Extracting introns from eukaryotic genomes

CS39440 Major Project Report

Author: Ben Cope (bec49@aber.ac.uk)

Supervisor: Dr Wayne Aubrey (waa2@aber.ac.uk)

5th May 2023

Version 2.0 (Final)

This report is submitted as partial fulfilment of a BSc degree in Computer Science (G400)

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name: **Ben Cope**

Date: **3rd May 2023**

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: **Ben Cope**

Date: **3rd May 2023**

Acknowledgements

I am grateful to the university for giving me the option to perform my best work in an area of computer science that interests me.

I'd like to thank my supervisor for assisting me throughout the project, as well as my family for their love and support. I'd also like to thank Nick and Chase from the FrameRate team for helping me train FrameRate on the intron data I gathered.

Abstract

The purpose of my project is to research the parts of DNA that aren't transcribed to DNA, also known as introns. Introns are a less researched topic in genetics since introns are not protein-coding, meaning their sequences never get converted to amino acids, which are what code for proteins. The project has involved me writing Python code to create a program 'Intron Finder' that extracts introns from multiples eukaryotic genome files, namely Fungi genomes, using known knowledge about the genome. The development and progress of this code was managed using a 7-day sprint. The acquired intron data can then be trained on a machine-learning model that predicts the correct reading frame for unassembled DNA sequences. A reading frame is the way that a sequence of DNA can be read. There are 6 possible reading frames, 3 in the forward direction and 3 in the reverse direction, but only 1 of them is correct. For protein-coding sequences of DNA, there is always a correct reading frame, but introns are always non-coding DNA sequences, so they do not have a correct reading frame. I want to see how much sequence similarity there is between the intron data I gather, and known proteins, by uploading the intron data I gather to a protein database. I aim to see if the model can recognise that the introns are non-coding sequences. This should educate me more about how the machine-learning model works and what features of protein-coding sequences help it to identify the correct reading frame.

My project has also involved me displaying graphical information about introns, including the average lengths of introns and their most common nucleobases. I've used a Python package to help me do this, which allows me to write code that automatically generates graphs, including things like bar charts and scatter graphs. This has helped to further educate me on fundamental information about the introns. This was done over 1 genome and then 10 genomes, to see how close to the average 1 genome on its own is.

Contents

1. Background, Analysis & Process

1.1. Background & Analysis

Introns are parts of DNA that are not transcribed to RNA, and they are therefore always non-coding sections of DNA (they don't code for amino acids which are what get converted into proteins). The parts of DNA that are transcribed to RNA are called exons, which often do code for proteins. A common misconception is that exons are always protein-coding sequences, but this misconception was understood by reading a scientific paper that highlights the key differences between exons and introns.

A protein-coding sequence of DNA can be read in 6 different ways. This is because it can be read forward or backward, and each way it's read can be read in 3 different 'reading frames'. DNA consists of 4 different DNA letters (called nucleobases or nucleotides), namely adenine, cytosine, guanine, and thymine, which are represented by the letters 'A', 'C', 'G', and 'T'. These nucleobases together make up codons, which are collections of 3 nucleobases, e.g., 'GAT' or 'CAA'. So, the start of the DNA sequence 'AGCTATGC' could be read as either 'AGC TAT GC', 'GCT ATG C', 'CTA TGC' in the forward direction, or 'CGT ATC GA', 'GTA TCG A', 'TAT CGA' in the backward (reverse) direction, hence meaning there's 6 different ways to read a DNA sequence. Here's an example of a short DNA sequence and its possible reading frames in the **forward** direction:

ATGTGCAATGCGCTATGCATAGCACCTGACGCCGTCTAA

Forward Direction->

ATG TGC AAT GCG CTA TGC ATA GCA CCT GAC GCC GTC TAA

A TGT GCA ATG CGC TAT GCA TAG CAC CTG ACG CCG TCT AA

AT GTG CAA TGC GCT ATG CAT AGC ACC TGA CGC CGT CTA A

Figure 1.1 - A sample DNA sequence and its 3 possible frames in the forward direction.

It is important to note that only 1 of these reading-frames is correct, and the other 5 are all incorrect. Predicting which frame is the correct frame is quite complex, but there exists a machine-learning model called 'FrameRate' that can predict the correct reading frame for an unassembled DNA sequence. This model has been trained on a huge amount of DNA sequences of which the correct frame is already known, and the model is told which frame is correct for each sequence. After being trained on so many sequences, it can quite accurately predict the correct frame for unassembled DNA sequencing reads of which the correct frame is not already known, which is very useful since genome assembly is usually very slow and requires a lot of computational power, but predicting the correct frame makes genome assembly much faster.

This model has currently only been trained on protein sequences that have been converted from exons; introns can never be protein-coding, and therefore don't have any correct reading frames. An

interesting question however is what happens when the model is given intron sequences? Since introns don't have any correct reading frames, what will the machine-learning model predict is the correct reading frame? Obviously, the model will always be wrong since there is no correct reading frame, but it will be interesting to see if there's any patterns of predictions that emerge after it has been given enough intron sequences that are long enough, ideally introns that are at least 200 nucleobases long.

The project has involved downloading many Fungi genome files from 'Ensemble' and writing Python code to extract the introns from them. For each genome, there is a FASTA file and a GFF file. The FASTA file is the file of the genome's entire sequence, split into sub-sections for each chromosome. Each chromosome contains a header on the first line, followed by millions of nucleotides (A's, C's, G's, and T's) on the following lines until the next chromosome.

```
>ML978066 dna:supercontig supercontig:Aaoar1:ML978066:1:5683700:1 REF
TTGGATAGGCGCCATAGCCCTCCATTGTGGGTGTTAGAACAAGGGCAATTCCTGCCACCT
ATACTGGCTAGAGGCCTGAGTGGCCCAGATTAGCTTAGTATGATTACATAATGCTCCCTA
TAACACTGGCTGAGAAACAATAAGTTTCCTCAGCGATTCGTCTCTATCATTTGGGGATAA
TAGAATTGACTCGATCTCACTATTGCTAATTGCCAACTTCTAAGTGCCCTGCAAACCAAT
CCTAACAGTTATCACCCTGTCTTGATATTCGCTTTACCCTATACAAACCTCTGCGATATA
CCCCTCTTACCTTGTGTAGTAAGTTACCTATCGTTACAGAAGAGGACGATAGAAGATCTT
TTCCTTGGATTATGAAAGGCACTACATCACTTGCGTGCTATTTAAAGCTGTTTAGCACAT
CAAGCGAGCCTATATTTCCCCAAAAATATAAAGTTGAAATATTTGTCTGATGTATCGATT
CTACGATCTTGCTGCTGACCCTGTTCTGTCATGTTATGCTCTATATAGTAGGCAATCTCG
GCCTCTTTACCGAGCATGCCGCTATCCACCATGTGAGTCTTAGTCTTCAAACTGTCGGTA
GGCCGCATCTGGGTGGGCAAATGGTAGGGTGGGGAGTCCTAGTCTTAAGGCCAGTGGGGA
GTCCAAGTGCGCAGTGGGGTGATTGAACAACCTCGGACTAGCGCTGTTGGGTGGCTTGCT
```

Figure 1.2 – FASTA file of 'Aaosphaeria arxii' genome (only the first few lines included).

The GFF file is the file that contains information for every gene in the genome. Each gene has its own section separated by 3 hashtags, and contains lots of information about the gene, including its coordinates (position) in the genome, its strand (meaning if the gene is read in the forward or reverse direction), whether the gene is protein coding or not, and a list of each exon in the gene with coordinates and other information for each exon. The GFF file contains a line for each feature (features can be exons, untranslated regions, coding-sequences, mRNA's, etc), and each feature has 9 fields, which are separated by tabs. These fields are:

1) The ID of the chromosome that the feature is on
2) The source of the feature
3) The type of feature it is, e.g. exon,
4) The start coordinate of the feature, meaning the start position of the feature in the genome (how many nucleotides along to get to the start of the feature)
5) The end coordinate of the feature (end position of the feature in the genome)
6) The score of the feature (how confident that it's accurate)
7) The strand (whether the feature is read in the forward or reverse direction)
8) The starting frame (whether the first codon of the feature's sequence is read starting from the 1st, 2nd, or 3rd base)
9) The features attributes, which contain additional information about the feature.

Fields 6 and 8 sometimes contain a full-stop to indicate that their values are empty.

```
##sequence-region   ML978118 1 1224
##sequence-region   ML978119 1 1153
##sequence-region   ML978120 1 1000
#!genome-build DOE Joint Genome Institute Aaoar1
#!genome-version Aaoar1
#!genome-build-accession GCA_010015735.1
#!genebuild-last-updated 2020-02
ML978066    Aaoar1  supercontig 1   5683700 .   .   .   ID=supercontig:ML978066;Alias=ML978066.1
###
ML978066    ena gene    774 2306    .   +   .   ID=gene:BU24DRAFT_416053;biotype=protein_coding;description=NAD(P)-binding proteir
ML978066    ena mRNA    774 2306    .   +   .   ID=transcript:KAF2020338;Parent=gene:BU24DRAFT_416053;biotype=protein_coding;trans
ML978066    ena five_prime_UTR  774 1087    .   +   .   Parent=transcript:KAF2020338
ML978066    ena exon    774 1123    .   +   .   Parent=transcript:KAF2020338;Name=KAF2020338-1;constitutive=1;ensembl_end_phase=0;
ML978066    ena CDS 1088    1123    .   +   0   ID=CDS:KAF2020338;Parent=transcript:KAF2020338;protein_id=KAF2020338
ML978066    ena exon    1190    1606    .   +   .   Parent=transcript:KAF2020338;Name=KAF2020338-2;constitutive=1;ensembl_end_phas
ML978066    ena CDS 1190    1606    .   +   0   ID=CDS:KAF2020338;Parent=transcript:KAF2020338;protein_id=KAF2020338
ML978066    ena CDS 1662    2186    .   +   0   ID=CDS:KAF2020338;Parent=transcript:KAF2020338;protein_id=KAF2020338
ML978066    ena exon    1662    2306    .   +   .   Parent=transcript:KAF2020338;Name=KAF2020338-3;constitutive=1;ensembl_end_phas
ML978066    ena three_prime_UTR 2187    2306    .   +   .   Parent=transcript:KAF2020338
###
ML978066    ena gene    9625    11341   .   -   .   ID=gene:BU24DRAFT_445593;biotype=protein_coding;description=carbohydrate-bindi
ML978066    ena mRNA    9625    11341   .   -   .   ID=transcript:KAF2020339;Parent=gene:BU24DRAFT_445593;biotype=protein_coding;t
ML978066    ena exon    9625    10829   .   -   .   Parent=transcript:KAF2020339;Name=KAF2020339-2;constitutive=1;ensembl_end_phas
ML978066    ena CDS 9625    10829   .   -   2   ID=CDS:KAF2020339;Parent=transcript:KAF2020339;protein_id=KAF2020339
ML978066    ena exon    10885   11341   .   -   .   Parent=transcript:KAF2020339;Name=KAF2020339-1;constitutive=1;ensembl_end_phas
ML978066    ena CDS 10885   11341   .   -   0   ID=CDS:KAF2020339;Parent=transcript:KAF2020339;protein_id=KAF2020339
###
ML978066    ena gene    13342   14862   .   -   .   ID=gene:BU24DRAFT_360743;biotype=protein_coding;description=hypothetical prote
ML978066    ena mRNA    13342   14862   .   -   .   ID=transcript:KAF2020340;Parent=gene:BU24DRAFT_360743;biotype=protein_coding;t
ML978066    ena three_prime_UTR 13342   13368   .   -   .   Parent=transcript:KAF2020340
ML978066    ena exon    13342   13778   .   -   .   Parent=transcript:KAF2020340;Name=KAF2020340-5;constitutive=1;ensembl_end_phas
ML978066    ena CDS 13369   13778   .   -   2   ID=CDS:KAF2020340;Parent=transcript:KAF2020340;protein_id=KAF2020340
ML978066    ena exon    13831   13962   .   -   .   Parent=transcript:KAF2020340;Name=KAF2020340-4;constitutive=1;ensembl_end_phas
ML978066    ena CDS 13831   13962   .   -   2   ID=CDS:KAF2020340;Parent=transcript:KAF2020340;protein_id=KAF2020340
ML978066    ena exon    14014   14064   .   -   .   Parent=transcript:KAF2020340;Name=KAF2020340-3;constitutive=1;ensembl_end_phas
ML978066    ena CDS 14014   14064   .   -   2   ID=CDS:KAF2020340;Parent=transcript:KAF2020340;protein_id=KAF2020340
ML978066    ena exon    14118   14553   .   -   .   Parent=transcript:KAF2020340;Name=KAF2020340-2;constitutive=1;ensembl_end_phas
ML978066    ena CDS 14118   14553   .   -   0   ID=CDS:KAF2020340;Parent=transcript:KAF2020340;protein_id=KAF2020340
ML978066    ena CDS 14605   14631   .   -   0   ID=CDS:KAF2020340;Parent=transcript:KAF2020340;protein_id=KAF2020340
ML978066    ena exon    14605   14862   .   -   .   Parent=transcript:KAF2020340;Name=KAF2020340-1;constitutive=1;ensembl_end_phas
ML978066    ena five_prime_UTR  14632   14862   .   -   .   Parent=transcript:KAF2020340
###
```

Figure 1.3 – GFF file of 'Aaosphaeria arxii' genome (only the first 3 genes included).

The GFF file does not explicitly state information about the introns, but it can be worked out based on the information of the exons, since introns are found between exons. Once this data was gathered, it was sent to one of the developers of FrameRate who ran it through the SwissProt database to look for any types of sequence similarity.

The project was of great interest after a module in the previous semester called 'Computational Bioinformatics' which made links between biology and computer science. Having done this module gave a good background for the project. Since introns are a less explored area of genetics, the project felt unique.

The main problem to be solved was the ability to successfully extract introns from any type of formatted genome. Steps were made to identify the different tasks that this would involve. Another problem was how to display the intron data. Originally this was displayed with a message saying the intron was found, as well as relevant detail about the intron including its size, coordinates, strand, and sequence. However, this was changed later to be in FASTA format due to one of the FrameRate developers requesting it be in this format, to make it easier to train the model. The main objectives of the program were as follows:

- Load GFF and FASTA files into the program.
- Be able to consistently process genes from GFF files correctly.
- Find coordinates of introns from the GFF file.

- Find sequences in the FASTA file that match the corresponding GFF file's coordinates.
- Display intron sequences.
- Display an intron's size, strand, and coordinates.
- Graphically display information about introns.
- Verify that the sum of coding sequences for a gene is divisible by 3.
- Run the intron data through a protein database.

1.2. Process

The software process followed was an agile approach that involved a 7-day iteration. This iteration started at Thursday mid-day and ended the following Wednesday. This was based around these days because of a weekly individual meeting with the supervisor every Thursday at 11:30 AM. In these meetings, the discussion involved what tasks had been completed each week and any problems that had occurred. Towards the end of each meeting, the supervisor set out a set of tasks to be completed before the following Thursday. This 7-day iteration was found to be very effective because it made sure that tasks were completed on time consistently and helped track less productive weeks based off how much progress had been shared with the supervisor.

For backing up of all the files, Aberystwyth University's Department GitLab server was used, where a folder was created for each week of the project so that the supervisor could see how much work was being done each week, and to help keep track of the progress. Every week, the work done was uploaded to that week's folder in the GitLab depository, often including things like changes to the main code, text-files that the Python program created, weekly blog updates etc. The reason GitLab was used is because it is a very organised way for storing files (it lists all the commits/uploads that have been made and their exact dates). It also supports formatting for viewing Python files and PDF files without having to download them, meaning the supervisor could easily view the files.

Name	Last commit	Last update
Week 1 (30th Jan - 5th Feb)	Updating a comment about where the se...	2 months ago
Week 10 (3rd April - 9th April)	image generated by matplotlib that show...	2 weeks ago
Week 11 (10th April - 16th April)	Upload New File	1 week ago
Week 12 (17th April - 23rd April)	introns file updated - sequences are now...	6 days ago
Week 13 (24th April - 30th April)	Replace Report.docx	23 hours ago
Week 2 (6th Feb - 12th Feb)	Text file containing what gets written to i...	2 months ago
Week 3 (13th Feb - 19th Feb)	Blog after week 3	2 months ago
Week 4 (20th Feb - 26th Feb)	Update for what I've done in Week 4 so far	2 months ago
Week 5 (27th Feb - 5th March)	Introns for first 3 genomes	1 month ago
Week 6 (6th March - 12th Mar...	Introns found from updated Intron Finder ...	1 month ago
Week 7 (13th March - 19th Mar...	Replace Blog.pdf	1 month ago
Week 8 (20th March - 26th M...	Introns for first 3 genomes, without chec...	1 month ago
Week 9 (27th March - 2nd April)	Updated introns-file that says whether th...	1 month ago
Blog.docx	Updated for end of week 2	2 months ago
Blog.pdf	Replace Blog.pdf	6 days ago
README.md	Initial commit	2 months ago
bec49_ProjectOutline.pdf	Project Outline (finished 3 days ago)	2 months ago

Figure 1.4 – The GitLab project repository containing folders for every week of the project.

Additionally, GitLab contains an ‘Issues Board’ which was found useful to use as a Kanban board. In the board, a task was added to the ‘Open’ section every time a new task was given by the supervisor or set by myself. Then any tasks that were actively being worked were moved to the ‘WIP’ section, and tasks completed were moved to the ‘Closed’ section. These 3 sections are visually displayed in the board which allowed easy viewing of how much work-in-progress there was, to make sure not too many tasks were being done at once, and how many tasks were completed, to keep track of progress. Another useful thing about GitLab issue boards is that each task is given a number in relation to how many tasks were made before that. For example the 5th task created was given the label ‘#5’. This made it clear which items were being ignored, so if task ‘#20’ had just been finished, whilst task ‘#3’ hadn’t been started yet, it was clear that either work needed to start being done on that task, or it should be evaluated as to whether it is still necessary to complete it.

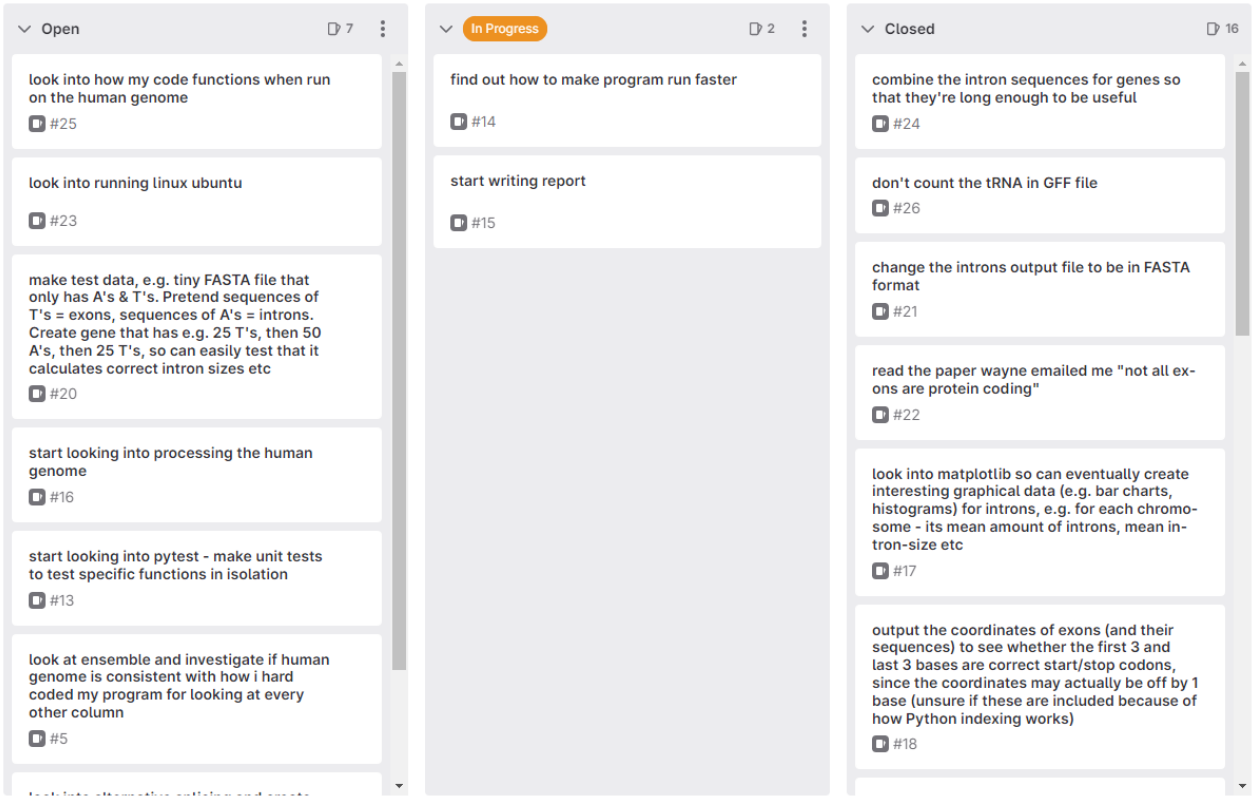


Figure 1.5 – GitLab issues board during the last couple weeks of the project.

A weekly blog was updated at least once a week with an average of a few paragraphs written for each week's section of what was done that week. The content written for each week was a brief summary of the key things that had been done that week and any issues encountered. This was another thing that helped keep track of the progress.

Major Project Blog

Week 1 (30th Jan – 5th Feb)

This week was the first week of the project. I met my tutor for the project and started to work on some of the coding. During my individual meeting with my tutor, I was given a list of zipped files of genomes, and was tasked with writing a program that finds the introns within the files. First, I downloaded WinZip so that I could unzip the genome files. Then, I began by writing some pseudocode outlining the basis of how I wanted to code this program. I realised that each gene had itleast 1 exon, and since introns are found between exons, it meant that I would need to inspect each gene containing 2 or more exons, and then would find the introns between the end of one exon and the start of the next exon. Afterward in Python, I wrote code to open one of these genome GFF files and then a function to go through each gene of the file and find the introns within it, and then write the coordinates and length of those introns to a text file. The output of my program seemed correct, although I only ran this for 1 genome, so I need to find a way of running this program for every genome file.

Week 2 (6th Feb – 12th Feb)

During this week, I wrote my project outline to highlight the key parts of my project that I intend to do. Also, following a meeting with my tutor, I realised that the intron finder program I had written last week only searched for the start and end coordinates within each gene but didn't look at the strand, meaning I had assumed that all the introns in my program were on the positive strand. I also only looked at the GFF file of a genome and not its FASTA file, which would provide a lot more detail about the intron. I was also told to start looking into glob which will eventually allow my program to iterate through all the DNA files. I will likely start doing this next week.

I altered my program so that I also checked whether each gene was found on the positive/forward strand or negative/reverse strand, so I could take this into account when outputting the coordinates of the intron. This wasn't too difficult since the strand is just the 7th field in the GFF file format. The next task was tricker – outputting the DNA string of each found codon, by finding its sequence of bases in the corresponding FASTA file. I realised that for each gene, they had an ID in the 9th field containing the ID of the gene in the corresponding genome's FASTA file. By obtaining this ID, I could use it to get the DNA string of the gene from the FASTA file and then use the intron coordinates to

Figure 1.6 – The first couple weeks of content written to the blog.

2. Design

An intron is always found between two exons, more specifically from the end-coordinate of one exon to the start-coordinate of the following exon. Since this exon-information is already in the GFF file, the coordinates for all introns can be found by traversing through each gene and checking if it has at least 2 exons (if a gene has only 1 exon, it cannot have any introns since introns are found between exons), and if it does, then traverse through that gene's exons and create a list of lists that contain the previous exon's end-coordinate and the current exon's start-coordinate, and store this as an intron-coordinate list. As well as this, the chromosome ID of the gene is also extracted since this is needed to find the DNA sequence of the intron in the FASTA file. With the chromosome ID, the chromosome sequence for that ID is searched for in the FASTA file and then the sequence at the indexes of the intron's coordinates is extracted. By doing this for every gene, every sequence of introns in a genome can be extracted.

Here is a snippet from a GFF file of a genome. There are 2 genes, both encapsulated by triplets of hashtags. Both genes are on chromosome 1. The first gene's start-coordinate is 7369 and end-coordinate is 11334. It only contains 1 exon, so there are no introns. The second gene starts at 12014 and ends at 12291. It contains 2 exons, so there must be 1 intron. The first exon **ends** at

12041, whilst the second exon **starts** at 12095, so the sequence between these coordinates is the intron.

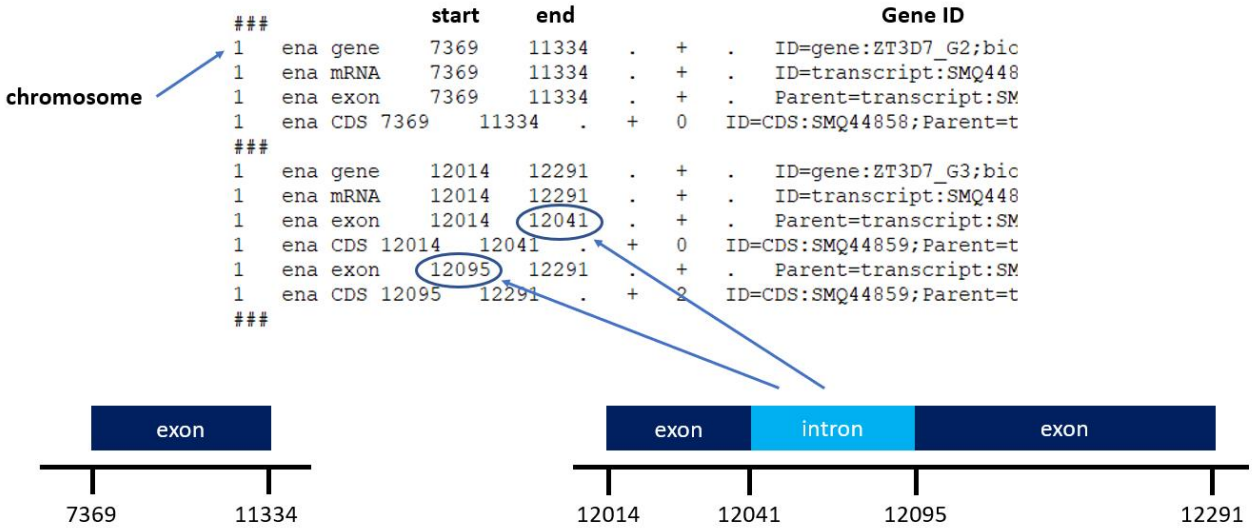


Figure 2.1 – First 2 genes of a GFF file.

Below is a snippet from the same genome's FASTA file. The first line contains the header, which contains information about the chromosome including its ID and its size. This chromosome is chromosome 1 so its sequence should contain the intron found in the GFF file since that was also on chromosome 1. By going along 12,041 nucleobases, the sequence of the intron can be found. Its coordinates are [12041, 12095] so its length is 55 nucleobases.

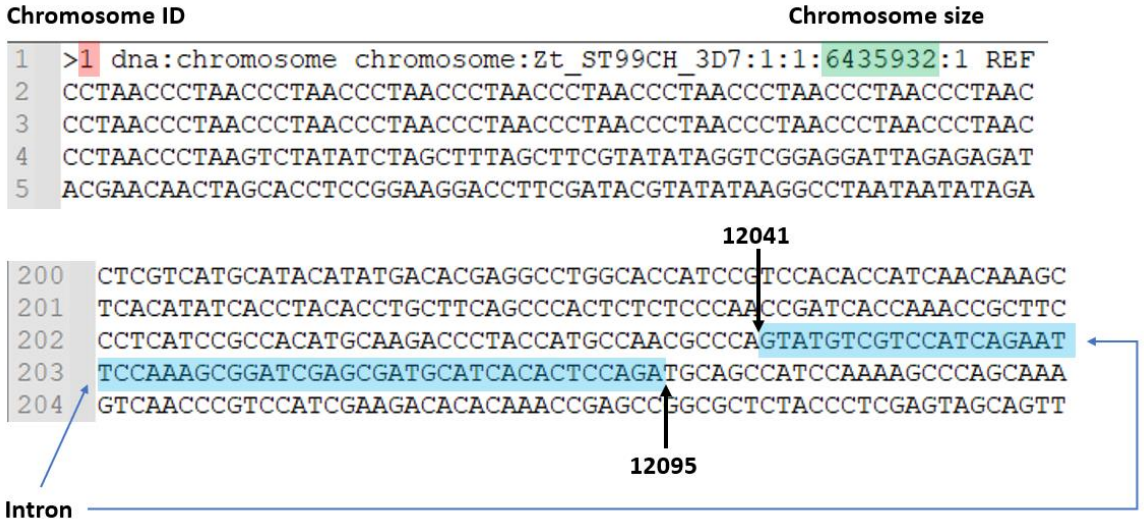


Figure 2.2 – First few lines of the first chromosome in the corresponding FASTA file.

After all the introns for a genome have been extracted, the intron sequences are stored in a text file. Originally this text file was in a format whereby it would say "intron found" whenever an intron is found, and then display its coordinates, its size, its strand (forward or reverse), and its sequence. However, it was decided to be changed later to be in FASTA format, whereby each intron would be

identified by the ID of the gene it's in, and whether it's the first, second, third etc intron in the gene. This is because it makes it easier to train FrameRate this way, since it is familiar with FASTA format. Also, another file was created where the intron sequences for a gene were combined together, since the average sequence length for introns was about 55, which is not long enough to be useful – one of the FrameRate team members highlighted that intron sequences should be ideally at least 200 nucleobases long, which only a small amount were, but when combining introns together for a gene, there were a lot more sequences that were at least 200 nucleobases long.

Below is the flowchart that covers the main aspects of the intron finding program. It starts by acquiring a list of the pathnames of every FASTA and GFF file. It then uses these pathnames to loop through every pair of files (every genome's FASTA and GFF file). For each genome, it opens that genome's FASTA and GFF file, and then loops through all the genes in the genome's GFF file. For each gene, it finds the coordinates for the introns based off the coordinates of the exons, then goes to these sets of coordinates in the FASTA file to retrieve the DNA sequences of these introns. These sequences, as well as other information about the introns, are then written to a text file, and then finally the program checks if the sum of the gene's coding sequences is divisible by 3 or not. If it isn't divisible by 3, then the original file must have had an assembly error since coding sequences are made up of codons which are 3 nucleobases long, so a message about the error is written to the text file.

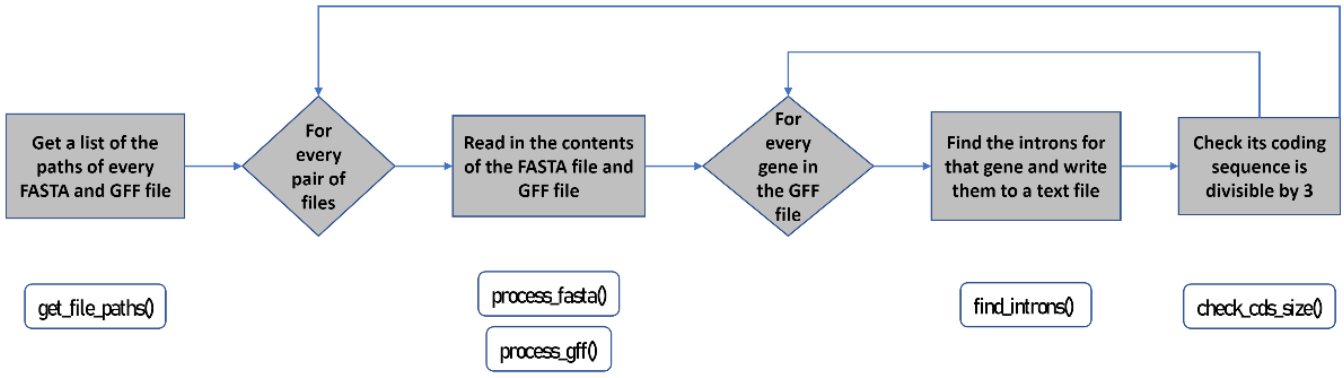


Figure 3.3 – Flowchart of the overall 'Intron Finder' program.

Pseudocode was written to outline how to find the introns in a gene, in more detail. In GFF format, genes are defined between three hashtags "###", so the code should loop between each section of data enclosed by "###". If the current gene enclosed by "###" contains only 1 exon, then skip to the next gene since that gene cannot contain introns. If the current gene contains 2 or more exons, then loop through each exon except the last, and each time retrieve the end-coordinate of the current exon and the start-coordinate of the next exon, and store this as the intron's coordinates. Then calculate the difference of these values to find the intron size and assign it to a temporary variable (which gets reassigned for each gene containing more than 2 exons). Then output these values to the terminal, and/or append these values to a text file.

```
CREATE variable 'temp_intron'
FOR every gene between "###"s DO
        IF gene contains 1 exon THEN
                SKIP gene
        ELSE
                FOR i = RANGE of 1 to number of exons DO
                        SUBTRACT end-coordinate of 'i'th exon from start-coordinate of 'i+1'th exon
                        ASSIGN 'temp_intron' the calculated value
                        OUTPUT 'temp_intron'
                        WRITE 'temp_intron' to new file
                ENDFOR
        ENDIF
ENDFOR
```

Figure 3.4 – Initial pseudocode for finding introns and retrieving their coordinates.

The programming language 'Python' was used for all the coding in this project. This was found to be the most efficient language for a variety of reasons. Firstly, Python is an interpreted language, which was useful for the program since the program involved often making small changes to the code and then running it again, which interpreted languages are much faster for, also meaning bugs can be fixed quicker. Additionally, the project involved processing a large amount of data – hundreds of files that are each around hundreds of thousands of lines long, so a language that is good for processing lots of data was needed. Furthermore, Python includes lots of useful built-in tools; the 'gzip' module was used for opening zipped files (which the genome files were when downloaded from Ensemble), the 'glob' module was used for iterating through all the files in the genome-files directory, and the 'matplotlib' module was used for creating statistical graphs for the intron data, which was found useful for displaying distinctive information about the introns, e.g. average intron size across a genome.

3. Implementation

3.1. Finding intron-coordinates in a GFF file

The start of the project involved being given a list of zipped genome files by the supervisor, specifically Fungi genomes from 'Ensemble'. These files were gunzipped, but the project was being

done on Windows, so the program 'WinZip' was downloaded to unzip the genome files, although only 1 genome file was unzipped since it was only the early stages of the project.

After looking through the GFF file for this genome, it was realised that each gene had at least 1 exon, and since introns are always found between exons, it meant that only genes containing 2 or more exons would need to be inspected, with code storing intron-coordinates as the end-coordinate of one exon and the start-coordinate of the following exon. This built upon the pseudocode discussed previously. Python code was written to open a genome's GFF file and a function that traverses each gene in the file and finds the introns within it, and then writes the coordinates and sizes of those introns to a text file.

```
import re

def process_gff(file_name):
    with open(file_name, 'r') as f:
        data = f.read().split("###")
    del data[0] # information before first '###' doesn't represent a DNA sequence
    return data


def find_introns(sequence):
    intron_file = open("introns.txt", 'a')

    row_length = 9
    prev_end_coord = sequence[(row_length * 2) + 4]

    # Exons are defined every other row, so the step should jump ahead 2 rows each
    # iteration of the loop. The loop uses the current start-coordinate and previous
    # end-coordinate, and should iterate through the loop a number of times equal
    # to the number of introns. It starts at the second exon since I already found
    # the end-coordinate of the first exon. The second exon is defined after the 4th row

    for index in range((row_length * 4), len(sequence), (row_length * 2)):
        # Start and end coordinates in a row are found at positions 4 and 5
        current_start_coord = sequence[index + 3]

        # Introns are from the end of one exon to the start of the next exon
        intron = [prev_end_coord, current_start_coord]
        intron_size = int(intron[1]) - int(intron[0])
        intron_info = "\n*INTRON*\tCoordinates: " + str(intron) + "\tSize: " +  str(intron_size)
        intron_file.write(intron_info)

        prev_end_coord = sequence[index + 4]

    intron_file.close()

def main_program():
    sequences = process_gff("Zymoseptoria_tritici_st99ch_3d7_gca_900091695.Zt_ST99CH_3D7.52.gff3")

    for index in range(len(sequences)):
        # Split the sequence on newline and tab characters
        seq_i = re.split('\n|\t', sequences[index])
        del seq_i[0], seq_i[-1] # Remove the empty strings at the start and end

        # If sequence has more than 36 items, then it has more than 4 rows and
        # therefore multiple exons, so there will be itleast 1 intron
        if (len(seq_i) > 36):
            find_introns(seq_i)


main_program()
```

Figure 3.1 – Initial 'Intron Finder' program, capable of extracting the coordinates of introns in 1 genome.

In the initial draft of code, there are 3 functions. Firstly, the 'process_gff' function which is given the name of a GFF file and then opens that file in read-mode and stores a list of all its genes by splitting the data into strings between every triplet of hashtags. It then deletes the first string since the information before the first "###" is just information about the file, and then returns the list of genes.

The third function 'main_program' uses 'process_gff' to retrieve a list of genes for a genome, and then loops through these genes, splitting each gene-string on its newline and tab characters, since these characters are what are used to separate fields in GFF format. These fields for this gene are then stored as strings in a list, with the first and last strings being removed because they are empty since there is always a newline character at the start and end of a gene. For this genome, it was discovered that gene's containing only 1 exon only contain 4 features (so are 4 rows long), but gene's containing 2 or more exons contain another 2 rows (features) for each exon. Since features have 9 fields each, it was figured that the genes containing intron(s) could be determined by how many fields they have, which can be calculated by working out the length of the list of fields. A gene containing only 1 exon has 4 rows/features, and each feature has 9 fields, so 9 times 4 = 36. Therefore, any gene with more than 36 fields must have 1 or more introns. This was used as a condition for each gene so only genes with over 36 fields would be searched for introns, which is where the second function came in.

The second function ('find_introns') is more complex. It starts by opening the intron output text file (or creating it if the program hasn't been run yet). It then sets up some key values; the constant 'row_length', which represents how many fields are in a row/feature, and the variable 'prev_end_coord' which keeps track of the previous exon's end-coordinate each time round the loop. Since exons in this genome are defined every other row in a gene starting from the 3rd row, and the end-coordinate is the 5th field in GFF format, the end-coordinate of the first exon is the 5th field of the 3rd row – rows contain 9 fields, so row 1 starts at index 0 (9 * 0), row 2 starts at index 9 (9 * 1), row 3 starts at index 18 (9 * 2), row 4 starts at index 27 (9 * 3), etc, so the field is found at (9 * 2) + 4 (add 4 to get the 5th field since index 0 is the first field). It was therefore figured that finding field **a** of row **b** can be calculated by:

(row_length * (b – 1)) + (a – 1), where 'row_length' = 9.

This logic was used to traverse through every exon of a gene. A FOR loop was created that starts at the 2nd exon, which is the 5th row (row_length * 4)), and each iteration of the loop jumps ahead to the next exon (row_length * 2). The loop starts at the 2nd exon and not the 1st since each iteration of the loop only needs the end-coordinate of the previous exon, which was already calculated earlier. Also, this means the loop iterates for 1 less than the number of exons, which is the number of introns. Inside the loop, the start-coordinate of the current exon is found (from the 4th field) and then a variable for the intron's coordinates is defined as from the previous exon's end-coordinate to the current exon's start-coordinate. The intron's size is then calculated by finding the difference of these coordinates, which is then written to the intron output text file, as well as the intron's coordinates. Finally at the end of the loop, 'prev_end_coord' is re-calculated to be the end-coordinate of the current exon, so the next exon can use this value.

3.2. Finding the sequences of introns from the corresponding FASTA file

The next major task after this was to extract the sequences of these introns from the genome's FASTA file. The first step was to create a function to return a dictionary of each gene and its sequence.

This involved writing a function 'process_fasta' that opens the FASTA file in read-mode and stores a list of all its gene-sequences by splitting the file's content into strings between every '>' character, since this is what separates genes in FASTA format. Similar to the 'process_gff' function, it then deletes the first item in this list since information before the first '>' just contains information about the FASTA file. After this, a dictionary called 'genome' is created that will hold each gene's ID as keys and each gene's sequence as values. The function then loops through every gene in the list of gene-sequences. For each gene, it creates a partition of the gene at the newline character – this creates a list containing 3 items.

These items are everything before the first newline character, the newline character itself, and then everything after the first newline character. The 1st item of the list therefore contains the header line of the gene, which can be partitioned again at the space character to get the gene ID, since the gene ID is the only thing before the first space character. The 3rd item of the list is the gene's sequence so is stored in a string variable, and then gets all its newline characters removed by replacing them with empty strings. This gene ID and gene sequence are then stored in the genome dictionary as a key-value pair.

```
def process_fasta(file_name):
    with open(file_name, 'r') as f:
        data = f.read().split('>')
    del data[0] # Delete empty string

    genome = {}

    for gene in data:
        # The first line contains the gene's ID and information about the gene
        gene_partition = gene.partition('\n')
        # The gene ID is the string before the first space on the first line
        gene_id = gene_partition[0].partition(' ')[0]
        # Everything after the first line of the gene is its sequence
        gene_string = gene_partition[2]
        gene_string = gene_string.replace('\n', '')

        genome[gene_id] = gene_string

    return genome
```

Figure 3.2 – Function for extracting each gene's ID and sequence from a FASTA file.

Afterwards the 'find_introns' function was updated to include this. The function was updated to additionally output each intron's sequence, strand (forward/reverse), and the ID of the gene it is on. For each gene, its ID was found in the 9th field of its 2nd row, since that's where the gene's attributes are. Inside the loop, the strand is found in the 7th field, and this information was used to determine how to display the intron's coordinates – if the intron was found on the forward strand, then its coordinates were displayed in ascending order, and similarly if it was found on the reverse strand, then its coordinates were displayed in descending order. To get the intron's coordinates relative to

the coordinates of the gene it is in, the difference of the intron's coordinates and the gene's start-coordinate were taken.

For example, if a gene starts at coordinate 117 in the genome, and an intron of that gene starts at coordinate 123 in the genome, then the start-coordinate of the intron relative to the start-coordinate of the gene is 123 – 117 = 6.

The intron sequence could then be obtained by looking up the gene's ID in the genome dictionary to retrieve the gene's sequence, and then using these relative coordinates to find the intron sequence within the gene's sequence.

```
def find_introns(sequence, genome):
    intron_file = open("introns.txt", 'a')
    row_length = 9
    prev_end_coord = sequence[(row_length * 2) + 4]

    # The first row is the gene itself
    gene_start_coord = int(sequence[3])

    # The attributes containing the gene's ID is found in the last column of
    # the 2nd row. The ID is the first item in the attributes
    attributes = sequence[row_length + 8].split(';')
    fasta_id = attributes[0].split(':')[1]

    # Exons are defined every other row, so the step should jump ahead 2 rows each
    # iteration of the loop. The loop uses the current start-coordinate and previous
    # end-coordinate, and should iterate through the loop a number of times equal
    # to the number of introns. It starts at the second exon since I already found
    # the end-coordinate of the first exon. The second exon is defined after the 4th row

    for index in range((row_length * 4), len(sequence), (row_length * 2)):
        current_start_coord = sequence[index + 3] # Start coords is at position 4
        strand = sequence[index + 6] # Strand (+ or -) is found at position 7

        # Introns are from the end of one exon to the start of the next exon
        if (strand == '+'):
            intron = [prev_end_coord, current_start_coord]
        else:
            intron = [current_start_coord, prev_end_coord]

        # Calculate difference between the coordinates
        intron_size = abs(int(intron[1]) - int(intron[0]))

        # The intron sequence within the current gene starts at the difference
        # of the intron's start coordinates and the gene's start coordinates
        intron_start_index = int(prev_end_coord) - gene_start_coord
        intron_end_index = int(current_start_coord) - gene_start_coord
        intron_seq = genome[fasta_id][intron_start_index:intron_end_index]


        # Finally write all of this information to a file
        intron_info = "\n*INTRON*\tGene: " + fasta_id + "\t\t\
Coordinates: " + str(intron) + "\t\tSize: " +  str(intron_size) + "\t\t\
Strand: " + strand + "\t\tSequence: " + intron_seq

        intron_file.write(intron_info)

        prev_end_coord = sequence[index + 4] # End coords is at position 5

    intron_file.close()
```

Figure 3.3 – Updated 'find_introns' function that is now able to find the sequences of introns.

Below contains the output file after running this program on the 'Zymoseptoria tritici' genome. Only the first few nucleobases of sequences are shown for compactness. Some sequences are empty or much smaller than their 'Size' value, for reasons discovered later in the project. At the time, this was assumed to be a mismatch between the FASTA file and GFF file. Also, this data is completely meaningless as it was discovered later that the wrong file was being processed – the FASTA file of CDS's (coding sequences) of the genome was being processed instead of the raw FASTA file for the whole genome. This meant the 'intron sequences' were actually parts of exon sequences.

```
*INTRON*    Gene: SMQ44859    Coordinates: ['12041', '12095']    Size: 54     Strand: +    Sequence: AATGCAGC
*INTRON*    Gene: SMQ44860    Coordinates: ['12992', '12907']    Size: 85     Strand: -    Sequence: CAACAAGA
*INTRON*    Gene: SMQ44860    Coordinates: ['13299', '13247']    Size: 52     Strand: -    Sequence: TTCTACAT
*INTRON*    Gene: SMQ44860    Coordinates: ['14220', '14168']    Size: 52     Strand: -    Sequence: GGCAATTG
*INTRON*    Gene: SMQ44861    Coordinates: ['14830', '14770']    Size: 60     Strand: -    Sequence: ATTGGTGG
*INTRON*    Gene: SMQ44861    Coordinates: ['15045', '14864']    Size: 181    Strand: -    Sequence: CCTGA
*INTRON*    Gene: SMQ44861    Coordinates: ['15163', '15095']    Size: 68     Strand: -    Sequence:
*INTRON*    Gene: SMQ44863    Coordinates: ['16544', '16596']    Size: 52     Strand: +    Sequence: GGACACAC
*INTRON*    Gene: SMQ44864    Coordinates: ['17299', '17240']    Size: 59     Strand: -    Sequence: TTCTTTGC
*INTRON*    Gene: SMQ44866    Coordinates: ['34106', '34042']    Size: 64     Strand: -    Sequence: GAAGAGGA
*INTRON*    Gene: SMQ44867    Coordinates: ['35277', '35978']    Size: 701    Strand: +    Sequence: GGACACGA
*INTRON*    Gene: SMQ44867    Coordinates: ['36228', '36282']    Size: 54     Strand: +    Sequence: GGGCGCGG
*INTRON*    Gene: SMQ44870    Coordinates: ['51302', '51242']    Size: 60     Strand: -    Sequence: CCGCTCCA
*INTRON*    Gene: SMQ44870    Coordinates: ['52199', '52142']    Size: 57     Strand: -    Sequence: TGGAGTTG
*INTRON*    Gene: SMQ44870    Coordinates: ['52617', '52550']    Size: 67     Strand: -    Sequence: TCATCCTG
*INTRON*    Gene: SMQ44873    Coordinates: ['60202', '60142']    Size: 60     Strand: -    Sequence: CCGCTTTG
*INTRON*    Gene: SMQ44873    Coordinates: ['60431', '60383']    Size: 48     Strand: -    Sequence: CGGTCGCG
```

Figure 3.4 – Text file containing the extracted intron information for the 'Zymoseptoria tritici' genome. Note that the gene IDs are incorrect here.

3.3. Processing multiple genomes

The next big task was to allow the program to process multiple genomes instead of just one. The initial idea was to store the names of all FASTA files in a text file, and the names of all GFF files in another text file. These files could then be read line by line so every file could be unzipped and then processed.

A small program was created that traverses the directory of the FASTA files and the directory of the GFF files by using the 'glob' module. For each file ending in '.gz' (meaning gunzipped), the program wrote the filename of that file to a new text file, so that a readable text-file of all genome filenames was created. Only the files ending in '.gz' were written since there is sometimes a very small number of other types of files in the directory, for example 'readme' files. To only retrieve files ending in '.gz', the regular expression ***.gz** was used, meaning any string of characters followed by '.gz' and then nothing afterwards.

```
import glob

# Links of directories holding the FASTA and GFF files
fasta_paths = "C:/Users/benja/OneDrive - Aberystwyth University/Desktop\Major Project/ensemble_fungi/ensemble_fungi/ensembl_fungi/"
gff_paths = "C:/Users/benja/OneDrive - Aberystwyth University/Desktop\Major Project/ensemble_fungi/ensemble_fungi/ensembl_fungi_gff/"

# Calculate length of path names
fasta_paths_length = len(fasta_paths)
gff_paths_length = len(gff_paths)

with open("fasta_names.txt", "w") as file:
    # Find every gunzipped file in the FASTA files directory
    for name in glob.glob(fasta_paths +"*.gz"):
        # Remove the name of path since it's the same for every file
        name = name[fasta_paths_length : :]
        file.write(name + "\n")


with open("gff_names.txt", "w") as file:
    for name in glob.glob(gff_paths + "*.gz"):
        name = name[gff_paths_length : :]
        file.write(name + "\n")
```

Figure 3.5 – Program that stores the filenames of every FASTA and GFF file in a text file.

In this program, a text file called ‘fasta_names’ is created, and then the file directory of the FASTA files is searched for every pathname matching the regular expression – for each matched pathname, a substring is created that just contains the filename since every file has the same path. The substring is then written to the ‘fasta_names’ text file. The exact same process is done for the GFF files.

The idea was that after these filenames had all been stored in text files, they could be opened and each line stored as a string in a list, so the list could be iterated through and the file of each filename be opened and then processed through the intron finder program. Another small program was created that used the ‘gzip’ module to read the contents of a ‘gunzipped’ FASTA file so it could be written to a new text file, so that each file did not have to be manually unzipped.

```
import gzip

def read_fasta(file_name):
    # It will write the FASTA file to the 'genomes_fasta' folder as a text file called 'Zymo_Test'
    file = open("genomes_fasta/Zymo_Test.txt", "w")

    # If the file is gunzipped, then unzip it
    if (file_name.endswith(".gz")):
        fasta_in = gzip.open(file_name, "rt")
    else:
        fasta_in = open(file_name, "r")
    # Re-write every line to the new file so I can open it in Windows
    for line in fasta_in:
        file.write(line)


read_fasta("C:/Users/benja/OneDrive - Aberystwyth University/Desktop/Major Project/ensemble_fungi/er
```

Figure 3.6 – Program that unzips a ‘gunzipped’ FASTA file and rewrites its contents to a readable text file.

In this program, the bottom line is cut short as it's very long, but it calls the 'read_fasta' function on the "Zymoseptoria_tritici_st99ch_3d7_gca_900091695.Zt_ST99CH_3D7.cds.all.fa.gz" file. The function starts by creating a new text file in the 'genomes_fasta' folder, which will hold all the unzipped genome FASTA files eventually. It then uses the 'gzip' module to unzip the contents of the file, and then re-writes those contents to the new text file, since the new text file is readable in Windows.

By combining these 2 small programs together, a program was created that automatically unzips every genome file. This was essential since manually unzipping every genome file would be a tedious task, given there are hundreds of files. However, this program was very slow since it was creating a new text file for every single unzipped genome file, which is unnecessary. It was figured that for the purpose of the program, the contents of the files could just be directly stored in dictionaries and then used directly after. Also, there was no need to store the genome filenames in a text file since this meant they had to be retrieved from that text file. It would be more efficient to store the filenames in an array instead of being written to a text file, since they're being used directly afterwards anyway.

```
import re
import glob
import gzip
import matplotlib.pyplot as plt

home_path = "C:/Users/benja/OneDrive - Aberystwyth University/Desktop/Major Project/"




# Function to return an array of every FASTA and GFF file
def get_file_paths():
    # Locations of directories holding the FASTA and GFF files
    fastas_directory = home_path + "ensemble_fungi/ensemble_fungi/ensembl_fungi_genomes/"
    gffs_directory = home_path + "ensemble_fungi/ensemble_fungi/ensembl_fungi_gff/"

    fasta_paths, gff_paths = [], []

    # Find every file in the FASTA files directory and the GFF files directory.
    # Files must contain '.fa' / '.gff3' and then optionally '.gz' afterward.
    for path in glob.glob(fastas_directory +"*.fa*"):
        fasta_paths.append(path)

    for path in glob.glob(gffs_directory + "*.gff3*"):
        gff_paths.append(path)

    return fasta_paths, gff_paths

```

Figure 3.7 – First part of the new program that unzips and stores the contents of every genome's FASTA and GFF file.

```
# Function to open files that match the names of all the FASTA and GFF paths
def open_files(fasta_paths, gff_paths, amount):

    # Dictionaries to store the path as key and the file's data as value
    fasta_files, gff_files = {}, {}

    # Only open the first 'amount' files
    for path in fasta_paths[0:amount]:
        # If the current file is gunzipped, then unzip it
        if (path.endswith(".gz")):
            file = gzip.open(path, "rt")
        else:
            file = open(path, "rt")

        fasta_files[path] = file.readlines()
        file.close()

    for path in gff_paths[0:amount]:
        if (path.endswith(".gz")):
            file = gzip.open(path, "rt")
        else:
            file = open(path, "rt")

        gff_files[path] = file.readlines()
        file.close()

    return fasta_files, gff_files




fasta_paths, gff_paths = get_file_paths()
fasta_files, gff_files = open_files(fasta_paths, gff_paths, 5)
```

Figure 3.8 – Second part of the new program that unzips and stores the contents of every genome's FASTA and GFF file.

The updated program stores the pathname of every FASTA and GFF file in separate arrays. These 2 arrays are then fed into another function alongside an integer called 'amount' that states how many genome files to open and read. This was essential for assisting testing since it could be set to a small number such as 3 when a small change was made to the code – otherwise the program would open and store hundreds of genomes every time it's re-run, which would take a long time. The function creates a dictionary for the FASTA files and a dictionary for the GFF files. The pathname of the file is the key, and the contents of the file is the value. Then, the first 'amount' files are opened one-by-one. Each file is unzipped, and then its contents are read line-by-line and added to the dictionary.

Below is the introns output text file when 'amount' is set to 3. Each genome has a lot more introns but these have been cut from the screenshot to show that there is intron data for 3 different genomes.

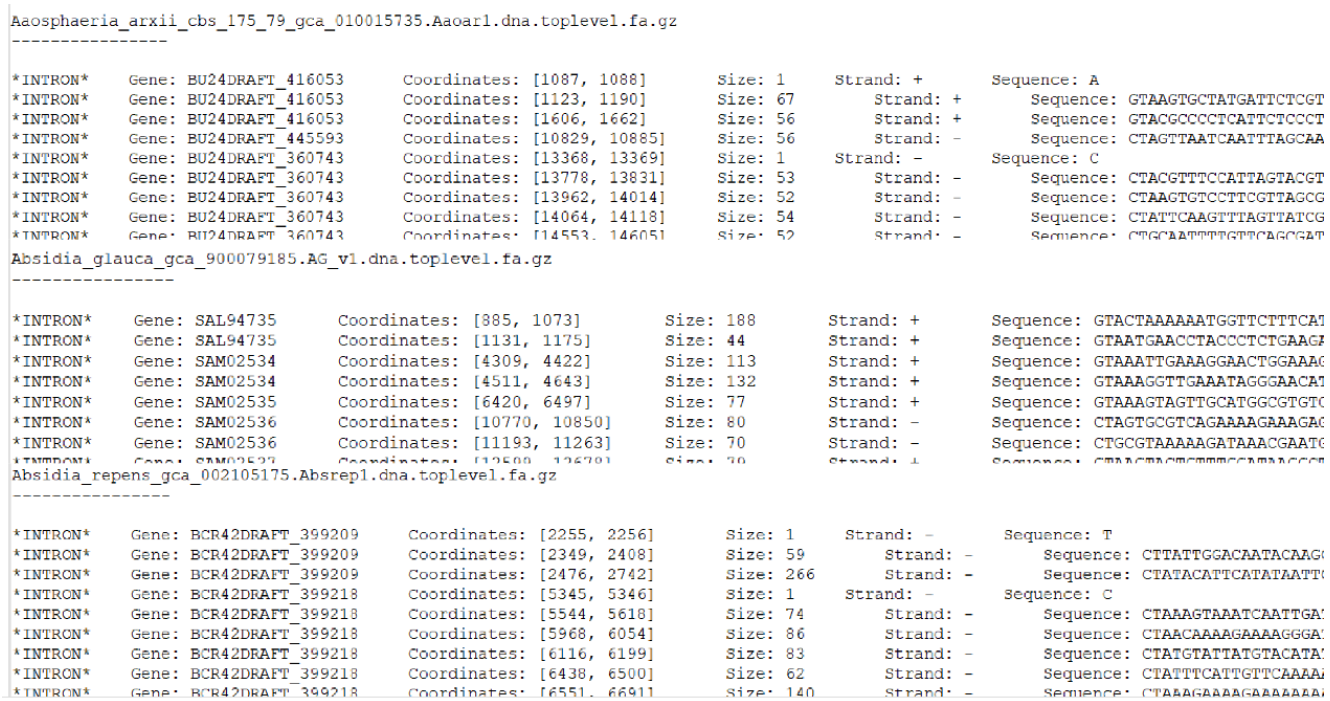


Figure 3.9 – Resulting output text file for first 3 genomes. Note that the issue of intron sequences not matching their respective sizes was partially solved in this screenshot.

3.4. Issue of incomplete intron sequences

The issue of intron sequence lengths not aligning to their respective 'Size' values was investigated. For example, an intron with a size of 120 might have a sequence that is displayed as 'ACGTAA'. The chosen approach was to open the created 'introns' text file from when the program only processed 1 genome, namely the 'Zymoseptoria tritici' genome, and find the first intron where the sequence is incomplete. The first intron with an incomplete sequence was on the 6th line of the text file and was part of the 'ZT3D7_G5' gene. Its 'Size' value is 181, which matches with its coordinates [15045, 14864], but its sequence does not match as it's "CCTGA" which is only 5 nucleobases. It was noticed that the sequence ends with 'TGA' which is a stop codon, meaning the codon that terminates a protein-coding sequence. Introns aren't protein-coding, but this was still found to be interesting.

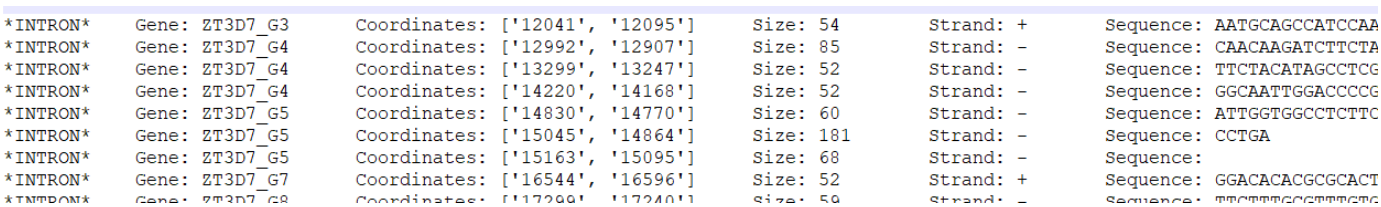


Figure 3.10 – Text file containing intron data for the 'Zymoseptoria tritici' genome. Unlike Figure 3.4, the gene IDs are correct in this screenshot.

Upon looking at the 'ZT3D7_G5' gene in the GFF file, it was noticed than there's an exon that ends at 14864, and another exon that starts at 15045, so the output file is undoubtedly correct that there's an intron between these coordinates.

```
###
1   ena gene    14665   15174   .   -   .   ID=gene:ZT3D7_G5;biotype=protein_coding;gene_id=ZT3D7_G5;logic_name=ena
1   ena mRNA    14665   15174   .   -   .   ID=transcript:SMQ44861;Parent=gene:ZT3D7_G5;biotype=protein_coding;transcr
1   ena exon    14665   14770   .   -   .   Parent=transcript:SMQ44861;Name=SMQ44861-4;constitutive=1;ensembl_end_phas
1   ena CDS 14665   14770   .   -   1   ID=CDS:SMQ44861;Parent=transcript:SMQ44861;protein_id=SMQ44861
1   ena exon    14830   14864   .   -   .   Parent=transcript:SMQ44861;Name=SMQ44861-3;constitutive=1;ensembl_end_phas
1   ena CDS 14830   14864   .   -   0   ID=CDS:SMQ44861;Parent=transcript:SMQ44861;protein_id=SMQ44861
1   ena exon    15045   15095   .   -   .   Parent=transcript:SMQ44861;Name=SMQ44861-2;constitutive=1;ensembl_end_phas
1   ena CDS 15045   15095   .   -   0   ID=CDS:SMQ44861;Parent=transcript:SMQ44861;protein_id=SMQ44861
1   ena exon    15163   15174   .   -   .   Parent=transcript:SMQ44861;Name=SMQ44861-1;constitutive=1;ensembl_end_phas
1   ena CDS 15163   15174   .   -   0   ID=CDS:SMQ44861;Parent=transcript:SMQ44861;protein_id=SMQ44861
###
```

Figure 3.11 – The 'ZT3D7_G5' gene in the GFF file.

This gene was then searched for in the FASTA file and it was noticed that the sequence for this gene is smaller than its supposed size given the gene's coordinates, and that its last 5 nucleobases were indeed "CCTGA". This led to the belief that the rest of the intron sequence is somewhere else within the file, or that there is just an empty region afterwards. The name of this gene 'ZT3D7_G5' was queried for in the FASTA file to see if it contained another sequence elsewhere, but no other instance was found.

```
>SMQ44861 cds chromosome:Zt_ST99CH_3D7:1:14665:15174:-1 gene:ZT3D7_G5 gene_biotyp
ATGCGTTCTCTTGGTAGCTCCTCGATAGACCGGTATACTTGTCAATCAATCGGAGTTGTG
CCCGGTTCACCACCCGGGACGGCTACGCTACTCGAGAAGCAGTTCATTGGTGGCCTCTTC
CTCTCGAGCGTCGAAATCTCAGCCACCGATAAGCTGGATGCCGACAGAACTAGCGACTCT
TTACAGGCTTCCCTCACAGCCTGA
```

Figure 3.12 – The 'ZT3D7_G5' gene in the FASTA file.

After a meeting with the supervisor, it was discovered that this issue was occurring because the supervisor had accidentally shared the wrong type of FASTA files. Each genome has 2 types of FASTA files; one of them is the complete sequence of the genome, and the other is the collection of the coding sequences of the genome combined, aka the genome's sequence after transcription, which is the process that removes introns and non-coding exons. So, the supposedly 'found' introns were just segments of protein-coding exons. This explains why the 'found' introns often contained an empty or incomplete sequence, because genes after transcription are shorter since they only contain the protein-coding exons of the gene. After this realisation, the supervisor shared the FASTA files of the complete genomes containing the sequences before transcription.

These FASTA files have headers for chromosomes (instead of genes). The code was therefore altered so that these files were opened and processed correctly. Since the format for these FASTA files is slightly different, the altered code was verified by only processing one genome. It successfully outputted sequences of the correct size for every intron.

```
def process_fasta(file_name):
    # If the current file is gunzipped, then unzip it
    if (file_name.endswith(".gz")):
        file = gzip.open(file_name, "rt")
    else:
        file = open(file_name, "rt")

    data = file.read().split('>')
    file.close()
    del data[0] # Delete empty string

    chromosomes = {}

    for chrom in data:
        # Chromosome's ID is the first item on the first line
        chrom_id = chrom.split(" ")[0]

        # Chromsome's sequence is everything after the first line
        chrom_sequence = chrom.partition("\n")[2].replace('\n', '')

        chromosomes[chrom_id] = chrom_sequence

    return chromosomes
```

Figure 3.13 – The 'process_fasta' function updated to process the differently formatted FASTA files. It stores each chromosome's ID and sequence as key-value pairs in a dictionary.

3.5. Validating the coding sequences in GFF files

Coding sequences are made up of codons (triplets of nucleobases), with their first codon being the start codon 'ATG' and their last codon being one of the 3 possible stop codons 'TAG', 'TGA', 'TAA'. All genes contain one or more coding sequences. During transcription, the coding sequences of a gene are merged whilst the non-coding sequences are spliced. Since codons are 3 nucleobases, the sum of the coding sequences of a gene should be divisible by 3.

Before code was written to verify this, the GFF file for the 'Zymoseptoria tritici' genome was manually inspected. In this file, the first gene containing multiple exons is the 'ZT3D7_G3' gene. It contains 2 exons which have coordinates (12014, 12041) and (12095, 12291). This means the exon lengths are 28 and 197 respectively. Neither of those lengths are divisible by 3, but their sum is 225 which is divisible by 3. Upon looking at the 'ZT3D7_G3' gene in the FASTA file of coding sequences for this genome, the sequence was found to start with the start codon 'ATG' and end with a stop codon 'TAG', with a length of 225 nucleobases as expected.

A function named 'check_cds_size' was written to automatically verify this for every gene. For each gene, the function iterates through the gene's exons and calculates the difference of the start/end coordinates. It then sums these differences and checks if the sum is divisible by 3 using the modulo operator. If the sum isn't divisible by 3, the function writes an error message to the output file.

```
def check_cds_size(gff_gene):
    intron_file = open("introns_15_march.txt", 'a')
    row_length = 9 # Number of columns in GFF format

    # Counter to add up size of every exon
    cds_size = 0

    # Iterate through every exon
    for index in range((row_length * 2), len(gff_gene), (row_length * 2)):
        # Current exon's start/end coordinates are in columns 4/5
        coords = [int(gff_gene[index + 3]), int(gff_gene[index + 4])]

        # Add size of current exon, inclusive of its start and stop positions
        cds_size += (coords[1] - coords[0]) + 1

    # If coding sequence isn't divisble by 3, there was an issue with splicing
    if ((cds_size % 3) != 0):
        error = "\n\n***ASSEMBLY ERROR*** - CDS of gene is incomplete\n"
        intron_file.write(error)

    intron_file.close()
```

Figure 3.14 – The function that verifies that the sum of a gene's exons is divisible by 3.

At the time the function was wrote, there was a misconception where exons were assumed to always be coding sequences, although this is not true. Coding sequences are always found in exons, but not all exons are protein-coding. Regardless, running the updated program on the 'Zymoseptoria tritici' genome only produced 1 error message in the output file, and it was for the 'ZT3D7_G2228' gene. However, upon looking at this gene in the GFF file, it was found that the sum of its coding sequences is divisible by 3. The reason for the miscalculation was because this gene contains an extra row at the top due to containing an assembly gap. The program assumes that exons are defined every other row starting from the 3rd row, but since this gene contains an extra line at the top, it means the exons are defined starting from the 4th row. Because of this, changes were later made to the way GFF files are processed.

In the 'Zymoseptoria tritici' genome, exons in the GFF file are always defined every other row starting from the 3rd row, apart from the 'ZT3D7_G2228' gene. However, this isn't the case for other genomes. For example, the 'Aaosphaeria arxii' genome has lots of variations. These include exons starting on the 4th row, 2-row gaps between exons, untranslated regions on the last row, more than 4 rows for genes containing only 1 exon, etc.

The updated 'process_gff' function stores each gene as an array with the first item being the first row of the gene (which contains key information about the gene), and all following items being coordinates of exons. Beforehand, each row of the gene was being stored, which is unnecessary since the first row contains all the gene-specific information including its ID, length, strand, and the ID of the chromosome it is on. This information is repeated in the following rows, but all that is needed from the following rows is the coordinates of exons.

The updated function stores gene-information in the first item of the array, then loops through every row of the gene, and if the feature-type of the current row is an exon, the coordinates are added to the gene's array.

```
def process_gff(file_name):
    if (file_name.endswith(".gz")):
        file = gzip.open(file_name, "rt")
    else:
        file = open(file_name, "rt")

    data = file.read().split('###')
    file.close()
    del data[0] # Information before first '###' doesn't represent a DNA sequence
    del data[-1] # Delete empty string at end of file

    genome = []
    row_length = 9 # Number of columns in GFF format

    for gene in data:
        # Split the GFF gene on newline and tab characters
        gene = re.split('\n|\t', gene)
        del gene[0], gene[-1] # Remove the empty strings at the start and end

        # Will store all exons. First item is information about the gene.
        exons = [gene[0:row_length]]

        # Loop through each row of the GFF gene, starting at the 2nd row
        for index in range(row_length, len(gene), row_length):
            if (gene[index + 2] == "exon"):
                # If current row is an exon, add its coordinates to 'exons' array.
                exon_coords = [int(gene[index + 3]), int(gene[index + 4])]
                exons.append(exon_coords)

        genome.append(exons)

    return genome
```

Figure 3.15 – The 'process_gff' function updated to store information more efficiently.

This in-turn simplified the 'find_introns' function since the exon coordinates are found prior to it. The first item of the GFF gene array contains the gene's information, so this is used to gather relevant information about the gene. Then, looping through the gene's exons is much simpler now since they're directly stored in the GFF gene array after the first item.

```
def find_introns(gff_gene, fasta_chromosomes):
    # First item of array is an information-array. All other items are exon coords.
    gene_info = gff_gene[0]

    # Retrieve all relevant information about the gene
    chromosome_id = gene_info[0]
    attributes = gene_info[8].split(';')
    gene_id = attributes[0].split(':')[1]
    strand = gene_info[6]

    # Add this gene information to the introns-information string
    intron_info = "\n\n***GENE***\tID: " + gene_id + "\t\tStrand: " + strand

    # Loop through gene's exons in order to find locations of all introns
    for index in range(2, len(gff_gene)):
        # Intron is from previous exon's end coord to current exon's start coord
        prev_end_coord = gff_gene[index - 1][1]
        current_start_coord = gff_gene[index][0]

        intron_coords = [prev_end_coord, current_start_coord]
        size = (intron_coords[1] - intron_coords[0])
        seq = fasta_chromosomes[chromosome_id][intron_coords[0] : intron_coords[1]]

        intron_info += "\n*INTRON*\tCoordinates: " + str(intron_coords) + "\
\t\tSize: " + str(size) + "\t\tSequence: " + seq

    return intron_info
```

Figure 3.16 – Updated 'find_introns' function, simplified due to the new way of storing genes.

Additionally, the 'check_cds_size' function was simplified to process the newly formatted GFF gene arrays. For each coding sequence, the difference of its coordinates are taken and then 1 is added to that difference. The reason for this is because taking the difference is not inclusive of the start and stop positions. For example, if there was a sequence from positions 3 to 7, then it would contain a nucleobase at positions 3,4,5,6,7 which is 5 positions. But the difference of 7 and 3 is 4, so 1 is added to make it inclusive of the first and last positions.

Also, the function now only opens the introns output file if the coding sequence isn't divisible by 3. Beforehand, it was being opened every time, which is inefficient since the file is only written to if the coding sequences aren't divisible by 3. Upon running this function on the 'Aaosphaeria arxii' genome, there was found to be a lot of instances where coding sequences weren't divisible by 3. This is again because at the time, the function was looping through exon coordinates instead of CDS coordinates, which are not the same since not all exons are protein-coding. This is easily noticeable since a lot of matching CDS rows and EXON rows in GFF files often do not have the same values for coordinates. To fix this issue, the CDS's from genes need to be extracted and stored in another array.

```
def check_cds_size(gff_gene):
    # Counter to add up size of every exon
    cds_size = 0

    # Iterates through every set of exon-coordinates for the current gene
    for index in range(1, len(gff_gene)):
        exon_coords = gff_gene[index]

        # Add size of current exon, inclusive of its start and stop positions
        cds_size += (exon_coords[1] - exon_coords[0]) + 1


    # If coding sequence isn't divisble by 3, there was an issue with splicing
    if ((cds_size % 3) != 0):
        intron_file = open("introns_22_march.txt", 'a')
        error = "\n\n***ASSEMBLY ERROR*** - CDS of gene is incomplete\n"
        intron_file.write(error)
        intron_file.close()
```

Figure 3.17 - Updated 'check_cds_size' function, simplified due to the new way of storing genes.

The 'main_program' function was also changed slightly so that it doesn't open the introns output file for every gene; it only does it once, after all the introns have been extracted from the genes. This is much more efficient since there are so many genes that the file was previously being repeatedly opened and closed many times. After making this change, the overall program ran about 4 times quicker than previously.

```
def main_program():
    # Get a list of the paths of every FASTA and GFF file
    fasta_paths, gff_paths = get_file_paths()

    # Index = how many genomes to run this for
    for index in range(3):
        # Read in the contents of the genome's FASTA and GFF files
        fasta_chromosomes = process_fasta(fasta_paths[index])
        gff_genes = process_gff(gff_paths[index])

        # Find the genome name so can write it to the introns file
        genome_name = fasta_paths[index].split("\\")[1].split(".cds")[0]

        # Will store all introns for genome
        genome_introns = ""

        for gene in gff_genes:
            # Gene contains info, then coords for all exons. So if length is 2, then
            # there is 1 exon meaning no introns.
            if (len(gene) > 2):

                # Retrieve information about all of the introns in this gene
                intron_info = find_introns(gene, fasta_chromosomes)
                genome_introns += intron_info

            # Make sure CDS of gene is divisble by 3
            # check_cds_size(gene)

        intron_file = open("introns_22_march.txt", 'a')
        intron_file.write("\n\n\n\n\n" + genome_name + "\n----------------\n")
        intron_file.write(genome_introns)
        intron_file.close()
```

Figure 3.18 – Updated 'main_program' function, where the output file is only written to once.

Another small change was made shortly after, whereby the introns output file is opened once at the start in 'write' mode and then immediately closed. This clears the file of any content it contained previously, so that it doesn't have to be manually cleared every time the program is re-run.

Finally, the program's ability to check the sum of coding-sequence sizes for genes (instead of the sum of exons) was programmed. Currently, the 'process_gff' function loops through each gene for a certain genome, and for each gene loops through its rows and checks for each row if it's feature-type is an exon. This was altered so that now, if a row is confirmed to not be an exon, it checks if the row's feature-type is a CDS and if it is, it adds the coordinates of the CDS to a coding-sequences array. Once all the gene's rows have been traversed, the array of coding-sequences is appended to a larger array which stores all the arrays of coding-sequences for the genome. Once every gene has been traversed, both the larger coding-sequences array and the array of exon-arrays are returned, instead of just the latter (which is what it did before).

```
# Loop through each row of the GFF gene, starting at the 2nd row
for index in range(row_length, len(gene), row_length):
    if (gene[index + 2] == "exon"):
        # If current row is an exon, add its coordinates to 'exons' array
        exon_coords = [int(gene[index + 3]), int(gene[index + 4])]
        exons.append(exon_coords)
    elif (gene[index + 2] == "CDS"):
        # If current row is a CDS, add its coordinates to 'coding_seqs' array
        cds_coords = [int(gene[index + 3]), int(gene[index + 4])]
        coding_seqs.append(cds_coords)

genome_exons.append(exons)
genome_coding_seqs.append(coding_seqs)
```

Figure 3.19 – Snippet from the updated 'process_gff' function that includes a condition to check if the current row's feature type is a CDS.

```
def check_cds_size(gene_coding_seqs):
    # Counter to add up size of every CDS
    cds_size = 0

    # Iterates through every set of CDS-coordinates for the current gene
    for CDS in gene_coding_seqs:
        # Add size of current CDS, inclusive of its start and stop positions
        cds_size += (CDS[1] - CDS[0]) + 1

    # Default outcome message if an error is not caught
    outcome_message = "\n\n***Assembly Success! - CDS of gene is correct\n"

    # If coding sequence isn't divisble by 3, there was an issue with splicing
    if ((cds_size % 3) != 0):
        outcome_message = "\n\n***ASSEMBLY ERROR*** - CDS of gene is incomplete\n"

    return outcome_message
```

Figure 3.20 – Updated 'check_cds_size' function that correctly calculates the sum of the lengths of a gene's coding sequences.

In the main function, each gene's exon-array and CDS-array are both traversed so that the total CDS size of the gene can be checked directly after it has been searched for introns. When running the updated program on the 'Aaosphaeria arxii' genome, it found 14,276 instances of CDS sizes divisible by 3, and only 71 instances not. This was confirmed by manually calculating the CDS size for the first gene stated as not divisible by 3 and as expected, it was not divisible by 3.

3.6. Inclusivity of intron coordinates

Following began investigation of whether the intron-coordinates being calculated are inclusive of the nucleobase positioned 1 after it. For example, if an intron starts at coordinate 7009, check if the sequence is successfully being output from 7009, or incorrectly from 7008 or 7010, due to Python's

way of indexing. It was realised that a good way of testing this would be by displaying the sequences of exons to see whether they start with the start codon 'ATG' as they should. If the sequence started with a single nucleobase followed by 'ATG', for example 'CATG', it would be clear the previous nucleobase is being displayed. Similarly, if the sequence started with the last 2 nucleobases of 'ATG', for example 'TGCC', it would indicate that the first nucleobase is not being displayed. A function was written to implement this.

```
def output_exons(gff_gene, fasta_chromosomes):

    chrom_id = gff_gene[0][0] # ID of chromosome so know where to look in FASTA file

    # gff_gene[0] is gene-info so start looping after that item
    for exon_coords in gff_gene[1:]:
        seq = fasta_chromosomes[chrom_id][exon_coords[0] : exon_coords[1]]
        print("\n\n\nCoords:", exon_coords, "\nSequence:", seq)
```

Figure 3.21 – Function that displays the sequences of exons. The gene's set of exon coordinates are looped through, and then the sequences of those coordinates are found and displayed.

This was tested for the first gene in the 'Aaosphaeria arxii' genome. Strangely, the sequences never started with anything similar to 'ATG', nor did they end that way. For example, the exon at [774, 1123] starts with "GGCAGCC". This implied that the wrong sequences were being output, but this idea was invalidated by the sequence "GGCAGCC" being searched for in the FASTA file, and the first occurrence of this sequence started at 774. This issue was never resolved, although it didn't negatively impact the project.

```
Coords: [774, 1123]
Sequence:
GGCAGCCAAGCTGAATTGGTTTGCACTGTA
AACTTGTGGGTACCTTGGGGTGGGTACTTT
CCACCTGAAAGGAAAAGAAGCAGAAGCGGC
ATAACAAATGGCTGCACCGGTCAAGACTGT



Coords: [1190, 1606]
Sequence:
CCAGTGGAAATTTGGGAGCTGTACTCCTGG
TTCCAGACTCTTTAAAAGTAGCCTACACCG
CCAAGGTGCCTCACGAACGCATCATTGACG
CCGAGGTCGTGGCGATAACGAAGAAGAAAC
CTGACTTT



Coords: [1662, 2306]
Sequence:
GTGTGGTTGATGGTCTTCTCGGCTTCGACC
GAAAGGCTTTCTACGCGCTCCTCACCAACG
TGGCTACGGTGGAGAACATTCTGGGCGAGA
```

Figure 3.22 – The exons of the first gene in the 'Aaosphaeria arxii' genome.

3.7. Visualisation of intron statistics

Research began into how the Python library 'matplotlib' works. This library is used to display statistics in the form of graphical data, including bar charts, histograms, pie charts, etc. It was decided to investigate the occurrence of the 4 possible nucleobases, 'A', 'C', 'G', and 'T', and display this information in a bar chart.

A new function was added that used 'matplotlib' to generate this. The 'genome_introns' parameter contains the content that is eventually written to the introns output file. It contains lines for every intron found, but also lines stating assembly success/errors. So, in this function, every line of 'genome_introns' is looped through and each line beginning with '*INTRON*' is split to gain its sequence and then that sequence is appended to a string called 'genome_intron_sequence' which will contain all the intron sequences in this genome combined. An array is then created that holds each of the 4 possible nucleobases. These will be the values on the x-axis of the bar chart. Then the occurrence of each of these nucleotides is counted in 'genome_intron_sequence' and the occurrence of each is added to another array. The occurrence of each will be the values on the y-axis of the bar chart.

These values are then plotted onto the bar chart, with each bar being green and the gap between each bar being the same width as a bar. Each axis is then labelled appropriately, and the bar chart is displayed to the screen.

```
def display_graphical_statistics(genome_introns):
    # String to be used to combine all intron sequences together
    genome_intron_sequence = ""

    lines = genome_introns.split("\n")
    for i in range(len(lines)):
        # Each line starting with *INTRON* has an intron sequence at the end
        if (lines[i].startswith("*INTRON*")):
            intron_sequence = lines[i].split("Sequence: ")[1]
            genome_intron_sequence += intron_sequence

    x_markers = ['A', 'C', 'G', 'T']
    # Count the amount of each nucleotide in genome_intron_sequence
    y_values = [genome_intron_sequence.count(nt) for nt in x_markers]

    # Display the data in a readable bar chart
    plt.bar(x_markers, y_values, width=0.5, color="green")

    plt.title("Genome nucleotide frequencies")
    plt.xlabel("Nucleotide")
    plt.ylabel("Occurences in genome")
    plt.show()
```

Figure 3.23 – The 'display_graphical_statistics' function that creates a bar chart to show the occurrence of each nucleobase across the introns of a genome.

Below is the resulting generated bar chart for the occurrence of each nucleobase in the introns of the 'Aaosphaeria arxii' genome. It is apparent that Adenine and Thymine occur roughly 20% more often than Cytosine and Guanine. This makes a low GC content. It was found interesting that

Adenine and Thymine had a near identical occurrence, as did Cytosine and Guanine. This is likely because of how Adenine and Thymine pair up (they are complementary nucleobases), as do Cytosine and Guanine.

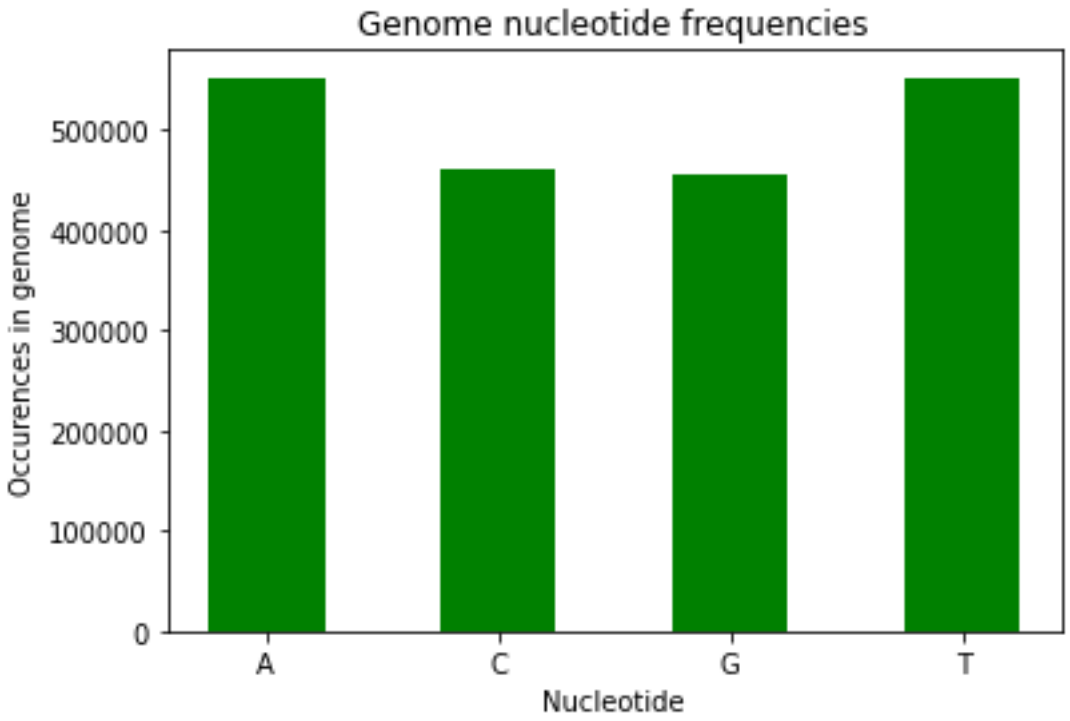


Figure 3.24 – Bar chart showing the occurrence of each nucleotide across the introns of the 'Aaosphaeria arxii' genome.

Additionally, it was decided to gather the average size of introns across the 'Aaosphaeria arxii' genome. This meant plotting the number of times each intron size occurred. For example, the number of introns that are 10 nucleobases long, 11 nucleobases long, 12 nucleobases long, etc. This was achieved by storing the number of times each intron-size occurred, in a dictionary. This was displayed as a histogram at first, and as a subplot, meaning it is displayed next to the previous graph.

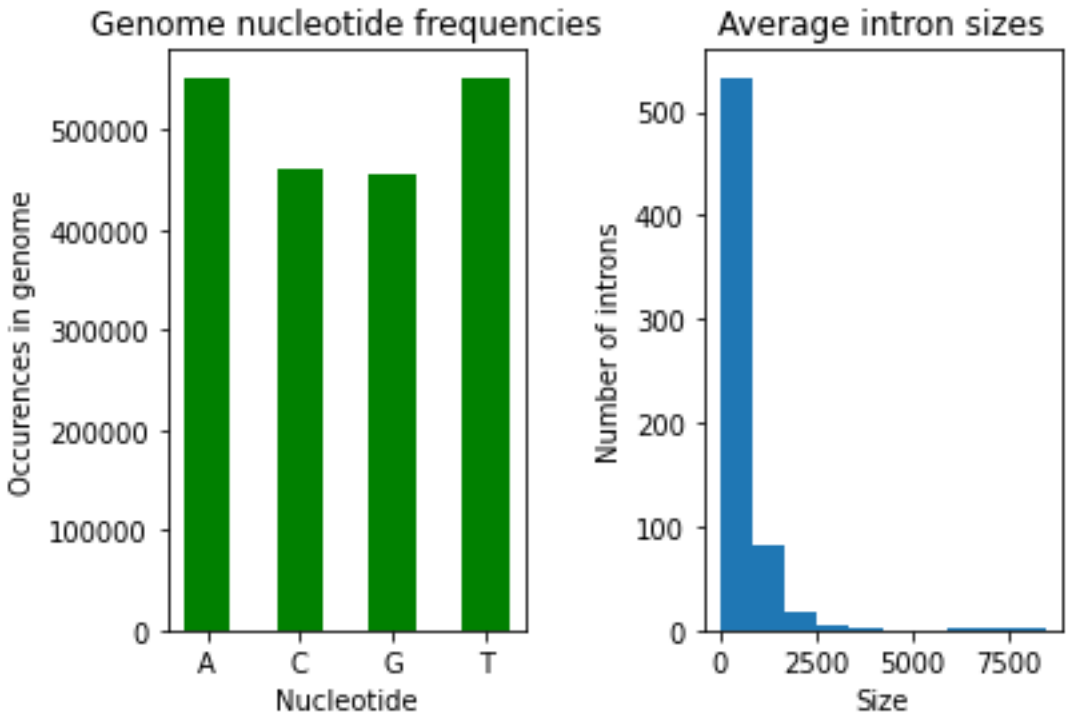
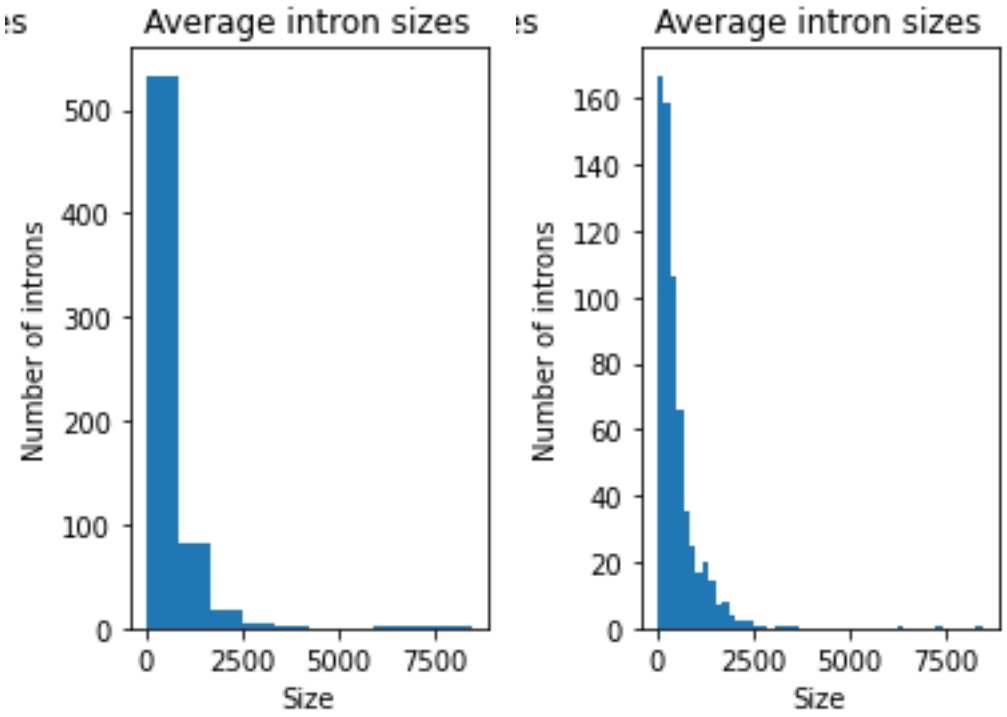


Figure 3.25 – The Bar chart from Figure 3.24 alongside a histogram showing how many times each intron-size occurs in the 'Aaosphaeria arxii' genome.

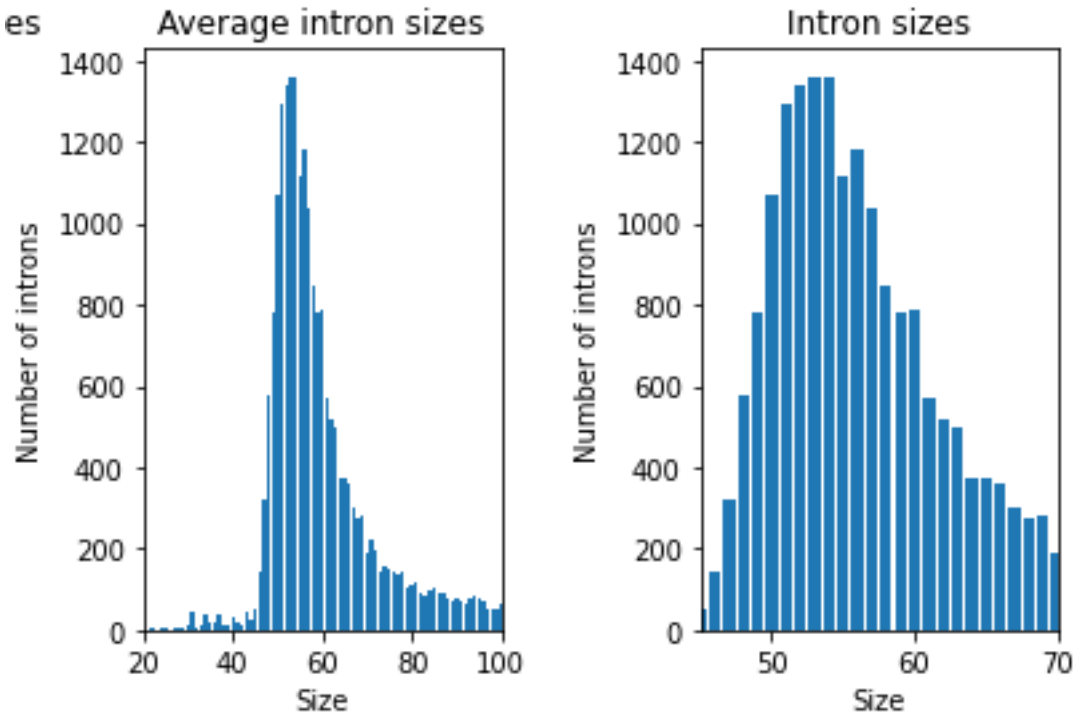
The matplotlib 'hist' method automatically decides the value-ranges for the histogram based off the biggest intron-size, but this was unideal since the biggest intron was about 8000 nucleobases long whereas most introns are less than 100 nucleobases long, so it was not reflective of most introns. The number of bins (histogram bars) can be adjusted to gain more insight into the spread of values.

For example, here is the histogram for 10 bins compared to the histogram for 50 bins. With more bins, more insight can be gained into where specifically the graph peaks.



Figures 3.26 / 3.27 – Comparison of the histogram from Figure 3.25 against a new histogram containing more bins.

This was later changed to be a bar chart like the genome nucleotide frequencies graph, since this allowed manual selection of the range of values. The range was chosen as 20-100, and then 45-70 to find the mode size. The mode size was found to be about 53-54.



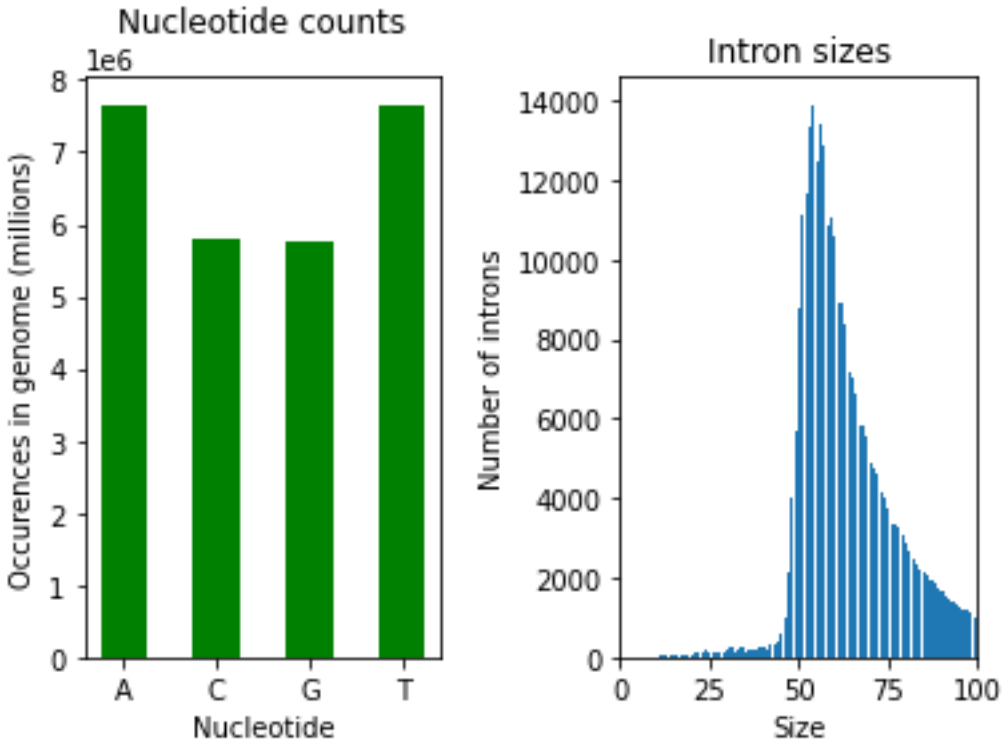
Figures 3.28 / 3.29 – Occurrence of intron-sizes in the 'Aaosphaeria arxii' genome plotted as bar charts, with a range from 20-100 and 45-70.

In the updated version of the 'display_graphical_statistics' function, each intron-line has its size retrieved. This size is then checked to see if it exists as a key in the dictionary containing the intron size occurrences. If it exists, then the value for that key is incremented by 1. Otherwise, it is created as a key and its value is set to 1. This dictionary is then displayed as a plot next to the first graph.

```
def display_graphical_statistics(genome_introns):
    # String to be used to combine all intron sequences together
    genome_intron_sequence = ""
    genome_intron_sizes = {}

    lines = genome_introns.split("\n")
    for i in range(len(lines)):
        # Each line starting with *INTRON* has an intron sequence at the end
        if (lines[i].startswith("*INTRON*")):
            intron_sequence = lines[i].split("Sequence: ")[1]
            genome_intron_sequence += intron_sequence

            intron_size = int(lines[i].split("Size: ")[1].split("\t")[0])
            if (intron_size in genome_intron_sizes):
                genome_intron_sizes[intron_size] += 1
            else:
                genome_intron_sizes[intron_size] = 1



    # ** PLOT 1 **
    # ----------------
    x_markers = ['A', 'C', 'G', 'T']
    y_values = [genome_intron_sequence.count(nt) for nt in x_markers]

    #plt.subplots_adjust(hspace=1)

    plt.subplot(1,2,1)
    plt.bar(x_markers, y_values, width=0.5, color="green")
    plt.title("Nucleotide counts")
    plt.xlabel("Nucleotide")
    plt.ylabel("Occurences in genome")


    # ** PLOT 2 **
    # ----------------
    plt.subplot(1,2,2)
    plt.bar(genome_intron_sizes.keys(), genome_intron_sizes.values())
    plt.xlim([45, 70])

    plt.title("Intron sizes")
    plt.xlabel("Size")
    plt.ylabel("Number of introns")


    plt.subplots_adjust(wspace=0.5) # Stops them overlapping
    plt.show()
```

Figure 3.30 – Updated 'display_graphical_statistics' function that displays bar charts for the nucleotide occurrences as well as the intron-size occurrences.

To get a more representative view of different genomes, the program was run for the first 10 alphabetically named genomes. The limit for intron sizes was chosen as 0-100 since most intron-sizes fall into this range. Below are the results.



Figures 3.31 / 3.32 – Bar charts of the nucleotide occurrences and intron-size occurrences across the introns of the first 10 alphabetically named genomes.

3.8. Reformatting the output file

A call with the supervisor and 2 developers of the FrameRate team led to being given instructions to change the format of the introns output file. Previously, it was formatted in a way that makes it easy to read, but it was suggested by Nick (one of the FrameRate developers) that it would be better to be formatted in FASTA format, whereby each intron of a gene is a header. The code was updated to write the intron data to the file in this format.

Here's a snippet of the first few introns of the 'Aaosphaeria arxii' genome after updating the format of the output. Although the output is in the correct format, each intron is very small.

```
Aaosphaeria_arxii_cbs_175_79_gca_010015735.Aaoar1.dna.toplevel.fa.gz

>BU24DRAFT_416053_intron1
GTAAGTGCTATGATTCTCGTGTCTGTGGCTCTAACGTAGTACGGAAGCTGAAATTGGATGTTTCAGG
>BU24DRAFT_416053_intron2
GTACGCCCCTCATTCTCCCTCTCTCTTTGCAACGCCCTGCTAACATTCGCCCCAGT
>BU24DRAFT_445593_intron1
CTAGTTAATCAATTTAGCAACAATTAGCAGCATTGGATTTAATAGGAGGACAAACA
>BU24DRAFT_360743_intron1
CTACGTTTCCATTAGTACGTATGCTAAACAAACGTGAGATGAAGGTGCTTACT
>BU24DRAFT_360743_intron2
CTAAGTGTCCTTCGTTAGCGAGAGTGTGTTACGGATGCGGTACCATCATACA
>BU24DRAFT_360743_intron3
CTATTCAAGTTTAGTTATCGGTGTTGACAAACCCGAGGAGTATGATGACCTACA
>BU24DRAFT_360743_intron4
CTCCAATTTTGTTCAGCCATACCCGTCACCAGCAGAATATTCTCTTCTTACC
```

Figure 3.33 – Newly formatted intron output file for the 'Aaosphaeria arxii' genome. Each intron is a header.

It was suggested by Nick that only sequences more than 200 nucleobases long should be displayed in the output, since these are long enough to be useful. Also, it was figured that sequences should trail onto newlines after 60 nucleobases to make the file easier to read. The code was updated to meet these new requirements. Below is how the output file appeared.

```
Aaosphaeria_arxii_cbs_175_79_gca_010015735.Aaoar1.dna.toplevel.fa.gz

>BU24DRAFT_469138_intron1
GTAAGTCGGAACGGAGTAGGCCGGGGTTATGAATAACTAACATCAGCATCAGACGCTTCC
AGTCCGATAGGAGTGCTACTAGGCTGGTAATAGGTTAGATTAGATAAGAGGCAGCACCTC
GGCTATACTGTAGCTTGATACGTAGGGGACGATGAATACAAGGGGATCAACGGATATTCT
ATGATAGAATGGAGAGATGTGTCTAGGCCGTTCATGGCTACCCCTTGTACCCCGGTAGTG
TCCTTGTAACTAAAGGTCTCGGGATGTGGGTAGTTCGGCAAGTGGAGGAAATCATTCACT
ATTTATTTCTACGGTGGAAGTAGTTTACACTAGTATAGGTTTATTTTACTTAACGATCTC
GATTAAAAAATGGCACCTCTTACATCGTTTCCGAAGTGCCTATTGATTTGCTGCTCAATT
TTCAGC
>BU24DRAFT_416076_intron1
GTATGTTAGCATTCGCATCCCAATTCCTATTCCTATTGTCCTTCTTGGAGGTGCGGACTT
TGGTAGGGAGGGGTAGCTCCTGGCAGTCTTTCCCCACTCCCTCCGACATCATCGCGAATT
GTTGAAGTTGAGAAAGAACAAATAACAACTCGCGAGGAATTGTACCTTCTCTATGACCAA
TTGCGATGTAGAGATACTGTAAAGACCTCATACTGACTCTGTGACATTGTCCAGC
>BU24DRAFT_403502_intron1
GTATGTACCTCCGTACTCGGTCGCGGTCTGTCATTCTGTGTACAATCATAAATCAATAAT
AATCATAATCAATCCCACCCCTCCCATCCCCTACCCTACCCCACCCCCCCCTCTCCCCTT
```

Figure 3.34 – Intron output file for the 'Aaosphaeria arxii' genome re-formatted again so that sequences carry onto new lines after 60 characters, and only introns longer than 200 nucleobases are displayed.

Here is part of the 'find_introns' function with the code updates. Once the intron sequence is obtained, a new sequence is created that contains every 60 nucleobases of the intron sequence

followed by a newline. This is then added to a string that has a header, made up of the gene ID and a number that identifies how many introns are before this intron.

```
# Loop through gene's exons in order to find locations of all introns
for index in range(2, len(gff_gene)):
    # Intron is from previous exon's end coord to current exon's start coord
    prev_end_coord = gff_gene[index - 1][1]
    current_start_coord = gff_gene[index][0]

    intron_coords = [prev_end_coord, current_start_coord]
    seq = fasta_chromosomes[chromosome_id][intron_coords[0] : intron_coords[1]]

    # Create a new sequence that has a newline at every 60th position
    new_sequence = ""
    for j in range(0, len(seq), 60):
        new_sequence += seq[j : (j + 60)] + "\n"

    intron_info += f">{gene_id}_intron{index - 1}\n{new_sequence}"

return intron_info
```

Figure 3.35 – Snippet of the updated 'find_introns' function that displays sequences in the updated format.

The format of the output file was updated one last time so that the intron sequences for a gene were combined, since not many introns on their own were more than 200 nucleobases. This made it more likely for sequences to be over 200 nucleobases long, which meant there was more training data for FrameRate.

```
Aaosphaeria_arxii_cbs_175_79_gca_010015735.Aaoar1.dna.toplevel.fa.gz

>BU24DRAFT_360743_introns
CTACGTTTCCATTAGTACGTATGCTAAACAAACGTGAGATGAAGGTGCTTACTCTAAGTG
TCCTTCGTTAGCGAGAGTGTGTTACGGATGCGGTACCATCATACACTATTCAAGTTTAGT
TATCGGTGTTGACAAACCCGAGGAGTATGATGACCTACACTGCAATTTTGTTCAGCGATA
GCCGTGAGGACGAGAATATTCTGTTCTTACC
>BU24DRAFT_469138_introns
GTAAGTCGGAACGGAGTAGGCCGGGGTTATGAATAACTAACATCAGCATCAGACGCTTCC
AGTCCGATAGGAGTGCTACTAGGCTGGTAATAGGTTAGATTAGATAAGAGGCAGCACCTC
GGCTATACTGTAGCTTGATACGTAGGGGACGATGAATACAAGGGGATCAACGGATATTCT
ATGATAGAATGGAGAGATGTGTCTAGGCCGTTCATGGCTACCCCTTGTACCCCGGTAGTG
TCCTTGTAACTAAAGGTCTCGGGATGTGGGTAGTTCGGCAAGTGGAGGAAATCATTCACT
ATTTATTTCTACGGTGGAAGTAGTTTACACTAGTATAGGTTTATTTTACTTAACGATCTC
GATTAAAAAATGGCACCTCTTACATCGTTTCCGAAGTGCCTATTGATTTGCTGCTCAATT
TTCAGCGTTTGATCTGCAAAATGCAAAGAATGCGCTGGTGGCCGACAGT
>BU24DRAFT_456447_introns
CTGTTCTTTAGATCAGTATACATCCACTCGCACTTCTCGGAAGTCGTGATACGCACGCTG
CCGACAATATTAGCACACCAGTCCTGGAGTTAGAGTTGGAGTACCTCTCACTCTGCATGC
GCGATTAGTTTATATATCTACGGCTGCCTTGGTCTTCGGACTCGCACCCTAGAGTGGCAT
TAGCAACAGAACTGAAGTAGCATCTCGAGTACCGCCATCCTTACC
>BU24DRAFT_416076_introns
GTATGTTAGCATTCGCATCCCAATTCCTATTCCTATTGTCCTTCTTGGAGGTGCGGACTT
```

Figure 3.36 – Final updated format of the output file – the introns for a gene are now combined.

Here is the updated part of the 'find_introns' function once again. Each intron sequence in the gene is added together into another sequence that holds all the introns. After each intron has been looped through, the program checks if the combined introns sequence is more than 200 nucleobases long, and if it is, it writes the sequence to the file.

```
for index in range(2, len(gff_gene)):
    # Intron is from previous exon's end coord to current exon's start coord
    prev_end_coord = gff_gene[index - 1][1]
    current_start_coord = gff_gene[index][0]

    intron_coords = [prev_end_coord, current_start_coord]
    seq = fasta_chromosomes[chromosome_id][intron_coords[0] : intron_coords[1]]

    combined_intron_seq += seq

# Create a new sequence that has a newline at every 60th position, as long
# as the combined sequence was itleast 200 nucleobases long
if (len(combined_intron_seq) > 200):
    final_sequence = ""
    for j in range(0, len(combined_intron_seq), 60):
        final_sequence += combined_intron_seq[j : (j + 60)] + "\n"

    intron_info += f">{gene_id}_introns\n{final_sequence}"

# If combined sequence was less than 200 bases long, intron_info will be empty
return intron_info
```

Figure 3.37 – Snippet of the final updated 'find_introns' function – each gene's intron sequences are now merged.

The program was run for 100 genomes to get enough intron data. It produced 384,897 intron sequences. This was then sent to Nick who ran it through the 'SwissProt' database (a large database of protein sequence information) using 'diamond blastx' (a command that translates DNA sequences into protein sequences and then searches for these sequences in a protein reference database). Of the 384,897 intron sequences, 2,525 were found to have some level of sequence similarity, meaning a good job had been done of extracting the introns.

4. Testing

A new program was created that imports the intron finding program and tests its robustness. This was done by defining unit tests. Each unit test was defined as a function containing the assert keyword to validate whether the outcome of running an imported function from the intron finding program with some data / file would work when it's meant to work, or not work when it shouldn't work.

This involved creating 'dummy' test data. For example, it was decided to test that the 'process_fasta' function could process any type of FASTA file correctly. This meant manually creating a very small text file in FASTA format. Six chromosomes were created, each containing random DNA sequences. They were all very small, all less than 100 nucleobases long. Some of the chromosomes tested some extreme conditions, for example 'CHR5' only had one nucleobase for its sequence, whereas 'CHR1' contained a multi-line sequence (since the sequence was more than 60 nucleobases long). All chromosomes had 1 extra attribute in the header, except 'CHR6' which had 3.

```
>CHR1 dna:chromosome
CAGTTACCGTATCGTAGTACTAACTCGAACTGATGCTATCAGCATGACTACGTCGACATG
GCATCTCGACGTGCACTACTTCGAGT
>CHR2 dna:chromosome
ACGTCATGCATGACTGATT
>CHR3 dna:chromosome
CGAGCTCATCTCTACTGCAGCTAGCAGTCATCAGTACTATCACTACGTCT
>CHR4 dna:chromosome
CAGTACTTCACTCGTCAGT
>CHR5 dna:chromosome
A
>CHR6 dna:chromosome test1:a test2:b test3:c
ACAGTACTG
```

Figure 4.1 – Dummy FASTA file created to test the 'process_fasta' function.

A unit test was then written that used this data. The test was written to verify that the 'process_fasta' function correctly stores the ID of a chromosome as a key in a dictionary, and the chromosome's sequence with newline characters removed as that key's value. The 'process_fasta' function is run with the dummy FASTA file's contents as its parameter, to return a dictionary of the sequences. Each item in the dictionary is then iterated through and the program uses the 'assert' keyword to check if each key is the correct chromosome ID, by checking if it equals 'CHR' followed by the number that states which chromosome it is. The function then asserts if the key's value contains any newlines. If it does, then it means the 'process_fasta' function didn't successfully remove the newline characters from the DNA sequence. Removal of these is important since when the size of a string is taken, newline characters are included. The function did not return an assertion error, so the file must have been processed correctly.

```python
def test_process_fasta():
    sequences = FIP.process_fasta("dummy_FASTA.txt")

    for index in range(len(sequences)):
        ID = list(sequences.keys())[index]
        sequence = list(sequences.values())[index]

        assert (ID == f"CHR{index + 1}")
        assert (not("\n" in sequence))
```

Figure 4.2 – Function that tests that FASTA files are stored correctly. FIP is an abbreviation for the imported Final Intron Program.

A dummy GFF file was also created. The file-information (part before the first '###') is ignored by the program so this didn't need to be filled out deeply. The 2nd field was defined as 'bec49' for each gene since that's my university ID, and I am the source that created the dummy file. The 9th field (attributes) of the first row was somewhat detailed since the program extracts the gene ID from this field, but the attributes of every other row had no detail since they aren't needed by the program.

Only 3 chromosomes were included, each containing 1-2 genes to simplify testing.

```
##gff-version 3
#!genome-version DUMMY
CHR1    DUMMY   CHR1    1   500 .   .   .   ID=CHR:1
###
CHR1    bec49   gene    8   64  .   +   0   ID=gene:dummy_aaa;biotype=protein_coding;etc;etc;etc
CHR1    bec49   mRNA    8   64  .   +   0   ID=etc;etc;etc;etc
CHR1    bec49   exon    8   64  .   +   0   ID=etc;etc;etc;etc
CHR1    bec49   CDS 8   64  .   +   0   ID=etc;etc;etc;etc
###
CHR1    bec49   gene    145 280 .   +   0   ID=gene:dummy_aab;biotype=protein_coding;etc;etc;etc
CHR1    bec49   mRNA    145 280 .   +   0   ID=etc;etc;etc;etc
CHR1    bec49   exon    145 195 .   +   0   ID=etc;etc;etc;etc
CHR1    bec49   CDS 145 195 .   +   0   ID=etc;etc;etc;etc
CHR1    bec49   exon    211 280 .   +   0   ID=etc;etc;etc;etc
CHR1    bec49   CDS 211 280 .   +   0   ID=etc;etc;etc;etc
###
CHR2    DUMMY   CHR2    1   400 .   .   .   ID=CHR:2
###
CHR2    bec49   gene    145 320 .   +   0   ID=gene:dummy_cce;biotype=protein_coding;etc;etc;etc
CHR2    bec49   mRNA    145 320 .   +   0   ID=etc;etc;etc;etc
CHR2    bec49   exon    145 195 .   +   0   ID=etc;etc;etc;etc
CHR2    bec49   CDS 145 195 .   +   0   ID=etc;etc;etc;etc
CHR2    bec49   exon    211 320 .   +   0   ID=etc;etc;etc;etc
CHR2    bec49   CDS 211 320 .   +   0   ID=etc;etc;etc;etc
###
CHR3    DUMMY   CHR3    1   800 .   .   .   ID=CHR:3
###
CHR3    bec49   gene    16  752 .   +   0   ID=gene:dummy_fao;biotype=protein_coding;etc;etc;etc
CHR3    bec49   mRNA    16  752 .   +   0   ID=etc;etc;etc;etc
CHR3    bec49   exon    16  109 .   +   0   ID=etc;etc;etc;etc
CHR3    bec49   CDS 16  109 .   +   0   ID=etc;etc;etc;etc
CHR3    bec49   exon    344 390 .   +   0   ID=etc;etc;etc;etc
CHR3    bec49   CDS 344 390 .   +   0   ID=etc;etc;etc;etc
CHR3    bec49   exon    506 752 .   +   0   ID=etc;etc;etc;etc
CHR3    bec49   CDS 506 752 .   +   0   ID=etc;etc;etc;etc
###
```

Figure 4.3 – Dummy GFF file created to test that GFF files are processed correctly.

A test was written that loads this file using 'process_gff' and then uses 'check_cds_size' on the first 2 genes. The first gene should lead to an assembly success message, since it only contains 1 CDS which has a size of (64 – 8) + 1 = 57 which is divisible by 3. The second gene should lead to an assembly error message, since the sum of its CDS's is ((195 – 145) + 1) + ((280 – 211) + 1) = 121 which is not divisible by 3. The function finally asserts that the length of the 'coding_seqs' is array is 4, since there are 4 genes in the GFF file.

```
def test_check_cds_size():
    genes = FIP.process_gff("dummy_GFF.txt")
    coding_seqs = genes[1]

    gene_1_coding_seqs = coding_seqs[0] # Sum of CDS's is divisble by 3
    gene_2_coding_seqs = coding_seqs[1] # Sum of CDS's isn't divisible by 3

    assert "Success" in FIP.check_cds_size(gene_1_coding_seqs)
    assert "ERROR" in FIP.check_cds_size(gene_2_coding_seqs)
    assert len(coding_seqs) == 4
```

Figure 4.4 – Function that tests that the sum of coding sequences for a gene are calculated correctly.

This function resulted in one assertion error, namely the 3rd assertion, the one that tests that the length of the 'coding_seqs 'array is 4 (since there are 4 genes in the file).

```
Reloaded modules: final_intron_finder
Traceback (most recent call last):

  File "C:\Users\benja\OneDrive - Aberystwyth University\Desktop\Major Proje
line 49, in <module>
    test_check_cds_size()

  File "C:\Users\benja\OneDrive - Aberystwyth University\Desktop\Major Proje
line 40, in test_check_cds_size
    assert len(coding_seqs) == 4

AssertionError
```

Figure 4.5 – Assertion error from the 'test_check_cds_size' function.

This was investigated and it was discovered that there were 2 additional items which were both empty lists. This was found to be because the program assumes everything between triplets of hashtags is a gene, but there is an exception to this – GFF files are hundreds of thousands of lines long, but there are usually only a tiny number of chromosomes, but when the genes for a new chromosome start, the chromosome's row for its ID and information is encapsulated by triplets of hashtags. This wasn't picked up upon earlier, but it hasn't resulted in any errors since running any of the functions in the main program with an empty item is failsafe.

Another function was written to verify that the 'process_gff' function correctly stores the CDS coordinates for a gene as well as its exons. In the dummy GFF file, each set of CDS coordinates are identical to the exon-coordinates above them. So, these sets of coordinates should be the same for every gene's exons. This was tested by looping through the file's exons and comparing the coding sequence coordinates at that index to the exon coordinates. No assertion errors were returned.

```
def test_process_gff():
    genes = FIP.process_gff("dummy_GFF.txt")

    exons = genes[0]
    coding_seqs = genes[1]

    for index in range(len(exons)):
        # Everything after the first item is sets of exon-coordinates
        exon_coords = exons[index][1:]

        assert exon_coords == coding_seqs[index]
```

Figure 4.6 – Function that tests that GFF files are processed correctly.

A final unit test was written to confirm that the 'output_exons' function worked correctly. It should be able use the exon coordinates from a gene to find the right sequence in the FASTA file. The dummy files didn't completely align since the FASTA file had 3 additional chromosomes, and each sequence was shorter than its supposed size in the GFF file, but this didn't matter. For CHR1, the first exon in the first gene has coordinates [8,64], and CHR1 in the FASTA file is longer than 64 nucleobases so this could be tested. However, the 'assert' keyword could not be used since the 'output_exons' function does not return any data; it outputs the data to the terminal. Regardless, this was tested by checking the output of the terminal.

```
def test_output_exons():
    sequences = FIP.process_fasta("dummy_FASTA.txt")
    genes = FIP.process_gff("dummy_GFF.txt")

    gene_1 = genes[0][0]
    FIP.output_exons(gene_1, sequences)
```

Figure 4.7 – Function that tests that exons can be found from the correct place in a FASTA file from GFF coordinates.

It output the correct coordinates [8,64] and the sequence "GTATCGTAGTACTAACTCGAACTGATGCTATCAGCATGACTACGTCGACATGGCAT", which was manually checked in the FASTA file and found to be the sequence in CHR1 from index 8 to index 64.

```
Coords: [8, 64]
Sequence: GTATCGTAGTACTAACTCGAACTGATGCTATCAGCATGACTACGTCGACATGGCAT

In [50]:
```

Figure 4.8 – The terminal output of the function from Figure 4.7.

5. Critical Evaluation

5.1. Strengths

The approach of using a 7-day iteration accompanied by a blog was a very good way of doing this project since it gave me a new set of tasks to work on each week and helped me assess what went well and what issues had been encountered, so I'd be able to discuss them with my supervisor the following week. The use of a Kanban board also proved very useful to me. It allowed me to keep all of the active tasks in one small place and move them around when necessary. Also seeing how many tasks had been completed in the project gave motivation.

GitLab also proved useful. This is not just because of the issues board it contains (which I used as a Kanban board), but also how it lists all the commits I made, as well as the dates I made them. Additionally, the fact it supports formatting for viewing PDF and Python files made it simple for my supervisor to see what I had done without having to download the files. This was extremely useful for times when I didn't bring my laptop to weekly meetings. Regularly backing up my data on this site provided comfort knowing I had a backup.

Overall Python was a good choice for a programming language since it's good for processing lots of data quickly, which my project was heavily revolved around. It's also very easy to write and debug code in Python, which made implementing difficult logic easier. I believe my code was very successful since it is able to extract intron information from different formatted GFF files of hundreds of different genomes, and also was validated to be correct when Nick ran the intron data through the protein database.

5.2. Weaknesses

The final program didn't run as fast as I'd have liked. Although improvements were made to the design of the code that increased the speed of the program, it could have been programmed more efficiently, perhaps by using more efficient data structures such as hash maps. Also, there were some loops where list comprehension could have been used as an alternative, which may have slightly improved speed.

I also wish I had investigated the use of alternative splicing in my program. This involves the different combinations that exons from a gene can be recombined after splicing. At the time however, I figured that this was slightly irrelevant to the project given it doesn't involve introns.

Also, the intron data was sent to the FrameRate developers very late in the project, which gave not much time for them to run it through the protein database and deliver results to me. There were commands I could have performed myself to test the data I had gathered, but I didn't leave myself enough time to do this in depth to produce any substantial results – I focused more on delivering the code in the most efficient manner than running tests on the data it generated.

One of the major setbacks was how I was unaware of the fact I was processing the wrong types of FASTA files for the first few weeks of the project. I was looking for introns in the FASTA files of coding sequences of a genome (the genome after transcription), meaning there were no introns in these files. If I'd had recognised this issue earlier on, I'd have fixed some of the related issues much quicker, for example intron sequences being incomplete / empty.

Furthermore, the function for displaying statistical graphs about the intron data does not work unless the content of the output file is formatted in a certain way. In the old format of the output file, introns were indicated by a line starting with *INTRON*, whereas other lines were often empty or contained error messages. At the time of writing the statistics for introns function, the output file was still formatted in this way, so the function found introns by checking if each line started with *INTRON*. But since the new format is now in FASTA format, the function doesn't detect any line as being an intron since the text '*INTRON*' is not found anywhere in the output file. The code for formatting the output file in the old way is still in the program but is commented out. To provide somewhat of a solution to this issue, I provided instructions in comments at the top of the program that state which lines to comment out and which lines to uncomment out, to get the program to generate bar charts. This problem could have been approached and solved in a much better way.

5.3. Conclusion

The final program extracted the introns from 100 different Fungi genomes. This seemed enough to me given how quickly the resulting output file builds up in size. If I were to do it again, I may have done a few hundred more genomes, or better yet genomes of a different kind such as the human genome, since I only processed Fungi genomes.

A big challenge for this project was because of how much it was based around biology. I only started learning about the links of biology and computer science last semester, so the jump from computer to biology was quite big and made various parts of this project challenging, for example getting my terminology right and understanding how to make sense of a GFF file, especially when they're formatted differently for different genomes and include things I'd never heard of previously, such as assembly gaps and UTRs.

Overall, the project went quite smoothly. The final program had many features, including the ability to extract intron coordinates from a GFF file based off exon coordinates, and then find the sequence that matches these coordinates in the same genome's FASTA file, and display all this data. It was also able to process multiple genomes of different formats and display statistical information about the intron data.

References

[1] Matplotlib: Visualisation with Python. https://matplotlib.org/

The Python package I used for generating statistics graphs for the introns.

[2] GitLab: The DevSecOps Platform. https://about.gitlab.com/

The version control system that I used to upload all my code and documents. It contains all my work and progress.

[3] Spyder: The Scientific Python Development Environment. https://www.spyder-ide.org/

The IDE I used for all the programming.

[4] Python: https://www.python.org/

The programming language I used. Note that this includes the 're', 'glob', and 'gzip' modules used since these are built-in packages.

[5] "Not all exons are protein coding: Addressing a common misconception": https://www.cell.com/cell-genomics/pdf/S2666-979X(23)00062-9.pdf

A paper I read that discussed how exons are the parts of DNA that are transcribed to RNA. They are often protein-coding, but not always.

[6] Intron: National Human Genome Research Institute - https://www.genome.gov/genetics-glossary/Intron#:~:text=An%20intron%20is%20a%20region,consist%20of%20exons%20and%20introns

A brief definition of what are introns are and their purpose.

[7] Microsoft Word: https://www.microsoft.com/en-ww/microsoft-365/word?activetab=tabs%3afaqheaderregion3

The word-processing software I used to write the report.

[8] How to use Glob() function to find files recursively in Python?: https://www.geeksforgeeks.org/how-to-use-glob-function-to-find-files-recursively-in-python/

A site I used to educate me on how to use the Python 'glob' module.

[9] Matplotlib Tutorial: https://www.w3schools.com/python/matplotlib_intro.asp

The site I used to learn about Matplotlib and how to use it.

[10] Ensemble: https://www.ensembl.org/index.html

The site I used to download Fungi genomes from.