/' + target_ID
    # Saving path to reference sequence and working directory for current target ID
    refseq_path = pipeline_dir + '/reference_sequences/' + target_ID + '.fasta'
    out_dir = args.out + '/tmp/' + target_ID
    # Check that the reference sequence file is available
    if check_file(refseq_path) is False:
        print('ERROR: Path to reference sequence {}'.format(refseq_path))
        print('Error: Check integrity of targets database. Shutting down.')
        sys.exit()

    # Initiating tmp directory for target ID
    create_dir(out_dir)

# READING REFERENCE MUTATIONS
ref_mutations = read_ref_mutations(target_ID, args.species, mutations_db,

X
```
targets_db)

# Check if 'target ID' sequence is a protein.
if len(ref_mutations.prot) > 0:
    translation_required = True
else:
    translation_required = False

# SEQUENCE EXTRACTION

# BLAST alignment
BLAST_alignment_path = BLAST_align(args.infile.name, refseq_path, out_dir, BLAST_database_path)

# BLAST parsing
match_dict, query, no_of_hits = parse_BLAST_alignment(BLAST_alignment_path, refseq_path)

if no_of_hits > 0:
    # Checking for multiple hits from any one FASTA sequence and merging these if appropriate
    match_dict = merge_same_fseq_hits(match_dict, query.length)

    # Extending BLAST hits to cover reference sequence, and deleting bad hits.
    match_dict, no_of_hits = extend_BLAST_hits(match_dict, query.length)

    if no_of_hits > 0:
        # Finding overlap between BLAST hits
        nucl_overlaps = find_nucl_overlap(match_dict)

        # Saving reference sequence and (padded) hit sequences
        nucl_sequences = get_padded_BLAST_sequences(match_dict, query.name, query.sequence)

# TRANSLATION AND SEQUENCE ALIGNMENT

# Writing nucleotide sequences to file for MAFFT
extracted_sequences_path = write_fasta(nucl_sequences, out_dir + '/extracted_sequences.fasta')

# MAFFT alignment and parsing for nucleotide sequences
MAFFT_nucl_alignment_path = MAFFT_align(extracted_sequences_path, out_dir + '/MAFFT_nucl_alignment')
MAFFT_aligned_nucl_sequences = parse_MAFFT_alignment(MAFFT_nucl_alignment_path)

if translation_required is True:
    # Translation to amino acid sequence and parsing translation
    translated_path = translate(extracted_sequences_path)
    remove_translation_suffixes(translated_path)

    # MAFFT alignment and parsing for amino acid sequences
    MAFFT_prot_alignment_path = MAFFT_align(translated_path, out_dir + '/MAFFT_prot_alignment')
    MAFFT_aligned_prot_sequences = parse_MAFFT_alignment(MAFFT_prot_alignment_path)

    # Finding overlaps between the amino acid sequences
    prot_overlaps = find_protein_overlap(MAFFT_aligned_prot_sequences.ref, MAFFT_aligned_prot_sequences.hits)

    # MUTATION CALLING

    nucl_mutations, bases_covered = call_mutations(MAFFT_aligned_nucl_sequences.ref, MAFFT_aligned_nucl_sequences.hits, ref_mutations, 'nucl')

    if translation_required is True:
B. Pipeline Code

176 prot_mutations, amino_acids_covered = call_mutations(
    MAFFT_aligned_prot_sequences.ref, MAFFT_aligned_prot_sequences.hits,
    ref_mutations, "prot")

177 
178 # Compare called mutations to reference sequence mutations and append
179 # results to the mutation report
180 mutation_report = append_mutation_report(mutation_report,
    MAFFT_aligned_nucl_sequences, nucl_overlaps, nucl_mutations, ref_mutations, "nucl")
181 if translation_required is True:
182     mutation_report = append_mutation_report(mutation_report,
    MAFFT_aligned_prot_sequences, prot_overlaps, prot_mutations, ref_mutations, "prot")

183 # Creating empty sets for append_coverage_report(), in case no hits were
184 # available
185 if no_of_hits == 0:
186     bases_covered = set()
187     amino_acids_covered = set()
188
189 # MUTATION COMPARISON AND REPORT

190 # Check which bases/amino acid weren’t covered in the sequence and append to
191 # coverage reports.
192 mutation_coverage_report, target_ID_coverage_report = append_coverage_report(
    mutation_coverage_report,
    target_ID_coverage_report,
    bases_covered, ref_mutations, targets_db, target_ID, query.length, "nucl")
193 if translation_required is True:
194     mutation_coverage_report, target_ID_coverage_report = append_coverage_report(
        mutation_coverage_report,
        target_ID_coverage_report,
        amino_acids_covered, ref_mutations, targets_db, target_ID, int(query.length / 3), "prot")
195
196 # Remove target ID tmp directory when it is no longer needed. Not necessary,
197 # but keeps the tmp directory from growing in size unnecessarily.
198 if args.keep_tmp_files is False:
199     shutil.rmtree(out_dir)
200
201 L_print("Finished with target '{0}'").format(targets_db[target_ID][1]), 2)
202
203 # Remove the entire tmp directory
204 if args.keep_tmp_files is False:
205     shutil.rmtree(args.out + "/tmp")
206
207 L_print("Finished with all target sequences.")

211 # Write mutation report to file
212 header = "Target name\tInvolved antibiotics\tMutation\tNucl/prot\tResistance\tFASTA sequence(s)\tMutation prevalence\tSequence coverage"
213 write_report_file(mutation_report, header, mutations_report_path)

216 # Write mutation coverage report to file
217 header = "Target name\tInvolved antibiotics\tNot covered\tNucl/prot\tResistance"
218 write_report_file(mutation_coverage_report, header, mutations_not_covered_path)

221 # Write target ID coverage report to file
222 header = "Target name\tInvolved antibiotics"
223 write_report_file(target_ID_coverage_report, header, targets_not_covered_path)
# Exit
L_print('Done. Exiting')
sys.exit()

# Basic input and parameter handling
# Defining arguments, and processing where direct access to the global args.
# (e.g. assigning printlevels for --verbose and --quiet)
def parse_parameters():
    # Defining arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--infile', required='True', type=argparse.FileType('r'),
                        help='Input genome file to be analysed - fasta format.')
    parser.add_argument('-o', '--out', help='Output and working directory. Choose a
                        non-existing directory to remove all risk of conflicting file names.')
    parser.add_argument('-s', '--species', required='True', help='Specify the species
                        of the input genome - abbreviated name (e.g. E. coli).')
    parser.add_argument('--list_species', action='store_true', help='Lists the
                        species that are supported, and exits.')
    parser.add_argument('-l', '--logfile', nargs='?', const='standard_path', help='Directs all output (verbose) to a log file.
                        --ext_program_output', action='store_true', default=False,
                        help='Includes external program output in the pipeline's output.
                        --keep_tmp_files', action='store_true', default=False, help='Does not delete any temporary files.
                        --BLAST_perc_identity', type=float, help='Sets percent
                        identity cutoff for BLAST (0-100). Default is 95 %.
                        --report_all_coverage', action='store_true', default=False,
                        help='Includes any position that is not covered in mutations_not_covered report
                        , regardless of whether it corresponds to a possible resistance mutation or not .

    global args # Declaring global inside parse_parameters, so that both other
    # Processing arguments
    if args.list_species:
        targets_db = read_database('proteins')
        supported_species = get_supported_species(targets_db)
        print('
Supported species:
')
        for species in supported_species:
            print(species)
            print()
    sys.exit()

    if args.BLAST_perc_identity is not None:
        if args.BLAST_perc_identity <= 0 or args.BLAST_perc_identity >= 100:
            error_msg('ERROR: Specified BLAST percent identity cutoff is not a float
                        between 0 and 100.')
            sys.exit()
        else:
B. Pipeline Code

```python
args.BLAST_perc_identity = 95

if args.quiet:
    args.printlevel = 0
elif args.verbose:
    args.printlevel = 2
else:
    args.printlevel = 1

if args.out is not None:
    if args.out.endswith('/ '):
        args.out = args.out.rstrip('/**
else:
    # Creates a directory named with today's date and some random numbers if no
directory was specified, in order not to overwrite anything
    date = datetime.date.today().isoformat()
    randbits = random.getrandbits(32)
    args.out = '{0}_{1}_{2}'.format('pipeline_out', date, randbits)

if argslogfile == 'standard_path':
    argslogfile = args.out + '/log.txt'

if argslogfile is not None:
    init_logfile()

# Custom print function that takes levels 0, 1, and 2 to designate in which
# operation modes
# it will be printed or suppressed (normal, quiet, verbose).
# Writes to logfile if one is specified.
def L_print(msg, level=1):
    if argslogfile is None:
        if level <= args.printlevel:
            print(msg)
        else:
            of = open(argslogfile, 'a')
            of.write(msg + '\n')
            of.close()

# Executes shell commands, and uses L_print to either display the output, suppress
# it, or write it to the logfile.
# Note that suppressOut=True can be set if output should be excluded from L_print
# e.g. for MAFFT, whose output is all alignments)
def L_execute(cmd, level=1, header='', suppress_out=False):
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
    (out, err) = proc.communicate()
    if args.ext_program_output is True:
        # Print header if it's defined - used for external programs with printed output
        if header != '':
            L_print('\n' + '−' * 80 + '\n' + header + '\n' + '−' * 80, level)
    # For shell commands whose output needs to be saved, but not L_printed, the
    # following conditional helps
    if suppress_out is False:
        # Empty output should not show up as empty lines in the log/output
        if out.decode('ascii') != '':
            L_print(out.decode('ascii'), level)
        if err.decode('ascii') != '':
            L_print(err.decode('ascii'), level)
        if header != '':
            L_print('−' * 80 + '\n', level)
    return out

# Removes files in file_list, as well as tmp directory, in output directory.
def remove_old_out_files(file_list):
    tmp_dir = args.out + '/tmp
    if os.path.exists(tmp_dir) and os.path.isdir(tmp_dir):
        shutil.rmtree(tmp_dir)
```

XIV
for file in file_list:
    silent_remove(args.out + '/' + file)

# Initialises logfile. If file exists: exit if not an old logfile, overwrite if it is

def init_logfile():
    # Creates log file directory if it doesn't exist
    if args.logfile.endswith('/ '):
        error_msg('Specified logfile path ended with a '/'. Please specify the desired path of the logfile, not its parent directory. *)
        sys.exit()
    log_dir = args.logfile[:args.logfile.rindex('/')]
    create_dir(log_dir, suppress_print=True)

    # Checks if file exists, and stops script if it is something else than an old log file.
    logfile_header = '# ' * 80 + '

    Logfile of ARM-find

    ' + '# ' * 80 + '

    '
    if os.path.isfile(args.logfile):
        with open(args.logfile, 'r') as myfile:
            contents = myfile.read()
        if re.match(logfile_header, contents) is None:
            error_msg('Logfile error: the specified file already exists, but is not an old logfile. Exiting. *)
            sys.exit()
    if os.path.isdir(args.logfile):
        error_msg('Logfile error: the specified path is a directory. Please specify the desired path of the logfile. Exiting. *)
        sys.exit()

    # Removes old logfile and writes header to new file
    silent_remove(args.logfile) # If args.logfile is the path to an old logfile, removes it
    of = open(args.logfile, 'a')
    of.write(logfile_header)
    of.close()

    # Creates directories recursively
    def create_dir(path, suppress_print=False):
        parts = path.rstrip('/').split('/ ')
        for i in range(len(parts)):
            tmppath = '/'.join(parts[:i+1])
            # This check makes sure paths starting with '/ ' don't cause crashes.
            if tmppath != '':
                if os.path.exists(tmppath) is False:
                    if suppress_print is False:
                        L_print('Creating {0}'.format(tmppath), 2)
                    os.makedirs(tmppath)

    # File removal
    def silent_remove(filename):
        try:
            os.remove(filename)
        except OSError as e:
            if e.errno != errno.ENOENT: # errno.ENOENT = no such file or directory
                raise # re-raise exception if a different error occured

    # Checks for a file
    def check_file(filename):
        if os.path.exists(filename) is False or os.path.getsize(filename) == 0:
            L_print('(Cannot find {0}). '.format(filename), 2)
            return False
        else:
            return True
def make_BLAST_db(infile, out_dir):
    # Making a BLAST database for alignment
    L_print(f"Making a BLAST database from {infile}".format(infile), 2)
    BLAST_database_path = f"{out_dir}/BLAST_database".format(out_dir)
    L_execute(f"makeblastdb -dbtype nucl -in {infile} -out {BLAST_database_path}".format(infile, BLAST_database_path), level=2, header="makeblastdb output:"
    return BLAST_database_path

def BLAST_align(infile, ref_seq_path, out_dir, BLAST_database_path):
    L_print(f"Making a Megablast alignment of {ref_seq_path} against {infile}".format(ref_seq_path, infile), 2)
    BLAST_result_path = f"{out_dir}/megablast_alignment.asm".format(out_dir)
    L_execute(f"blastn -db {BLAST_database_path} -query {ref_seq_path} -outfmt 11 -out {BLAST_result_path} -perc_identity {args.BLAST_perc_identity}".format(BLAST_database_path, ref_seq_path, BLAST_result_path, str(args.BLAST_perc_identity)), level=2)
    L_print("BLAST alignment complete\n", 2)
    return BLAST_result_path

def parse_BLAST_alignment(BLAST_alignment, ref_seq_path):
    L_print(f"Parsing BLAST alignment file: {BLAST_alignment}".format(BLAST_alignment), 2)
    # Conversion to BLAST XML (for parsing) and pairwise (for readability) output formats
    xml_file_path = f"{os.path.dirname(BLAST_alignment)}/{os.path.basename(BLAST_alignment)[:-4]}.xml".format(os.path.dirname(BLAST_alignment), os.path.basename(BLAST_alignment)[:-4])
    readable_file_path = f"{os.path.dirname(BLAST_alignment)}/{os.path.basename(BLAST_alignment)[:-4]}_pairwise.xml".format(os.path.dirname(BLAST_alignment), os.path.basename(BLAST_alignment)[:-4])
    L_print(f"Converting BLAST archive format to xml ({xml_file_path})".format(xml_file_path), 2)
    L_execute(f"blast_formatter -archive {BLAST_alignment} -outfmt 5 -out {readable_file_path}".format(BLAST_alignment, readable_file_path), level=2)
    L_print("Retrieving alignment information from file\n", 2)
    xml_contents = myfile.read()
B. Pipeline Code

# Reading the contents of the reference sequence file
with open(ref_seq_path, 'r') as myfile:
    ref_seq_contents = myfile.read()

# Number of alignments made
no_of_hits = xml_contents.count('<Hsp_num>')</n
# Name of query sequence in fasta file (after >).
query_name = re.search(r'<BlastOutput_query-def>(.+)</BlastOutput_query-def>', xml_contents).group(1)

# Replace any XML codes for characters with the actual characters.
query_name = replace_XML_codes(query_name)

# Sequence name correction (lines starting with > in fasta files). BLAST XML output format removes all but
# one space whenever there are two or more consecutive spaces in sequence names, but only for the query sequence.
query_name = correct_BLAST_name_spaces(query_name, ref_seq_contents)

query_seq = get_fseq(query_name, ref_seq_contents).upper()
query_length = len(query_seq)

Query = collections.namedtuple('Query', 'name, sequence, length')
query = Query(query_name, query_seq, query_length)

# The function has to be exited if no hits are found, with the same kind of data
# structure returned as if it had been successful.
# Further handling of this case in main()
if no_of_hits == 0:
    L_print("WARNING! No BLAST hits were found for {0}. Mutation calling not possible. " format(query_name))
    return None, query, no_of_hits

# Create dictionary where each key will be a property of the BLAST alignment (e.g.
# sequence length).
# The values will be lists, where the indices correspond to BLAST hits. For
# example, with 2 hits, one would get
# the sequence length of the second hit by accessing the list belonging to the
# key for sequence length, at the second index.
match_dict = {}

# Save all relevant attributes of the hits to the dictionary.
match_dict['db_strand'] = re.findall(r'<Hsp_hit-frame>(.+)</Hsp_hit-frame>', xml_contents)
match_dict['query_strand'] = re.findall(r'<Hsp_query-frame>(.+)</Hsp_query-frame>', xml_contents)
match_dict['db_hit_seq'] = re.findall(r'<Hsp_hseq>(.+)</Hsp_hseq>', xml_contents)
match_dict['query_hit_seq'] = re.findall(r'<Hsp_qseq>(.+)</Hsp_qseq>', xml_contents)
match_dict['db_hit_from'] = re.findall(r'<Hsp_hit-from>(.+)</Hsp_hit-from>', xml_contents)
match_dict['db_hit_to'] = re.findall(r'<Hsp_hit-to>(.+)</Hsp_hit-to>', xml_contents)
match_dict['query_hit_from'] = re.findall(r'<Hsp_query-from>(.+)</Hsp_query-from>', xml_contents)
match_dict['query_hit_to'] = re.findall(r'<Hsp_query-to>(.+)</Hsp_query-to>', xml_contents)

# As several hits can be in one FASTA sequence, and all are within one <Hit/></Hit>
# each FASTA sequence name has
# to be counted the right amount of times.
matching_fseqs = re.findall(r'<Hit>(.+?</Hit>', xml_contents, re.DOTALL)
match_dict['fseq'] = []
for fseq in matching_fseqs:
    fseq_name = re.search(r'<Hit_def>(.+)</Hit_def>', fseq).group(1)
    hits_in_fseq = fseq.count('<Hsp>')</n
        for i in range(hits_in_fseq):
            match_dict['fseq'].append(replace_XML_codes(fseq_name))

XVII
### B. Pipeline Code

```python
# Type conversions
match_dict["db_hit_from"] = [int(x) for x in match_dict["db_hit_from"]]
match_dict["db_hit_to"] = [int(x) for x in match_dict["db_hit_to"]]
match_dict["query_hit_from"] = [int(x) for x in match_dict["query_hit_from"]]
match_dict["query_hit_to"] = [int(x) for x in match_dict["query_hit_to"]]

# Create match_dict entries for query/db_hit_from/to + extension, and initialise
lists in them.
match_dict["query_hit_from_extended"] = [None] * no_of_hits
match_dict["query_hit_to_extended"] = [None] * no_of_hits
match_dict["db_hit_from_extended"] = [None] * no_of_hits
match_dict["db_hit_to_extended"] = [None] * no_of_hits

# Order hits by query_hit_from to get later reporting in a sensible order from
the perspective of the reference sequence
sorted_indices = sorted(range(no_of_hits), key=lambda k: match_dict['query_hit_from'][k])
for key in match_dict:
    match_dict[key] = [match_dict[key][i] for i in sorted_indices]
return match_dict, query, no_of_hits

# Sequence name correction (lines starting with > in FASTA files). BLAST XML output
# format removes all but one space whenever there are two or more consecutive spaces in
sequence names.
def correct_BLAST_name_spaces(sequence_name, fasta_file_contents):
    pattern = "\s+"
    for char in sequence_name:
        if char == " ":
            pattern += r"\s+";
        else:
            pattern += re.escape(char)
    sequence_name = re.search(pattern, fasta_file_contents).group(0)
    return sequence_name

def replace_XML_codes(string):
    codes = {"&quot;": "", "&amp;": ", "&apos;": ", "&lt;": ", "&gt;": ""}
    for code, char in codes.items():
        string = string.replace(code, char)
    return string

# Replaces any XML codes for characters with the actual characters.
def replace_XML_codes(string):
    codes = {"&quot;": ", "&amp;": ", "&apos;": ", "&lt;": ", "&gt;": ""}
    for code, char in codes.items():
        string = string.replace(code, char)
    return string

# Merges non-overlapping BLAST hits that are on the same FASTA sequence, and are
# not different copies of a gene.
def merge_same_fseq_hits(match_dict, query_length):
    multiple_hits_in_fseq = False
    no_of_hits = len(match_dict['fseq'])
    if no_of_hits > 1:
        L_print("Checking for multiple hits from any one genomic FASTA sequence.", 2)
        for fseq, click_list in query_length:
            fseq = match_dict['fseq'][i]
            count = match_dict['fseq'].count(fseq)
            if count > 1:
                multiple_fseq_hit_indices = {}
                for i in range(no_of_hits):
                    if fseq not in multiple_fseq_hit_indices:
                        multiple_fseq_hit_indices[fseq] = [i]
                else:
                    multiple_fseq_hit_indices[fseq].append(i)
```

XVIII
if len(multimatch_fseq) > 0:
    L_print('WARNING: multiple matches from one genomic FASTA sequence found. ')
else:
    L_print('No genomic FASTA sequence gave rise to more than one hit sequence. ')

for fseq, count in multimatch_fseq.items():
    L_print('WARNING: genomic FASTA sequence named {0} gave rise to {1}
alignments. '.format(fseq, count))

Looking for hits on the same FASTA sequence that should be merged.
This solution assumes no overlap and ascending order of query_hit_from. This
should be true, as we’ve sorted above.
# and BLAST should not find multiple matches that overlap in one FASTA sequence.
# Initialise list for remembering how many bases to add if hits are merged
match_dict['merged_hits_extra_bases'] = [0] * no_of_hits
# Initialise list for remembering how many bases to add if hits are NOT merged.
match_dict['no_merge_extra_bases'] = [0] * no_of_hits
# Initialise list for remembering which hits are involved in merges.
match_dict['merged_hit'] = [False] * no_of_hits

if multiple_hits_in_fseq is True:
    hits_to_merge = []
    for fseq, indices in multimatch_fseq_hit_indices.items():
        L_print('Checking if multiple hits from genomic FASTA sequence '{0}' can be
merged. '.format(fseq), 2)

        # Indices must be sorted by strand, and then by db_hit_from (ascending for
strand 1, descending for strand −1).
        # This is in order to make sure that hits that are next to each other in the
genome are next to each other here.
        # If there are multiple copies of a gene on one FASTA sequence, only using
query_hit_from sorting could cause problems.
        indices_by_strand = {1: [], −1: []}
        for i in indices:
            if match_dict['db_strand'][i] == '1':
                indices_by_strand[1].append(i)
            else:
                indices_by_strand[−1].append(i)

        indices_by_strand[1].sort(key=lambda k: match_dict['db_hit_from'][k])
        indices_by_strand[−1].sort(key=lambda k: match_dict['db_hit_from'][k],
reverse=True)

        # i can’t be the last index, since we’re comparing with i + 1.
        for i in range(len(indices) − 1):
            # Distance between the matching query sequence parts of the two hits
            query_hit_from = match_dict['query_hit_from'][indices[i]], match_dict['
query_hit_from'][indices[i + 1]]
            query_hit_to = match_dict['query_hit_to'][indices[i]], match_dict['
query_hit_to'][indices[i + 1]]

            query_distance = (max(query_hit_from) − min(query_hit_to) − 1)

        # Distance between the two hits on the FASTA sequence
        db_hit_from = match_dict['db_hit_from'][indices[i]], match_dict['
db_hit_from'][indices[i + 1]]
        db_hit_to = match_dict['db_hit_to'][indices[i]], match_dict['
db_hit_to'][indices[i + 1]]

        # Hit distance calculation depends on which strand the hits are on.
        if match_dict['db_strand'][i] == '1':
            hit_distance = max(db_hit_from) − min(db_hit_to) − 1
        else:
            hit_distance = max(db_hit_to) − min(db_hit_from) − 1

        # For hit overlap (on FASTA sequence) check
B. Pipeline Code

```python
overlap = range(max(match_dict['db_hit_from'][indices[i]], match_dict['db_hit_from'][indices[i + 1]]),
               min(match_dict['db_hit_to'][indices[i]], match_dict['db_hit_to'][indices[i + 1]] + 1))

# If the query distance is 0, or close to 0, we have an insertion, and the
# hits should be merged.
# If the query distance isn’t 0, but the difference between the hit and
# query distances is small (some room for indels),
# the we have a mismatching region, and the hits should still be merged.
# Also check that the matches are on the same strand—though problems with
# this are unlikely.
if (query_distance < 100 or abs(hit_distance - query_distance) < 100) and
   # Make sure hits are on the same strand
   match_dict['db_strand'][indices[i + 1]] == match_dict['db_strand'][indices[i]]
   # Don’t merge hits that are from different copies of a gene
   hit_distance <= query_length and
   # Checks to make sure that the hits don’t overlap in the genome file,
   # as this could cause problems. I don’t think they ever will, but I’m not 100 %
   # sure.
   len(overlap) == 0):

   # If there are multiple copies of a gene on the same FASTA sequence, it
   # is possible to erroneously try to merge the end of one copy with the start
   # of another. This check prevents that, by making sure that the order of
   # the hits in the genome matches the order in the query.
   hit_from_distances = query_hit_from[1] - query_hit_from[0], db_hit_from[1] - db_hit_from[0]
   if (not min(hit_from_distances) < 0 < max(hit_from_distances) and
        # If db_strand is 1, check that the sign of the differences between
        # the starts of the hits is the same (or that either/differences is 0)
        match_dict['db_strand'][indices[i]] == '1' or
        # If db_strand is -1 check that the sign of the differences between
        # the starts of the hits is different (or that either/differences is 0)
        min(hit_from_distances) <= 0 <= max(hit_from_distances) and
        match_dict['db_strand'][indices[i]] == '-1')):

      # Get FASTA sequence sequence to check whether interval between hits
      # contains too many N’s for merging
      fseq_sequence = get_fseq(fseq, genome_contents)
      # Get range of interval between hits, regardless of whether the strand
      # is '1' or '-1'
      space_from, space_to = sorted(db_hit_from + db_hit_to)[1:1]
      # Accept merge if amount of N’s in interval between hits is below 30 %.
      if fseq_sequence[space_from-1:space_to].upper().count('N') < (space_to + 1 - space_from) * 0.3:
        # Append adjacent hits that are to be merged, and remember that they’re
        # involved in merging.
        hits_to_merge.append({'indices': [indices[i], indices[i + 1]], 'insertion_size': hit_distance - query_distance})
        for index in [indices[i], indices[i + 1]]:
          match_dict['merged_hit'][index] = True
        # If two adjacent hits are not merged due to N content problems, but
        # there is an indel between them, then that needs to be compensated for during
        # hit extension for the hit with lower query_hit_from.
        elif:
          match_dict['no_merge_extra_bases'][min(indices[i], indices[i + 1])] = hit_distance - query_distance

      # If there are hits to merge, compare adjacent sets of hits to merge with each
      # other. If all hits should
      # be merged, add the last hit to the first set, and add the additional hit
distance. The second set is then disregarded.
      if hits_to_merge != []:
```

XX
B. Pipeline Code

```python
# Since info on each merge is collected in one list, other lists of the same
merge are disregarded.
skip_index = set()
# indices in hits_to_merge that will end up containing information on merging
merge_info_indices = set()
# indices in hits_to_merge that will end up containing information on merging
for i in range(len(hits_to_merge)):
    if i not in skip_index:
        merge_info_indices.add(i)
if len(hits_to_merge) > 1:
    for j in range(i + 1, len(hits_to_merge)):
        if hits_to_merge[i]['indices'][-1] == hits_to_merge[j]['indices'][0]:
            hits_to_merge[i]['indices'].append(hits_to_merge[j]['indices'][1])
            hits_to_merge[i]['insertion_size'] += hits_to_merge[j]['insertion_size']
            skip_index.add(j)

# Initiate list with indices to delete across all merges
all_indices_to_delete = []
# Going through sets of hits to merge
for i in merge_info_indices:
    # Saving indices of hits that should be deleted for current merge
    indices_to_delete = hits_to_merge[i]['indices'][:]
    # Saving the extra bases necessary for the hit that will be extended to
    # cover the others
    match_dict['merged_hits_extra_bases'][hits_to_merge[i]['indices'][-1]] =
    hits_to_merge[i]['insertion_size']

    L_print('{0} hits (hit numbers: {1}) were merged into one. '.format(len(indices_to_delete) + 1, ', '.join([str(hits_to_merge[i]['indices'][j] + 1) for j in range(len(hits_to_merge[i]['indices']))])), 2)

    # The hits that are to be deleted can contain indels, which will change how
    long the remaining sequence should be extended - this is taken care of here.
    for j in indices_to_delete:
        indel_extra = abs(match_dict['db_hit_to'][j] - match_dict['db_hit_from'][j]) -
        abs(match_dict['query_hit_to'][j] - match_dict['query_hit_from'][j])
        match_dict['merged_hits_extra_bases'][hits_to_merge[i]['indices'][-1]] +=
        indel_extra

        all_indices_to_delete += indices_to_delete

    # Delete hits in reverse order, so that the things that should be deleted don
    # 't change index in the loop
    for j in all_indices_to_delete[::-1]:
        no_of_hits -= 1
        for key in match_dict:
            del match_dict[key][j]
else:
    L_print('No hits from genomic FASTA sequence {0} could be merged. '.format(fseq), 2)

return match_dict

# Extends BLAST hits to cover as much of query sequence as possible
```
B. Pipeline Code

```python
for j in range(sorted_db_hits[0], sorted_db_hits[1] + 1):
    fseq_coverage[match_dict['fseq'][i]].add(j - 1)

no_of_hits = len(match_dict['fseq'])
for i in range(no_of_hits):
    L_print('Working with hit #{0} found on genomic FASTA sequence named: {1}'.format(i + 1, match_dict['fseq'][i]), 2)
    # Save number of extra bases needed from FASTA sequence (upstream and downstream) to cover as much of the query sequence as possible.
    # Upstream and downstream are in the perspective of the query sequence. 'merged_hits_extra_bases' are added upstream in case
    # several hits from one FASTA sequence are to be merged (the distance between
    # these hits must be taken into account).
    # 'no_merge_extra_bases' are added downstream in case two adjacent hits weren't
    # merged due to too many N's, to compensate for indels.
    upstream_extra = match_dict['query_hit_from'][i] - 1 + match_dict['merged_hits_extra_bases'][i]
    downstream_extra = query_length - match_dict['query_hit_to'][i] + match_dict['no_merge_extra_bases'][i]
    # Initialise extension counters to keep track of how much a match sequence has
    # been extended.
    # Necessary in order to keep track of what query index corresponds to the ends
    # of a match.
    added_upstream = 0
    added_downstream = 0

    # If the FASTA sequence doesn’t span the entire query:
    if upstream_extra > 0 or downstream_extra > 0:
        # Save DNA sequence of the matched FASTA sequence
        fseq_seq = get_fseq(match_dict['fseq'][i], genome_contents)
        if upstream_extra > 0:
            # If BLAST hit is on the strand presented in the FASTA sequence:
            if match_dict['db_strand'][i] == '1':
                # Save indices of needed upstream bases in order of proximity to hit
                indices = range(match_dict['db_hit_from'][i] - 2, match_dict['db_hit_from'][i] - 1 - upstream_extra)
                for j in indices:
                    # If index is in range of FASTA sequence string, add base at that index
to beginning of hit sequence
                    if j >= 0:
                        # Check that the base isn't already part of another hit on the FASTA
                        # sequence to avoid false overlap. Stop loop if it is.
                        if j in fseq_coverage[match_dict['fseq'][i]]:
                            break
                        else:
                            match_dict['db_hit_seq'][i] = fseq_seq[j] + match_dict['db_hit_seq'][i]
                            added_upstream += 1
                            fseq_coverage[match_dict['fseq'][i]].add(j)
                # Else (if BLAST hit is on the complementary strand to the FASTA sequence):
                else:
                    # Save indices of needed upstream bases in order of proximity to hit
                    # sequence
                    indices = range(match_dict['db_hit_from'][i] + upstream_extra, match_dict['db_hit_from'][i] + 1)
                    for j in indices:
                        # If index is in range of FASTA sequence string, add complement of base
                        # at that index to beginning of hit sequence.
                        # Not reverse complement, since adding bases one by one in the order in
                        # indices already takes care of the order.
                        if j < len(fseq_seq):
                            # Check that the base isn’t already part of another hit on the FASTA
                            # sequence to avoid false overlap. Stop loop if it is.
                            if j in fseq_coverage[match_dict['fseq'][i]]:
                                break
```

XXII
else:
    fseq_coverage[match_dict[‘fseq’][i]].add(j)
    added_downstream += 1

if downstream_extra > 0:
    # If BLAST hit is on the strand presented in the FASTA sequence:
    if match_dict[‘db_strand’][i] == ‘1’:
        # Save indices of needed downstream bases in order of proximity to hit sequence
        indices = range(match_dict[‘db_hit_to’][i] + downstream_extra)
        for j in indices:
            # If index is in range of FASTA sequence string, add base at that index to end of hit sequence.
            if j < len(fseq_seq):
                break
            else:
                match_dict[‘db_hit_seq’][i] += fseq_seq[j]
                added_downstream += 1
                fseq_coverage[match_dict[‘fseq’][i]].add(j)

    # Else (if BLAST hit is on the complementary strand to the FASTA sequence):
    else:
        indices = range(match_dict[‘db_hit_to’][i] - downstream_extra - 2, -1)
        for j in indices:
            # If index is in range of FASTA sequence string, add complement of base at that index to end of hit sequence.
            if j >= 0:
                break
            else:
                match_dict[‘db_hit_seq’][i] += complement(fseq_seq[j])
                added_downstream += 1
                fseq_coverage[match_dict[‘fseq’][i]].add(j)

message = “Added {0} bases upstream of hit sequence, and {1} bases downstream of hit sequence.”
L_print(message.format(added_upstream, added_downstream), 2)

# Save the range of the reference sequence covered by the hit sequence after extension
# ‘merged_hits_extra_bases’ only applies upstream, since only the highest query_hit_from hit is kept during merging.
match_dict[‘query_hit_from_extended’][i] = match_dict[‘query_hit_from’][i] - added_upstream + match_dict[‘merged_hits_extra_bases’][i]

# ‘no_merge_extra_bases’ only applies downstream. If hits aren’t merged, extra (or less) extension is handled downstream, to avoid frame shift for a whole hit in case of unknown regions (many N’s).
match_dict[‘query_hit_to_extended’][i] = match_dict[‘query_hit_to’][i] + added_downstream - match_dict[‘no_merge_extra_bases’][i]

# Save the range of the genome sequence covered by the hit sequence after extension
if match_dict[‘db_strand’][i] == ‘1’:
    match_dict[‘db_hit_from_extended’][i] = match_dict[‘db_hit_from’][i] - added_upstream
    match_dict[‘db_hit_to_extended’][i] = match_dict[‘db_hit_to’][i] + added_downstream

else:
    match_dict[‘db_hit_seq’][i] = complement(fseq_seq[j]) + match_dict[‘db_hit_seq’][i]
    added_upstream += 1
    fseq_coverage[match_dict[‘fseq’][i]].add(j)
B. Pipeline Code

```python
else:
    match_dict['db_hit_from_extended'][i] = match_dict['db_hit_from'][i] + added_upstream
    match_dict['db_hit_to_extended'][i] = match_dict['db_hit_to'][i] - added_downstream

# Remove lone hits that have to be extended too far.
match_dict, no_of_hits = remove_lone_short_hits(match_dict, query_length, fseq_coverage)

return match_dict, no_of_hits

# Remove lone hits that have to be extended too far, unless part of a merge.
def remove_lone_short_hits(match_dict, query_length, fseq_coverage):
    L_print("Checking if any BLAST hits need to be deleted.", 2)
    no_of_hits = len(match_dict['fseq'])
    indices_to_delete = []

    for i in range(no_of_hits):
        L_print("Working with hit #{0}, found on genomic FASTA sequence named: {1}".format(i + 1, match_dict['fseq'][i]), 2)
        # Don't delete merged hits.
        if match_dict['merged_hit'][i] is True:
            continue
        # Don't delete if hit extends up to another hit. Min/max solves strand issues.
        # Note that 'db_hit_to_extended' contains a base number (first number is 1),
        # while fseq_coverage contains indices (first index is 0).
        # OBS! This is problematic if the short incorrect hit is close to the actual
gene. It might extend up to a base that's already added and be safe from
deletion.
        if min(match_dict['db_hit_from_extended'][i], match_dict['db_hit_to_extended'][i] - 2) in fseq_coverage[match_dict['fseq'][i]]:
            continue
        if max(match_dict['db_hit_from_extended'][i], match_dict['db_hit_to_extended'][i]) in fseq_coverage[match_dict['fseq'][i]]:
            continue
        # Delete hits that did not meet the criteria above, and that were extended too
        # far.
        extension = abs(match_dict['query_hit_from_extended'][i] - match_dict['query_hit_from'][i]) + abs(match_dict['query_hit_to_extended'][i] - match_dict['query_hit_to'][i])
        if extension / query_length > 0.2:
            L_print("Hit #{0} was deleted due to long extension (likely erroneous hit).", format(i + 1), 2)
            indices_to_delete.append(i)

    # Delete hits in reverse order, so that the things that should be deleted don't
    # change index in the loop
    for i in indices_to_delete[::-1]:
        no_of_hits -= 1
        for key in match_dict:
            del match_dict[key][i]

        if len(indices_to_delete) == 0:
            L_print("No hits were deleted.", 2)
        return match_dict, no_of_hits

    # Return overlapping ranges in nucleotide sequence
    def find_nucl_overlap(match_dict):
        # No overlap until it's found
        overlap_exists = False
        no_of_hits = len(match_dict['fseq'])
        # Initialise list of lists for overlap info
        overlaps = [[] for i in range(no_of_hits)]
```

XXIV
# Check for several hits from one FASTA sequence, and hit sequence overlap, if
# more than one BLAST hit was found.
if no_of_hits > 1:
    L_print("Checking for overlap between hit sequences. ", 2)
    # Check for overlap between hit sequences
    for i in range(no_of_hits):
        for j in range(no_of_hits):
            # Disregard overlap with self
            if j != i:
                # Range object of overlap sequence. Length 0 if no overlap
                overlap = range(max(match_dict['query_hit_from_extended'][i], match_dict['query_hit_to_extended'][j]),
                                 min(match_dict['query_hit_to_extended'][i], match_dict['query_hit_from_extended'][j]) + 1)
                # If there is overlap:
                if len(overlap) > 0:
                    overlap_exists = True
                    # Save info on overlap for hit sequence i: overlapping sequence index (j), and query sequence numbers for the overlap (from, to)
                    overlaps[i].append([j, min(overlap), max(overlap)])
            if overlap_exists is True:
                L_print("WARNING: Overlap between sequences found. ", 2)
                for i in range(no_of_hits):
                    if overlaps[i] != []:
                        for j in range(len(overlaps[i])):
                            if overlaps[i][j] > i:
                                L_print("Genomic FASTA sequences '{0}' and '{1}' overlap in the reference sequence region of {2}−{3}.", format(match_dict['fseq'][i], match_dict['fseq'][overlaps[i][j][0]], overlaps[i][j][1], overlaps[i][j][2]), 2)
                        else:
                            L_print("No overlap between sequences was found.", 2)
    return overlaps

# Return reference and hit sequences, padding the latter if necessary
def get_padded_BLAST_sequences(match_dict, query_name, query_seq):
    # Adding sequences to a list of lists for passing to write_fasta. Starting here
    # and later appending padded_hit_seqs.
    seq_list = [[query_name, query_seq]]
    query_length = len(query_seq)
    no_of_hits = len(match_dict['fseq'])

    # Add appropriate n padding to the hits and save them to seq_list for return statement
    n_seq = "n" * query_length
    for i in range(no_of_hits):
        padded_hit_seq = n_seq + match_dict['query_hit_from_extended'][i] + n_seq + match_dict['query_hit_to_extended'][i] + n_seq
        # Gap hyphens need to be removed.
        padded_hit_seq = padded_hit_seq.replace("−", "")
        seq_list.append([match_dict['fseq'][i], padded_hit_seq.upper()])
    L_print("BLAST parsing of '{0} ' ' complete

    return seq_list

# ############################################################################
# Alignment
# ############################################################################
# Does a MAFFT alignment of sequences in infile and returns a string with the

XXV
B. Pipeline Code

```python
def MAFFT_align(infile, out_path):
    L_print("Making a MAFFT alignment of sequences in {0}\'.format(infile), 2)
    # Without --anysymbol, MAFFT removes * (stop codons) before alignment. Note that
    # this results in case-sensitivity.
    L_execute("ginsi --anysymbol {0} > {1}".format(infile, out_path), level=2, header=
    "MAFFT output:")
    L_print("MAFFT alignment complete", 2)
    return out_path

# Parse MAFFT alignment
def parse_MAFFT_alignment(MAFFT_alignment_path):
    L_print("Parsing MAFFT alignment file : {0}".format(MAFFT_alignment_path), 2)
    with open(MAFFT_alignment_path) as myfile:
        MAFFT_alignment = myfile.read()
        # Go through MAFFT alignment and save sequences and their names as lists in the
        # hit_seqs list.
        hits = [
            MAFFT_alignment.splitlines()
        for line in lines:
            if line.startswith(">"):  # no_of_hits]
                hits.append([line[1:], ''])
            else:
                hits[-1][1] += ".join(line.split())
        # Save the reference sequence list in hit_seqs to ref_seq, and delete it from
        # hit_seqs.
        ref = hits[0]
        del hits[0]
        AlignedSequences = collections.namedtuple('AlignedSequences', 'ref, hits')
        aligned_sequences = AlignedSequences(ref, hits)
        return aligned_sequences

# Returns positions of overlaps in terms of reference base numbers for all protein
# hit sequences.
def find_protein_overlap(ref, hits):
    no_of_hits = len(hits)
    ref_seq = ref[1]
    hit_seqs = [
        hits[i][1] for i in range(no_of_hits)
    prot_start_index = [None] * no_of_hits
    prot_end_index = [None] * no_of_hits
    overlaps = [[] for i in range(no_of_hits)]
    if no_of_hits > 1:
        # Saving overlap with other hit sequences for each hit sequence.
        for i in range(no_of_hits):
            for j in range(no_of_hits):
                if j != i:
                    # Range object of overlap sequence. Length 0 if no overlap
```

XXVI
B. Pipeline Code

def pipeline_code:
    overlap = range(max(prot_start_index[i], prot_start_index[j]), min(prot_end_index[i], prot_end_index[j]) + 1)
    if len(overlap) > 0:
        # Reporting amino acid numbers, not indices, so +1.
        overlaps[i].append([j, min(overlap) + 1, max(overlap) + 1])
    return overlaps

# Mutation calling, comparison, and report

def call_mutations(ref, hits, ref_mutations, sequence_type):
    if sequence_type == "nucl":
        message = "DNA"
        ref_mutations = ref_mutations.nucl
    elif sequence_type == "prot":
        message = "protein"
        ref_mutations = ref_mutations.prot

    L_print('Calling mutations in {0} sequence'.format(message), 2)

    # Get positions of potential resistance mutations. Necessary for cases where the reference sequence has a resistance mutation.
    # since only called mutations are checked for resistance.

    ref_mutation_positions = set()
    if ref_mutations != []:
        for ref_mutation in ref_mutations:
            ref_mutation_positions.add(int(ref_mutation[0]))

    no_of_hits = len(hits)

    subs = [collections.OrderedDict() for i in range(no_of_hits)]
    ins = [collections.OrderedDict() for i in range(no_of_hits)]
    dels = [collections.OrderedDict() for i in range(no_of_hits)]

    ref_numbers_covered = set()

    ref_seq = ref[1]
    hit_seqs = [hits[i][1] for i in range(no_of_hits)]

    # Saving mutations

    for hit in range(no_of_hits):
        # Initialise base number counter for reference sequence - for handling gaps.
        # The first base will be numbered 1.

        ref_number = 0
        for i in range(len(ref_seq)):
            # To report mutation positions in relation to reference sequence, gaps have to be handled
            if ref_seq[i] != "-":
                ref_number += 1

            # If base is the same in hit and ref, remember that this base is covered
            if ref_seq[i] == hit_seqs[hit][i]:
                ref_numbers_covered.add(int(ref_mutation[0]))

            # Check if ref_number is a resistance-associated position. If so, add like normal sub.
            if ref_number in ref_mutation_positions:
                subs[hit][ref_number] = [ref_seq[i].upper(), hit_seqs[hit][i].upper(), i]
            # Else (if bases differ), check how
            else:
                # If hit sequence base is a gap, save reference base number, reference base, and index in the hit
                if hit_seqs[hit][i] == "-":
                    dels[hit][ref_number] = [ref_seq[i], i]
B. Pipeline Code

```python
ref_numbers_covered.add(ref_number)

# If reference sequence base is a gap, save reference base number before
gap, inserted hit sequence base, and index in the hit.
elif ref_seq[i] == '-':
    if ref_number in ins[hit]:
        ins[hit][ref_number][1] += hit_seqs[hit][i]
    else:
        ins[hit][ref_number] = [ref_seq[i-1], hit_seqs[hit][i], i]

# Only normal substitution, save reference base number, reference sequence
# base, hit sequence base, and index in the hit.
# Normal substitution, and unknown base (N) remain, so check for 'not
# N'.
elif ((sequence_type == 'nucl' and hit_seqs[hit][i].upper() != 'N') or
    (sequence_type == 'prot' and hit_seqs[hit][i].upper() != 'X')):
    subs[hit][ref_number] = [ref_seq[i].upper(), hit_seqs[hit][i].upper(), i]
    ref_numbers_covered.add(ref_number)

CalledMutations = collections.namedtuple('CalledMutations', 'subs, ins, dels')
called_mutations = CalledMutations(subs, ins, dels)
return called_mutations, ref_numbers_covered

# Appends called mutations to report, while making a comparison with known
resistance mutations.
def append_mutation_report(mutation_report, MAFFT_alignment, overlaps,
    hit_mutations, ref_mutations, seq_type):
    if seq_type == 'nucl':
        message = 'DNA'
    elif seq_type == 'prot':
        message = ' protein '
    L_print('Comparing called mutations in {0} sequence to resistance mutations ,
        and adding to report .'.format(message), 2)

    antibiotics = ' , '.join(get_involved_antibiotics(ref_mutations))

    # Set variables for nucleotid/amino acid so the same name can be used.
    if seq_type == 'nucl':
        ref_resistance_mutations = ref_mutations.nucl
        unknown = 'N'
    elif seq_type == 'prot':
        ref_resistance_mutations = ref_mutations.prot
        unknown = 'X'

    ref = MAFFT_alignment.ref
    hits = MAFFT_alignment.hits

    no_of_hits = len(hits)
    report_list = []
    for i in range(no_of_hits):
        # Loop through hit sequences
        for mutation_type in ['subs', 'dels', 'ins']:
            # Loop through substitutions for the given hit sequence
            for ref_number, mutation in getattr(hit_mutations, mutation_type)[i].items():
                # No resistance unless found
                resistance = ''
                # Loop through resistance mutations
                for ref_mutation in ref_resistance_mutations:
                    if ((mutation_type == 'subs' and int(ref_mutation[0]) == ref_number and
                        ref_mutation[2] == mutation[1])
                        or
                        (mutation_type == 'dels' and int(ref_mutation[0]) == ref_number and
                        ref_mutation[2] == '-')):
                        if resistance != '':
                            resistance += ', '
                        resistance += ref_mutation[3]
```
B. Pipeline Code

# If reference base/aa is the same as mutated base/aa, and that mutation is not associated with resistance, skip adding it to report.
# mutation[0] == mutation[1] occurs because all resistance-associated positions are called as mutations, in case the reference sequence has a resistance mutation.
# This check removes all cases like R45R, except if that confers resistance
if mutation[0] == mutation[1] and resistance == "":
    continue

# Initialize variable for counting overlapping FASTA sequences at this mutation, and how many of them have the same mutation.
overlapping_fseqs = 0
same_mutation = 0

# Initialize variable containing the FASTA sequences a mutation was found on
fseqs = " '{0}'" . format( hits[i][0][0])
for overlap in overlaps[i]:
    # Have to check that there are overlaps. If there aren’t, overlaps will contain *no_of_hits* empty lists
    if len(overlap) > 0:
        # Check if base/aa at ref_number overlaps with any other FASTA sequence
        if overlap[1] <= ref_number and overlap[2] >= ref_number:
            overlapping_fseqs += 1
        # Appends the FASTA sequence name which overlaps the current one
        fseqs += " , " + " '{0}'" . format( hits[overlap[0]][0][0])
        # Check if the substituted/deleted base/aa in FASTA sequence is the same as the base/aa in the
        # overlapping one. (overlap[0] is the index of an overlapping FASTA sequence in hits, 1 refers to the sequence (not name),
        # and mutation[-1] is the index of the base/aa in that sequence). If same mutation, increment variable.
        if ((mutation_type == 'subs' and hits[overlap[0]][1][mutation[−1]] == mutation[1]) or
            (mutation_type == 'dels' and hits[overlap[0]][1][mutation[−1]] == ' −')):
            same_mutation += 1
        # Check if base/aa at ref_number overlaps with any other FASTA sequence
        elif mutation_type == 'ins':
            # Same check as above, but taking potentially longer-than-one sequence into account.
            if hits[overlap[0]][1][mutation[−1]:mutation[−1]+len(mutation[1])+1] == mutation[1]:
                same_mutation += 1

# Adjusting report depending on type of mutation
if mutation_type == 'subs':
    mutation_text = mutation[0] + str(ref_number) + mutation[1]
elif mutation_type == 'dels':
    mutation_text = mutation[0] + str(ref_number) + ' −'
elif mutation_type == 'ins':
    mutation_text = mutation[0] + str(ref_number) + mutation[0] + mutation[1]

# Only write mutation prevalence column value if there are overlapping
FASTA sequences
if overlapping_fseqs > 0:
    mutation_prevalence = (same_mutation + 1) / (overlapping_fseqs + 1)
else:
    mutation_prevalence = " "

# Info on how well the hit covers the reference sequence
if hits[i][1][0] != unknown and hits[i][1][−1] != unknown:
    coverage = 'Full'
elif hits[i][1][0] != unknown:
    coverage = 'Partial w/ start'
elif hits[i][1][0] != unknown:
    coverage = 'Partial w/o start'

# Append an item to internal report list with information on the mutation.

XXIX
B. Pipeline Code

```python
report_list_entry = [ ref[0], antibiotics, mutation_text, seq_type,
    resistance, fseqs, mutation_prevalence, coverage]
report_list.append([report_list_entry, ref_number])

# Sort internal report list by ref_number, instead of by ref_number for subs, ins,
# and dels separately
sorted_indices = sorted(range(len(report_list)), key=lambda k: report_list[k][1])
report_list = [report_list[i] for i in sorted_indices]

# Append internal report list (nucl/prot) to class instance
for item in report_list:
mutation_report.append(item[0])
return mutation_report

def append_coverage_report(mutation_coverage_report, target_ID_coverage_report,
    positions_covered, ref_mutations, targets_db, target_ID, seq_length, seq_type):
    L_print("Checking if any resistance mutations weren’t covered in hit sequences,
    and adding to coverage report", 2)
    antibiotics = ', '.join(get_involved_antibiotics(ref_mutations))
    # Set variables for nucleotide/amino acid so the same name can be used.
    if seq_type == "nucl":
        ref_resistance_mutations = ref_mutations.nucl
        any_coverage = False
    elif seq_type == "prot":
        ref_resistance_mutations = ref_mutations.prot
        any_coverage = False
    ref_mutations_dict = {}
    for mutation in ref_resistance_mutations:
        ref_mutations_dict[int(mutation[0])] = mutation[1:]
    for i in range(1, seq_length + 1):
        if i not in positions_covered:
            # If the current position isn’t covered and corresponds to a reference
            # mutation, add item to coverage report, including resistance type of the
            # mutation.
            if i in ref_mutations_dict.keys():
                mutation_coverage_report.append([targets_db[target_ID][1], antibiotics, i,
                    seq_type, ref_mutations_dict[i][1]])
            # If the current position isn’t covered and doesn’t correspond to a reference
            # mutation, add item to coverage report, but only if the —report_all_coverage
            # argument is given.
            elif args.report_all_coverage is True:
                mutation_coverage_report.append([targets_db[target_ID][1], antibiotics, i,
                    seq_type, ""])  
            # If no base is covered, add target ID to target_ID_not_covered report.
            if seq_type == "nucl" and any_coverage is False:
                target_ID_coverage_report.append([targets_db[target_ID][1], antibiotics])
    return mutation_coverage_report, target_ID_coverage_report

def write_report_file(report, header, file_path):
    L_print("Writing report: {0}".format(file_path), 2)
    file = open(file_path, ‘w+’)
    file.write(header + ‘
’)
for row in report:
    file.write(\"\t\".join([str(x) for x in row]) + "\n")
file.close()

# ############################################################################
# Database handling
# ############################################################################

# Read database file ('mutations' or 'targets') and return dict
# with keys being all unique values at index 0 in each line, and the corresponding
# values being lists with the remaining values of each line in list format.
def read_database(database_type):
    L_print(\"Reading {0} database.\".format(database_type), 2)
    # Set file path according to database type
    if database_type == "mutations":
        file_path = pipeline_dir + "/mutation_databases/mutations"
    elif database_type == "targets":
        file_path = pipeline_dir + "/mutation_databases/targets"
    else:
        return None
    if check_file(file_path) is False:
        print(\"ERROR: Could not find {0} database. Check if correct file is present with path '{1}'. Exiting.\".format(database_type, file_path))
        exit()

    with open(file_path, 'r') as myfile:
        contents = myfile.read()

    lines = contents.splitlines()
    db_dict = collections.OrderedDict()

    for i in range(1, len(lines)):
        lines[i] = lines[i].split(\"\t\")
        # Since there is only one entry for each target ID, each key in db_dict will
        # contain one list without sublists.
        if database_type == "targets":
            db_dict[lines[i][0]] = lines[i][1:]
        else:
            if lines[i][0] in db_dict:
                db_dict[lines[i][0]].append(lines[i][1:])
            else:
                db_dict[lines[i][0]] = [lines[i][1:]]

    return db_dict

# Takes the full targets database dict and returns a dict with only the targets
# corresponding to the correct species.
def get_targets_for_species(targets_db):
    filtered_targets_db = collections.OrderedDict()
    for target_ID, target_data in targets_db.items():
        if target_data[0] == args.species:
            filtered_targets_db[filtered_targets_db[target_ID] = target_data
        else:
            return filtered_targets_db

# Searches proteins database and returns species supported by the pipeline
def get_supported_species(targets_db):
    supported_species = set()
    for target_ID, data in targets_db.items():
        if data[0] not in supported_species:
            supported_species.add(data[0])

    return sorted(supported_species)
# Return reference sequence mutations (both nucleotide and protein) corresponding to a certain target ID, and species.
def read_ref_mutations(target_ID, species, mutations_db, targets_db):
    L_print("Reading resistance mutations.", 2)
    ref_mutations = mutations_db[target_ID]
    nucl_mutations = []
    prot_mutations = []
    for i in range(len(ref_mutations)):
        if targets_db[target_ID][0] == species:
            if ref_mutations[i][1] == 'nucl':
                nucl_mutations.append(ref_mutations[i][2:] + [ref_mutations[i][0]])
            else:
                prot_mutations.append(ref_mutations[i][2:] + [ref_mutations[i][0]])
    RefMutations = collections.namedtuple('RefMutations', 'nucl, prot')
    ref_mutations = RefMutations(nucl_mutations, prot_mutations)
    return ref_mutations

def get_involved_antibiotics(ref_mutations):
    antibiotics = set()
    for ref_mutation in ref_mutations.nucl + ref_mutations.prot:
        if ref_mutation[3] not in antibiotics:
            antibiotics.add(ref_mutation[3])
    return antibiotics

# Translates DNA sequences in infile. Table 11 is for bacterial translation (Use transeq −help for other organisms)
def translate(infile, outfile=None, table=11):
    if outfile is None:
        outfile = infile + '_translated'
    L_print(" Translating '{0} ' ' and saving output to '{1}'" . format( infile , outfile ) , 2)
    L_execute("transeq −sequence {0} −snucleotide1 −table {1} −outseq {2}" . format( infile , table , outfile ) , level=2, header="transeq output:")
    return outfile

# Removes reading frame suffix added to sequence names by EMBOSS’s transeq
def remove_translation_suffixes(infile):
    with open(infile, 'r') as myfile:
        translation = myfile.read()
    L_print("Translating '{0}' and saving output to '{1}" . format( translation , infile ) , 2)
    L_execute("transeq −sequence {0} −snucleotide1 −table {1} −outseq {2}" . format( infile , table , outfile ) , level=2, header="transeq output:")
    return outfile
B. Pipeline Code

```
# if there are no spaces. The following code removes that suffix.
if ' ' in line:
    corrected_name = re.sub(r'^.*(?=_1)(.*)', r'^\1\3\4', line)
else:
    corrected_name = re.sub(r'^.*(?=_)$', r'^\1', line)
sequences.append([corrected_name, ''])
else:
    sequences[-1][1] += ''.join(line.split())

# Overwrite old file.
with open(infile, 'w') as myfile:
    for i in range(len(sequences)):
        myfile.write(sequences[i][0] + '\n' + sequences[i][1] + '\n')

# Complement, including ambiguous base codes (except those that are their own complement)
def complement(sequence):
    trantab = str.maketrans('ATCGKMRYBVHDatcgkmyrbdhv', 'TAGCMYKRVBDHtagcmkyrbdh')
    sequence = sequence.translate(trantab)
    return sequence

def get_fseq(fseq, genome):
    pattern = r'^>' + re.escape(fseq) + r'(.*?)(>|$)'
    match = re.search(pattern, genome, re.DOTALL).group(1)
    match = ''.join(match.split())
    return match

def write_fasta(fasta_list, outfile):
    of = open(outfile, 'w')
    for i in range(len(fasta_list)):
        of.write('>' + fasta_list[i][0] + '\n' + fasta_list[i][1] + '\n')
    of.close()
    return outfile
```

```