Codano: A Tool for Comparing Codon Sites Frequencies

Michael Kelly
mpk0010@uah.edu
University of Alabama at Huntsville

Abstract. Oftentimes, genetic research calls for the comparison of codons at sites of interest in genomes coming from wild and mutant type samples. These situations typically call for millions of DNA sequences to be parsed and post-processed to find statistical trends. In the age of emerging software technologies, scientists need not worry about solving this daunting task by hand. There are plenty of biotechnology applications available to researchers that offer an impressive suite of tools. Many of these softwares, however, are either locked behind a large pay wall or have a steep learning curve. For those on a tight budget or only need a basic tool to compute codon frequencies, multipurpose software may not be the best solution. Codano alleviates both of these problems by providing a free and simplistic coding interface that allows users to parse large genetic datasets for codon comparisons the most efficient way possible. This paper will cover the fundamentals of each function in the interface at a surface level to promote its application in other fields.

1. Introduction

Codano is a tool that compares DNA sequences from different sample types and calculates the frequency of commonly occurring codons at user defined sites. This software was initially created for use in a research project at the University of Alabama in Huntsville involving the analysis of codon frequencies between wild and mutant type strains of *E-coli* bacteria. Millions of coding strand DNA sequences from these types were provided in the form of fastq files (roughly 30 gigabytes of data). These sequences were on average 25 to 30 nucleotide base pairs long. Of these input coding strand sequences, only 3 codon sites were of concern, designated as E, P, and A. These sites were defined to start 12, 15, and 18 nucleotides upstream respectively from the 3' end (***Figure 1.1)***.

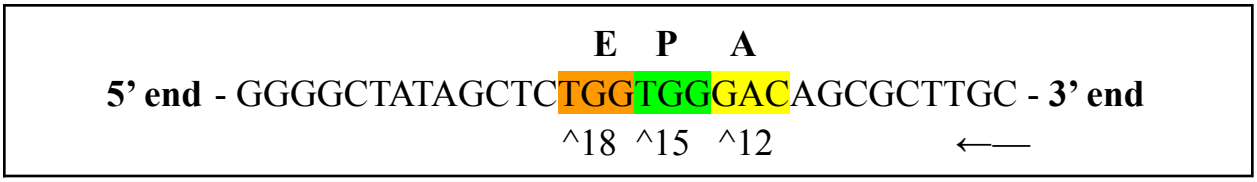

Figure 1.1

Codano was implemented to find the ratio of codon occurrences between two sample types (wild/mutant) at predefined sites. Additionally, the tool created plots for each site that further showcased the differences of codon appearances across both sample types. Aside from examining the frequency of codons at target sites in *E. coli*, this tool could be applied to any organism so long as two sample types of genetic data exist. Codano's new vision is to serve as a tool for researchers in related projects outside of its initial use case. This document takes the first step in the initiative by explaining Codano's coding interface in easily digestible modules to encourage its reuse in other applications of gene research.

2. Background

This paper assumes that the reader already has a solid understanding of fundamental genetic theory and some exposure to programming. For those who may not be well versed in these areas, this section will briefly cover key details needed to understand this document.

The following paragraph provides surface level context to genetics related terminology discussed throughout the paper. *E. coli* is a type of bacterium commonly found inside the intestines of warm blooded animals. DNA is a cellular molecule that contains genetic information. DNA is helical in structure and consists of two strands connected by matching sub molecules called nucleotides. These two strands run antiparallel to one another, meaning that one is a mirror reflection of the other. One of these strands is called the coding strand, which is said to be oriented in a 5' to 3' direction. Its counterpart, the template stand, is oriented in a 3' to 5' prime direction. When a DNA molecule is read for its genetic content, the information is ultimately derived from the coding strand. These nucleotides are read in triplet groups called codons. Each codon corresponds to a specific amino acid. Amino acids are the building blocks of proteins. Proteins perform the instructions outlined in DNA. Amino acids are chained together over a series of processes to create a protein. Ribosomal DNA is a special type of DNA that creates ribosomal RNA rather than proteins. Ribosomal RNA molecules form together to create a cellular component called the ribosome. Ribosomes are what facilitate the chaining of amino acids to create proteins. The term "wild type" refers to commonly occurring DNA sequences seen in an organism group, while "mutant type" refers to atypical sequences.

The following paragraph provides surface level context to computer science related terminology discussed throughout the paper. Python is a popular programming language used in software development. Anaconda is a specific version of Python that is often used in scientific settings. Anaconda supports a variety of external tools called packages. The packages used in Codano are biopython, seaborn, matplotlib, pandas, and tqdm. Biopython offers a suite of features that allow large genomic datasets to be unpacked. Seaborn is a package that converts data into a variety of graph types for visual representation. Matplotlib is also a graphing package that was used in

conjunction with seaborn. Pandas is a package that allows for the creation and manipulation of large dataframes. Dataframes are digital structures that can store large amounts of information (think of them as large spreadsheets). Tdqm provides a simple loading bar to gauge the progress of heavy computations in the program. More details on each package can be found at their respective websites (provided on the bibliography page). Large blocks of code logic dedicated to performing a singular task are called functions. Functions can be directly provided input data to perform their operation. These inputs are called parameters. There are two data structures commonly mentioned in this paper: lists and dictionaries. A list can contain data types (integers, decimals/floats, characters, etc) within a self-contained linear grouping. A dictionary stores information as multiple sets of key:value pairs. A key is used as an identifier to find its associated value. Think of this data structure as a mail truck that contains multiple packages (sets) addressed to specific recipients (keys) that vary in item contents (values).

With this knowledge in mind, we can now discuss how Codano works on a surface level. The software logic can be broken down into three major components: data extraction, data processing, and data output. Without going into too much detail, the data extraction phase involves locating files that contain genetic datasets and accessing their contents for sequence retrieval. The data processing stage serves as a bridge between the extraction and output phases. This stage is responsible for analyzing target codon-sites by parsing the incoming coding-strands from the extraction phase. Once all sequences have been read, the frequencies of codons seen at each site between two sample types (i.e. wild and mutant) are computed. Next, the collected analytics undergo a series of structural changes prior to the data output phase. Once these changes are made, three types of results are exported: a dataframe containing records of each sequence read, a dataframe showcasing codon-site frequencies, and several graphs that highlight the trend of codons seen across each sample type. More details on these results are discussed later in the paper. The next section will provide a closer look at Codano's data extraction phase.

3. Solution: Data Extraction

The Codano software begins by extracting sequences from genetic datasets provided by the user. This phase takes the longest to execute, as all sequences must be read in linear/sequential order, making computation time dependent on input size. Currently, Codano only accepts genetic datasets in the form of fastq files. For the script to work properly, it must be in the same directory/folder as the fastq files. These fastq files are expected to contain coding DNA strands only (RNA sequences are currently ***NOT*** supported). All sequences must only contain the standard base pairs A, T, G, and C. Ambiguous reads are also allowed so long as they are denoted as N in the sequence reads. These ambiguous reads are identified in the code logic and will ***NOT*** be included in the codon-site analysis (ensures only valid codons are considered). Lastly, the input data sets must be separated into two distinct sample type categories. For

example, the *E. coli* research this software was designed for provided six fastq files containing coding strand data: three denoted as wild type and three denoted as mutant type. Each sequence only contained standard nucleotide base pairs as well as occasional ambiguous reads. These files were all located within the same directory as the Codano script. With the input requirements properly defined, we can now discuss the code logic behind this phase. There are two core functions in this stage, both being separated by an outer and inner loop. The following subsection will provide more insight on the former.

Outer Read Loop

The outer read loop is facilitated by a function called extract_files. This function does not require any input parameters, and should be called at the very start of the program. It is responsible for finding fastq files in the script directory and compiling all of the codon-site observations. The function starts by creating an empty data frame to store the codon-site data gathered from each inner loop iteration. Once the empty dataframe is created, the function searches for a fastq file in the script directory and passes its name as a parameter into a function called read_file. This function is what initiates the inner loop logic, which will be described in the next subsection. Once the current file has been read, its codon-site results are then appended to the total dataframe. This sequence of events completes one iteration of the outer loop. This loop is repeated again for every fastq file found within the script directory. Once all fastq files have been read, the total results are saved as a csv file in the script directory and returned to the main function. ***Figure 1.2*** provides a general overview of the code logic involved in the outer read loop.

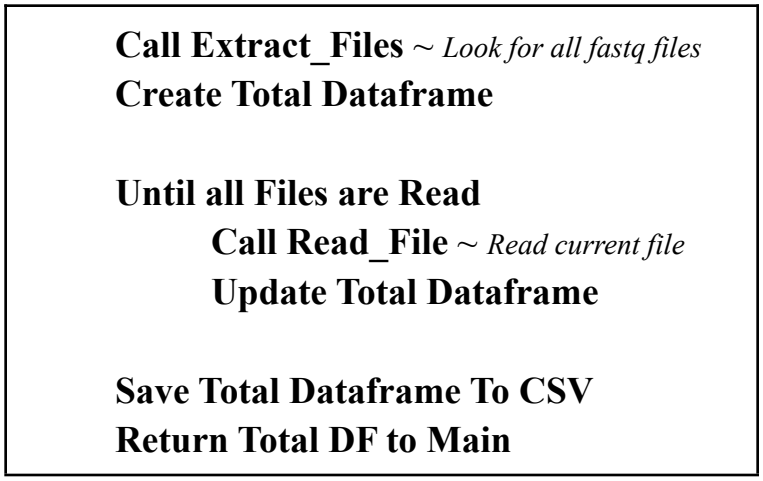

Figure 1.2

Inner Read Loop

The inner read loop is facilitated by a function called read_file. This function is responsible for parsing every sequence in a file for codon-sites. Read_file begins by creating a list of dictionaries, each dictionary representing a codon site. Within each site-dictionary, the observed codon serves as a key, while the number of its occurrences serves as a value. More details on this concept will be provided momentarily. Each sequence parsed from the file is passed into a function called track_codons. Track_codons reads user-defined codon-sites x nucleotides upstream from the 3' prime end. Using ***Figure 1.1*** as an example, to start the codon readings at site A, the user would need to specify x nucleotides as 9. This is because Codano aligns its reading frame based on the first nucleotide encountered upstream from the 3' prime end. In this case, the 9th nucleotide upstream is C. Once the reading-frame is aligned, it will identify codon site A as GAC, starting on nucleotide 12 from the 3' prime end. The user must also specify to track_codons how many codons will be read. In the case of ***Figure 1.1***, this value would be 3, as only sites E, P, and A are considered. Based on these specifications, track_codons scans its input sequence for codons seen at each user defined site. The results of these scans are saved into the list of dictionaries created at the beginning of read_file. ***Figure 1.3*** highlights the inner loop process as well as the results produced by track_codons. In the provided figure, a table denoted as *Site-List* represents the list of dictionaries. Each column of the table represents a site-dictionary. After scanning the DNA sequence from ***Figure 1.1***, each site-dictionary is updated accordingly based on the codon seen. This process repeats for every sequence scanned from the current file being read. It is important to note that the same site-list is updated each sequence scan. For example, if 1,000 sequences were scanned and 500 AAA codons were found at site E, the AAA entry of site-dictionary E would be updated to **AAA : 500** accordingly. Once every sequence in the file has been scanned for codon-sites, the site-list is converted to a dataframe and returned to the outer read loop for result merging. This concludes the file read process. Next section will discuss the subsequent data processing phase in more detail.

Create List of Dictionaries ~ Denoted as Site-List
Site-List Before =

	Site E	Site P	Site A
Codon : # Seen	**N/A : N/A**	**N/A : N/A**	**N/A : N/A**

For Every Sequence in Current File
Call Track_Codons
Site-List Updated

Site-List After =

	Site E	Site P	Site A
Codon : # Seen	**TGG : 1**	**TGG : 1**	**GAC : 1**

Convert Site-List to Dataframe
Return Site-List to Extract_File

Figure 1.3

4. Solution: Data Processing

Following data extraction, the acquired analytics is post-processed to compute codon-site frequencies. This is the fastest stage of the program, lasting only a handful of seconds. This phase begins by creating two copies of the total results dataframe produced from the initial data extraction. These two copies are then filtered to only contain entries from either wild or mutant type reads (2 x total dataframe copies -> 1 x wild dataframe, 1 x mutant dataframe). This filtration process is handled by the pandas package, which is designed to perform quick dataframe manipulations. Once the readings are separated by samples, the codon-site occurrences are summed up across each file representing both types. This process is handled by a function called get_site_occurrences. More information on this specific process will be discussed in the upcoming subsection. Once the total codon-site occurrences for each sample are gathered, data processing concludes with calculating the frequencies across both types. This is handled by a function called compute_site_frequencies. More on this will also be covered in an upcoming subsection. After processing is complete, the resulting analytics are packaged and exported in the subsequent data output phase.

Computing Codon-Site Occurrences For Each Sample Type

The operation of computing codon-site occurrences for each sample type is conducted by a function called get_site_occurrences. This function requires a filtered dataframe containing only wild or sample read entries (these dataframes were created at the start of the data processing phase). The goal of this function is to sum the codon-site occurrences across all files representing the input sample type. For example, let us assume a filtered dataframe containing only wild type data is passed into get_site_occurrences. Let us also assume that there were three fastq files providing wild type datasets (similar to the *E. coli* research). Each wild type fastq file has codon-site occurrences for its own sequences. To determine the ***total*** wild type codon-site occurrences, these individual readings must be merged/summed together. ***Figure 1.5*** gives a visualization of this concept. ***Figure 1.4*** provides the high level code logic for this function. These summed results are stored in a site-list (similar to the list of dictionaries seen in the data extraction phase) and returned to the main function.

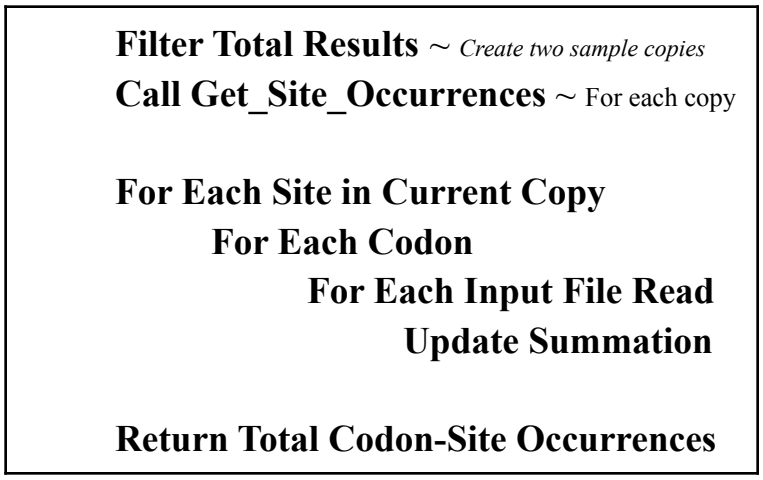

Figure 1.4

Fastq Files Provided:

Wild1.fastq	Wild2.fastq	Wild3.fastq	Mutant1.fastq	Mutant2.fastq	Mutant3.fastq

Total Results Post Data Extraction:

Wild1	Wild2	Wild3	Mutant1	Mutant2	Mutant3
Site E : AAA : 456	**Site E : AAA : 321**	**Site E : AAA : 407**	**Site E : AAA : 781**	**Site E : AAA : 542**	**Site E : AAA : 601**
Site P : AAC : 237	**Site P : AAC : 341**	**Site P : AAC : 412**	**Site P : AAC : 435**	**Site P : AAC : 245**	**Site P : AAC : 324**
. .	. .	. .	. .	. .	. .

Filtering Total Results at Data Processing Start:

Wild Dataframe	Mutant Dataframe
Wild1 : Site E : AAA : 456	**Mutant1 : Site E : AAA : 781**
Wild2 : Site E : AAA : 321	**Mutant2 : Site E : AAA : 542**
Wild3 : Site E : AAA : 407	**Mutant3 : Site E : AAA : 601**
. .	. .

Passing Wild Dataframe to get_site_occurrences:

Wild Dataframe
Wild1 : Site E : AAA : 456
Wild2 : Site E : AAA : 321
Wild3 : Site E : AAA : 407
. .

Results Returned From get_site_occurrences:

Total Wild Type Codon-Site Occurrences (Site-List)
E Site-Dictionary: {AAA : 1184, …}
P Site-Dictionary : {AAC : 990, …}
. .

Figure 1.5

Computing Codon-Site Frequencies

Once the total occurrences are acquired for both samples, computing the frequency of specific codon-site combinations between wild/mutant types becomes quite trivial. This process is handled by a function called compute_site_frequencies. The inputs required for this function are the total codon-site occurrences from both sample types previously generated by get_site_occurences. This function begins by creating an empty site-list (list of dictionaries) to later store the resultant frequencies. Compue_site_frequencies then loops through every target codon-site and for each codon in all sites. The computed frequency equals one sample's codon occurrence at a site over the other sample's codon occurrence at the same site. Once calculated, these frequencies are saved in the site-list and returned upon function completion.

The order of frequency division is user configurable by rearranging the function's parameters:

- compute_site_frequencies(Total Sample1, Total Sample2)
- Frequency of Codon at Site = (Sample1 Occurrences)/(Sample2 Occurrences)

  OR
- compute_site_frequencies(Total Sample2, Total Sample1)
- Frequency of Codon at Site = (Sample2 Occurrences)/(Sample1 Occurrences)

In regards to the *E. coli* research, the frequency was defined as the following:

- compute_site_frequencies(Total Wild Occurrences, Total Mutant Occurrences)
- Frequency of Codon at Site = (Wild Occurrences)/(Mutant Occurrences)

Figure 1.6 provides an example of how a codon-site frequency is calculated with wild type as the first parameter, and mutant type as the second parameter.

Type	Codon	Site	Occurrences
Wild	AAA	E	1,184
Mutant	AAA	E	1,924

Codon Frequency of E = (Wild Occurrences)/(Mutant Occurrences) = (1,184)/(1,924) = 0.6154

Figure 1.6

Figure 1.7 provides an overview of the code logic behind the compute_site_frequencies function.

```
Acquire Total Sample Occurrences
Create Empty Site-List

For Each Site
        For Each Codon
                Freq = (Sample1/Sample2)
                Save Freq to Site-List

Return Site-List
```

Figure 1.7

This concludes the data processing phase. After all computations are complete, Codano enters its final stage where all analytics are reformatted for easier viewing and exported as output files. The next section will explain this phase in greater detail.

5. Solution: Data Output

The Codano software ends by reformatting and exporting all results from major computations. These results include two graphs for **each** target site that highlight codon occurrences between sample types, a dataframe containing all codon-site frequencies, and a dataframe containing sequence information collected from fastq file reads. This phase begins by reformatting the frequency site-list produced from the previous data processing stage. Recall that this site-list is a list of dictionaries, with each dictionary representing a site, and each dictionary entry containing a codon key and a frequency value. The reformatting process is handled by a function called frequencies_to_dictionaries. This function takes a frequency site-list as a parameter and returns a dataframe of the input data. This dataframe contains three columns: one for codon entries, one for site entries, and one for frequency entries. The conversion process is fairly trivial code wise. ***Figure 1.8*** showcases the logic behind frequencies_to_dictionaries.

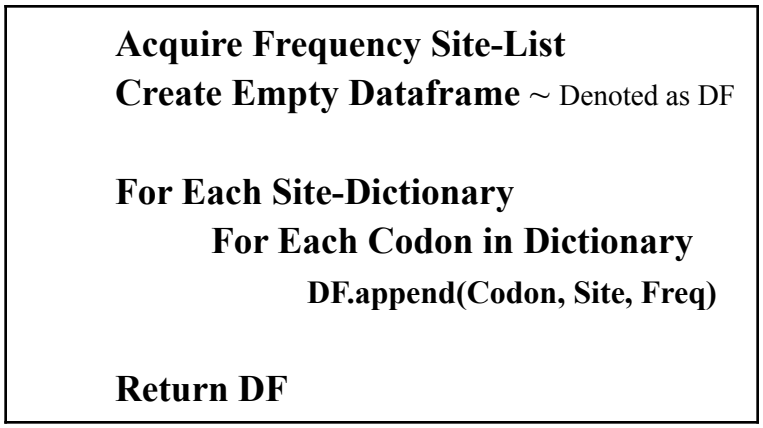

Figure 1.8

Exporting Graphical Data

After the frequency site-list has been converted to a dataframe, graphs are the next items to be exported. This process begins by filtering out ambiguous reads from the total results dataframe (produced from data extraction phase) via pandas. Display_graph takes this dataframe and creates two graphs for **each** target site to showcase the codon occurrence trends between sample types. One graph is a boxplot, which contains side-by-side comparisons of codon occurrences between wild and mutant samples. The other graph is a stripplot, which contains occurrence comparisons as two sets of colored points on the same x-codon-coordinate. Using the *E. coli* research as an example, three target codon-sites (E, P, and A) generated two graphs each, one boxplot and one stripplot. The total number of graphs produced was 12, half being boxplots and the other half stripplots.

Figure 1.9 provides insight on the code logic within display_graph.

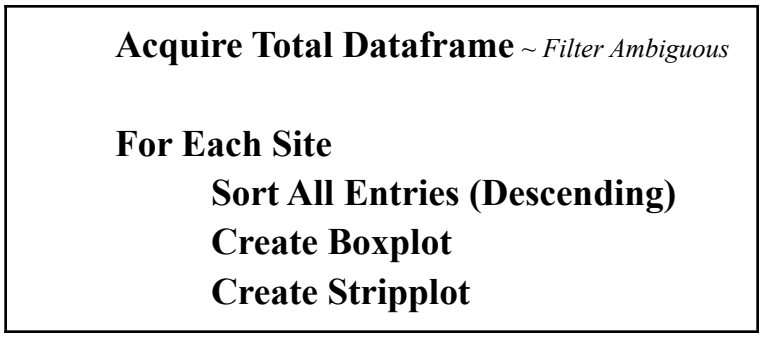

Figure 1.9

Figure 1.10 provides two graph examples produced from the *E. coli* data. These graphs were created using the seaborn and matplotlib packages.

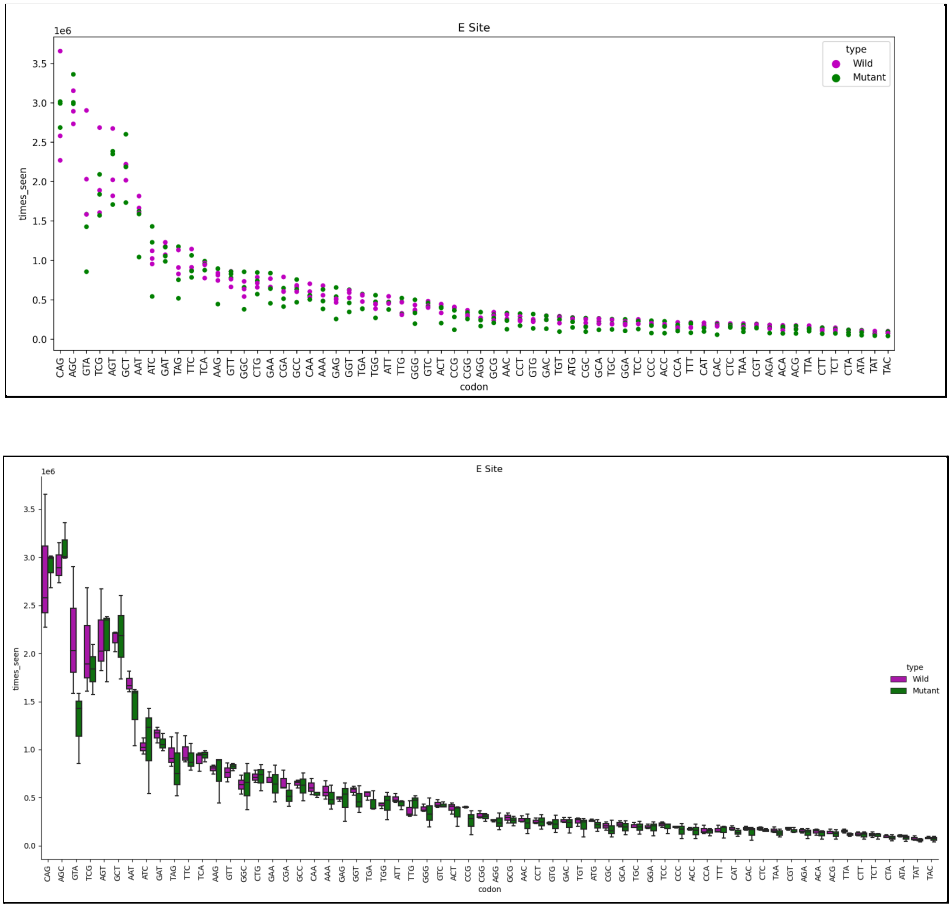

Figure 1.10

Exporting Dataframes

After all graphs have been created, Codano concludes by exporting the frequency dataframe created by frequencies_to_dictionaries. The dataframe is sorted in descending order by frequency value and then exported as a csv file via pandas. The program ends with this process, however, there is still one more output from the data extraction phase that is important to note. The end of the data extraction phase produces a csv file containing raw sequence-read analytics. There are 7 columns in this csv file: one for filename, codon, amino acid, occurrence, site, ambiguous flag, and type flag entries. This file may be useful for scientists who wish to further manipulate the data with external tools such as SQL based applications. ***Figure 1.11*** provides example csvs that were generated from the *E. coli* research.

Table: wild_vs_mutant_codon_site_frequencies

	codon	site	frequency
	Filter	Fil...	Filter
1	CAC	A	1.69124928801677
2	GTA	E	1.68567712118806
3	GAG	P	1.61594959854527
4	CCG	E	1.58280920230583
5	TTT	A	1.41684404434116
6	AAC	A	1.39426972459335
7	TGT	P	1.34610470019216
8	CGG	A	1.3321824304766
9	CAC	E	1.32822524911407
10	CTT	P	1.31614736789579
11	CAT	E	1.31495694132953
12	TAT	E	1.29874672599769
13	AGA	P	1.29822467886537
14	TTA	E	1.29578193835731
15	TCG	A	1.29111679809271
16	GTG	A	1.28626436250501
17	CGA	E	1.27024111545829
18	CCG	A	1.26449352844734
19	GCA	P	1.26278822543965
20	ATA	E	1.25783144475336
21	GCT	A	1.25363683991445
22	TAT	A	1.25205710130506
23	TCT	A	1.25093140255986
24	ATC	P	1.24973237228188
25	ATT	A	1.23909407837979
26	CAG	A	1.23318592891966
27	TAC	P	1.23288628766176
28	TGT	E	1.23040949967194
29	ATG	E	1.22845815144115
30	ATG	P	1.22754778362319

1 - 30 of 192

Table: total_results

	read_name	codon	amino_acid	times_seen	site	contains_ambiguous	type
	Filter	Filter	Filter	Filter	Fil...	Filter	Filter
154	RIBO_WT3_R1_trimmed	TTT	F	232482	P	False	Wild
155	RIBO_WT3_R1_trimmed	TAA	STOP	145695	P	False	Wild
156	RIBO_WT3_R1_trimmed	ACG	T	123755	P	False	Wild
157	RIBO_WT3_R1_trimmed	GTG	V	194931	P	False	Wild
158	RIBO_WT3_R1_trimmed	CGT	R	398855	P	False	Wild
159	RIBO_WT3_R1_trimmed	GGT	G	856365	P	False	Wild
160	RIBO_WT3_R1_trimmed	TTA	L	207581	P	False	Wild
161	RIBO_WT3_R1_trimmed	AAT	N	366746	P	False	Wild
162	RIBO_WT3_R1_trimmed	ATT	I	363725	P	False	Wild
163	RIBO_WT3_R1_trimmed	TAC	Y	114388	P	False	Wild
164	RIBO_WT3_R1_trimmed	GGC	G	248300	P	False	Wild
165	RIBO_WT3_R1_trimmed	TCT	S	389185	P	False	Wild
166	RIBO_WT3_R1_trimmed	CTC	L	197562	P	False	Wild
167	RIBO_WT3_R1_trimmed	CCA	P	179520	P	False	Wild
168	RIBO_WT3_R1_trimmed	ACA	T	106620	P	False	Wild
169	RIBO_WT3_R1_trimmed	CTA	L	101642	P	False	Wild
170	RIBO_WT3_R1_trimmed	CGA	R	400163	P	False	Wild
171	RIBO_WT3_R1_trimmed	CGC	R	179201	P	False	Wild
172	RIBO_WT3_R1_trimmed	TGA	STOP	234915	P	False	Wild
173	RIBO_WT3_R1_trimmed	ACC	T	334572	P	False	Wild
174	RIBO_WT3_R1_trimmed	GAC	D	230936	P	False	Wild
175	RIBO_WT3_R1_trimmed	AAC	N	116553	P	False	Wild
176	RIBO_WT3_R1_trimmed	ATG	M	228946	P	False	Wild
177	RIBO_WT3_R1_trimmed	CAT	H	151499	P	False	Wild
178	RIBO_WT3_R1_trimmed	CAC	H	89833	P	False	Wild
179	RIBO_WT3_R1_trimmed	GNG	NULL	12	P	True	Wild
180	RIBO_WT3_R1_trimmed	NTG	NULL	5	P	True	Wild
181	RIBO_WT3_R1_trimmed	CNT	NULL	2	P	True	Wild
182	RIBO_WT3_R1_trimmed	ANG	NULL	3	P	True	Wild
183	RIBO_WT3_R1_trimmed	GCN	NULL	8	P	True	Wild

154 - 183 of 2005

Figure 1.11

6. Conclusion

Codano provides a free and relatively simple solution to assess the frequency of codons at target sites between two sample type datasets. Codano accomplishes this feat by partitioning its major computations into three stages. The process begins with the data extraction phase, which reads all input files for DNA sequences and reads user-specified codon-sites. Data extraction is followed by data processing, which takes the analytics from the previous stage and computes the frequencies of codon seen at each site. Finally, the script concludes by outputting both graphical and numerical results related to the solution. This white paper covered each of these three phases in great detail to promote its use in other research projects. Armed with this knowledge, researchers and developers alike should be able to repurpose this software for their personal research needs.

References

"About Us." *Anaconda*, https://www.anaconda.com/about-us.

"Biopython." *Biopython · Biopython*, https://biopython.org/.

MARKEL, MIKE. *Practical Strategies for Technical Communication*. BEDFORD BKS ST MARTIN'S, 2021.

Kelly, Michael. "Codon-Analysis-Tool." *GitHub*, https://github.com/MPKellyy/codon-analysis-tool.

"Pandas." *Pandas*, https://pandas.pydata.org/.

"Ribosomal DNA." *Biology Articles, Tutorials & Dictionary Online*, 27 Feb. 2021, https://www.biologyonline.com/dictionary/ribosomal-dna.

"Statistical Data Visualization." *Seaborn*, https://seaborn.pydata.org/.

"The Genetic Code & Codon Table (Article)." *Khan Academy*, Khan Academy, https://www.khanacademy.org/science/ap-biology/gene-expression-and-regulation/translation/a/the-genetic-code-discovery-and-properties.