

The HUMANIZER:

A Much Needed Tool for Genetic Engineering

By

Stacy Vang

An Honors Project Submitted in Partial Fulfillment

of the Requirements for Honors

in

The Department of Mathematics and Computer Science

Rhode Island College

2018

Abstract

The Humanizer is a program developed to solve a problem genetic researchers encounter when humanizing genes. To humanize a gene means to make modifications to a model organisms gene so it may perform in a way that is more like how it would perform in humans. This is done by making changes to a model organisms gene only when a discrepancy exists between the model organisms gene and the human gene. Today, researchers in this field spend long hours humanizing genes manually because there is no other way to do it. In addition, the length of genes can range from hundreds to thousands of nucleotide base pairs which can make the process even more dreadful. If done manually, it is inevitable that researchers have to humanize each nucleotide, one by one. Such a process is time consuming, error-prone, and requires the coordination of many tools. Therefore, the purpose of The Humanizer is to change the manual process into an automatic process, cutting the wait time from several days to several minutes. This application performs as any researcher would: determining genes in test organisms by querying through databases, comparing and retrieving related sequences, and humanizing the gene one base at a time, among other things.

Contents

1	Introduction	1
2	Background and Related Work	6
2.1	Understanding the biology	6
2.2	Learning BioPython	9
2.3	Related bioinformatics tools	9
3	Design	11
3.1	System Architecture	11
3.2	User interface design	15
4	Implementation	15
4.1	A Step-by-Step Walkthrough	16
5	Testing	35
5.1	Functional testing	35
5.2	Usability testing	40
6	Conclusions and Future Work	40
A	Program Code	44

List of Figures

1	Exons and Introns	2
2	Codon Bias: Drosophila Melanogaster (Fruit Fly)	5
3	Codon to Amino Acid Table	7
4	Architecture Diagram	11
5	The Humanizer: Implementation Flow Chart	16
6	The Humanizer: Finding the Executable Application	17
7	Codon Bias Database: Instruction	18
8	Codon Bias Database: Elements in database	18
9	Codon_Bias_2D_List and Codon_Bias_Directory	20
10	Welcome Greeting	21
11	Test Organism Error	21
12	Test Organism and Gene	22
13	Entrez Search	22
14	Entrez Results	23
15	Select Test-Organism DNA Sequence Record	24
16	BLASTing to Retrieve Test-Organism Protein Sequence	25
17	BLASTing to Retrieve Human Protein Sequence	26
18	Select Human Protein Sequence Record	27
19	Aligning Protein Sequences	28
20	3-Frame Translation	29
21	BioPython: Codon Table	30
22	The humanizing process	31
23	Returns Humanized DNA Sequence and Prompts to Save Sequence	33
24	Saving results in a text file on user's desktop	33
25	End of Program	34
26	Text file of saved results	34
27	Automatic Nucleotide Search with the Humanizer: finding the test organism's DNA sequence	35
28	Manual Nucleotide Search: finding the test organism's DNA sequence	36
29	Automatic BLASTX Search: finding the test organism's protein sequence	37
30	Manual BLASTX Search: finding the test organism's protein sequence	37
31	Automatic BLASTX Search with the Humanizer: finding the human protein sequence	38
32	Manual BLASTX Search: finding the human protein sequence	38
33	Automatic humanized form: includes protein sequences alignment, 3-frame translation, and determination of the open reading frame	39
34	Manual humanized form: reverse check with BLASTX against homo sapiens	39
35	Flowchart of next steps	42

1 Introduction

Working with Dr. Geoffrey Stilwell, head of the genetics research team at Rhode Island College, I have designed and developed an automated tool called “The Humanizer”. The purpose of this program is to improve the way genetic researchers perform a task known as “humanizing” genes, by converting the task from a manual process into an automatic process.

The Humanizer automates an inherently tedious and error-prone manual process and, for the first time, makes it possible for researchers to perform this task in minutes rather than days. The Humanizer performs as any researcher would: determining genes in test organisms by querying through databases, comparing and aligning results to retrieve related sequences, and humanizing the gene one base at a time, among other things. Not only that, but one of the strengths of this application is its ability to incorporate all the different modules required in a manual job into just one application.

One of the uses of the Humanizer will be to help with research of human diseases. When we take notice of the years passing by, as if we were watching a time-lapse video, we would notice that the planet we call Earth has quickly transformed into a technology driven environment by the dominant omnivores we call homo sapiens, humans. As we learn more, our curiosity grows generating the beginning of an ever-growing loop of learning.

Among our many curiosities lies the need and urge to figure out more about diseases that hinder the lives and bodily functions of our own kind. Genes are segments of DNA that hold the instructions to code for specific traits, or proteins, of an organism. More than 3,000 human diseases are caused by heritable mutations in genes which produce mutant proteins. Mutant proteins are proteins encoded by a gene with alterations in its DNA sequence. Thus, when constructing a protein out of this DNA sequence, it can lead to missing or malformed proteins, which causes diseases. More explanation on mutations is given in Chapter 2. Two examples of diseases caused by mutations include sickle-cell anemia and cystic fibrosis. Sickle-cell anemia is the cause of a mutation in the gene for hemoglobin, a protein responsible for transporting oxygen in the blood. This causes red blood cells to distort into a sickle shape and clogging capillaries which leads to cut-offs of blood circulation. On the other hand, cystic fibrosis is a disease that causes build-ups of mucus in the lungs, pancreas, and other organs which leads to the clogging of airways, respiratory failure, and prevention of the breakdown of food and absorption of nutrients [The Tech Museum of Innovation, 2018].

Today, we can learn quite a bit about human diseases directly from humans, though how much we are able to learn is limited. For example, researchers can learn more about diseases by giving out surveys to people who have a certain type of disease and figuring out how they feel, if they experience any restrictions with their type of disease, etc., but even with these answers, researchers have to be aware of the possibility that answers are not 100% accurate. Or, an experimentation can

occur where there is a control group and an experimentation group: say a group of people with a certain disease who are not taking medications versus another group of people with that same disease who are taking a type of medicine. This can determine whether or not the medication helps people in any way. Still, these types of responses can vary from person to person. So, there is only so much researchers can learn about diseases in this way without going against any ethical values. Therefore, more concrete experimentation must occur to really study the origin, symptoms, and possible cures of diseases. This leads to experimentation on model organisms.

Because humans are so genetically different and there are so many variables to account for when studying the human population, researchers have begun to rely on model organisms. Model organisms are a group of organisms that aid in the understanding of biology in humans. More specifically, their genetic make-up is well-known to researchers, and they have similar biology compared to humans. Examples of some model organisms used throughout research includes fruit flies, mice, and zebrafish

[National Institute of General Medical Sciences (NIGMS), 2017].

The theory of evolution, made prominent by Charles Darwin, states that organisms change over time because of changes made in heritable physical or behavioral traits. Therefore, genes in animals are evolutionarily conserved. This means that although related, they are not identical in sequence across species. As a result, proteins encoded by related genes are similar and perform similar functions generally; however, subtle differences in the structure and function of the proteins may exist [Than, 2018].

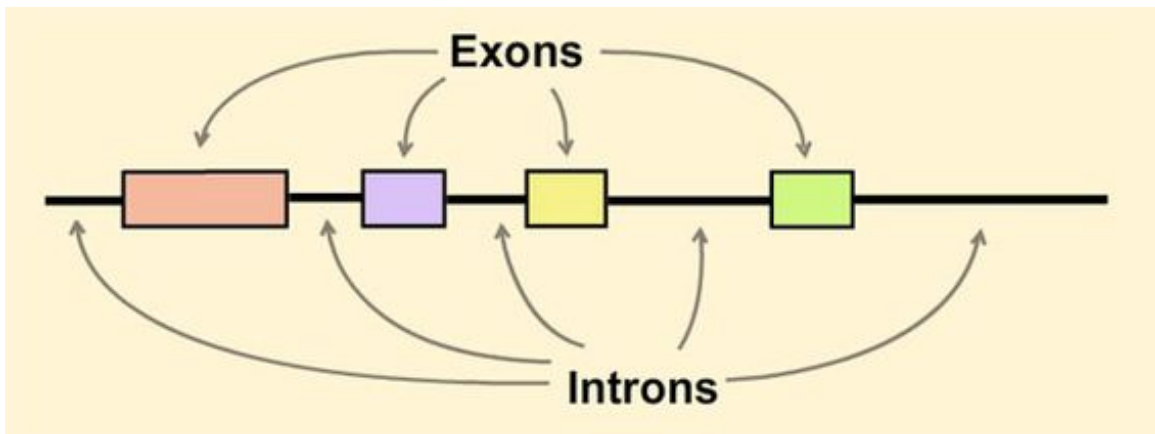


Figure 1: Exons and Introns

source: <https://www.pinterest.com/pin/235031674278137353/?autologin=true>

One type of experimentation that has begun to pick up in recent years are experiments run by directly injecting a gene's coding sequence (CDS) with a mutation that causes a disease into a test organism, or model organism. A gene is a portion of

DNA sequence which includes exons and introns as shown in Figure 1.

A CDS represents the exons of a gene which are the DNA sequences that code for a particular protein. Thus, exons code for a protein whereas introns are intervening non-coding regions which play no important functional role at this time. Mutations within exons (the CDS) are often the basis of human disease. Researchers can inject a mutated CDS into a test organism as a way to study the disease.

Human disease research relies on being able to manipulate genes in non-human organisms and recent technological advances has made this process less cumbersome and time-intensive. It is now possible to insert human genes into non-human organisms and study their effects. Ideally, the best approach to 'humanizing' a gene is to make the fewest changes possible and only within exon regions of a gene.

Because the technology to conduct these biological studies is so new, Biologists humanize genes manually because no automated tool exists. The length of genes can range from hundreds to thousands of nucleotide base pairs. If done manually, it is inevitable that researchers have to humanize each nucleotide in a DNA sequence, one by one. Such a process is, both, time consuming and prone to errors. At RIC, Dr. Stilwell has adopted the manual technique of humanizing genes used in the industry.

The National Center for Biotechnology Information (NCBI) contains a large DNA database and bioinformatic tools to access and analyze genomic information. Biologists access and analyze DNA through a GUI interface which limits the available tools. In part, the 'humanizer' utilizes some of the existing algorithms including the Basic Local Alignment Search Tool (BLAST). Within BLAST exists sub-tools: BLASTP, BLASTN, BLASTX, TBLASTN, and TBLASTX. Each sub-tool focuses on a different purpose, but the one used in the Humanizer is BLASTX.

The first step in humanizing a gene by hand is to first search for the protein sequence of the model organism and gene the user wishes to humanize. This is done by entering in the organism's name and gene name into the protein database search provided within the NCBI website. A number of results may appear and it is up to the user to decide which record they wish to use. Then, the user retrieves a protein sequence for the gene within humans by using the BLASTP tool, also provided on the NCBI website. This requires a query sequence, which is the sequence retrieved earlier from the protein database. This sequence gets put into the BLASTP search along with an optional parameter to limit the search within humans only. In this case, the user would enter in 'homo sapiens' into the optional 'organism' text box in the GUI interface.

After the researcher retrieves the two protein sequences the researcher can obtain the DNA sequence of the gene within the model organism. This is done by entering in the name of the organism and the name of the gene directly into the nucleotide database within the NCBI website. A number of results may appear and it is up to the user to select the record they want.

Now, the user can perform an alignment on the two protein sequences, meaning compare and modify them to make them more comparable. This requires the use of

a pairwise alignment tool. Dr. Stilwell uses a pairwise alignment tool called Emboss Water, provided in a separate website. The user pastes in the protein sequence of each organism into the GUI interface for alignment. Once the protein sequences are aligned, differences between the model organism and human sequences are identified and displayed to the user where changes have been made and where there are discrepancies between the amino acids of both sequences.

Going back to the NCBI website, the user obtains a DNA sequence of the model organism and gene from the nucleotide database. This step is similar to how the user obtains a protein sequence: by entering in the organism's name and gene name into the nucleotide database. A number of results may appear and it is up to the user to decide which record they wish to use.

Then, using the nucleotide sequence retrieved from the nucleotide database, the user will run it in another tool to obtain the correct reading frame of the DNA sequence. Dr. Stilwell uses a tool called Emboss Transeq, also provided in the same website as Emboss Water. This tool takes a DNA sequence, inputted directly by the user, and translates it into six possible reading frames. Three (3) reading frames from reading the DNA sequence from the beginning to the end of the sequence, and then again in reverse, from the end of the sequence to the beginning. More information about what reading frames are is provided in Section 3.1. Emboss Transeq returns the six possible reading frames to the user and the result that matches the protein sequence retrieved from the protein database from NCBI in step one determines the correct reading frame.

Finally, now that the user has an alignment of the two protein sequences for comparison and the DNA sequence that codes for the model organism's protein sequence, the user can now manually make changes to the DNA sequence nucleotide by nucleotide.

***Drosophila melanogaster* [gbinv]: 42417 CDS's (21945319 codons)**

fields: [triplet] [frequency: per thousand] ([number])

UUU 13.2(289916)	UCU 7.0(154186)	UAU 10.8(236811)	UGU 5.4(118088)
UUC 21.8(479372)	UCC 19.6(429341)	UAC 18.4(403675)	UGC 13.2(288853)
UUA 4.5(97715)	UCA 7.8(171695)	UAA 0.8(17807)	UGA 0.5(10767)
UUG 16.1(353621)	UCG 16.6(365159)	UAG 0.7(14362)	UGG 9.9(217518)
CUU 9.0(196787)	CCU 6.9(151856)	CAU 10.8(236061)	CGU 8.8(192276)
CUC 13.8(303153)	CCC 18.1(396168)	CAC 16.2(354699)	CGC 18.0(395106)
CUA 8.2(180360)	CCA 13.5(297071)	CAA 15.6(342415)	CGA 8.4(185119)
CUG 38.2(839127)	CCG 15.8(347206)	CAG 36.1(792657)	CGG 8.2(180473)
AUU 16.6(363497)	ACU 9.5(208889)	AAU 21.0(460669)	AGU 11.5(252555)
AUC 22.9(502821)	ACC 21.3(467509)	AAC 26.2(575297)	AGC 20.4(447808)
AUA 9.5(208315)	ACA 11.0(241893)	AAA 17.0(372524)	AGA 5.1(112784)
AUG 23.6(518200)	ACG 14.4(315479)	AAG 39.5(866960)	AGG 6.3(137902)
GUU 11.0(240735)	GCU 14.4(315879)	GAU 27.6(604730)	GGU 13.3(291161)
GUC 13.9(304893)	GCC 33.6(736394)	GAC 24.6(540386)	GGC 26.7(587016)
GUA 6.4(139476)	GCA 12.8(280181)	GAA 21.1(462468)	GGA 18.0(395377)
GUG 27.8(609794)	GCG 14.0(307977)	GAG 42.5(933622)	GGG 4.7(102708)

Figure 2: Codon Bias: *Drosophila Melanogaster* (Fruit Fly)

source: <http://www.kazusa.or.jp/codon/cgi-bin/showcodon.cgi?species=7227>

To humanize a gene by hand requires the usage of a codon bias table. An example of this is shown in Figure 2. More information about this is provided in Section 3.1. To get this though, users would have to look up the model organism's codon bias online. Using this information, amino acid by amino acid, the user will see where there is a discrepancy between the two protein sequences. If the amino acids in the same position of both sequences do not match, a change must occur. For example, let's say the amino acid in position five (5) of the model organism's protein sequence is 'L', and the amino acid in position (5) for the human protein sequence is 'A'. The user will have to look at the codon bias of the model organism and find out what three (3) nucleotides code for the amino acid 'A'. Then, the user changes the nucleotide bases in the DNA sequence, of the model organism, that coded for 'L' and changes that to the nucleotide bases that code for 'A'. This occurs until the user reaches the end of the two protein sequences.

The humanized gene is crucial, but such a technique is both tedious and prone to mistakes, as you can probably tell, which can lead to inaccurate results. Researchers

also spend a great deal of time humanizing genes when they can be experimenting and researching. In addition, for different test organisms, a different humanized form is required regardless if it is intended for the same gene or not. This also results in repeated work being done.

In Dr. Stilwell's lab, the genetics research team has been studying a disease called Amyotrophic Lateral Sclerosis (ALS). ALS is a disease that causes nerve cells to break down which weakens the muscles and reduces muscle functionality. This causes symptoms such as muscle twitching, also known as fasciculations, muscle cramps and weakness, and difficulty with chewing or swallowing. As time passes, the symptoms only get worse. It is estimated that between 14,000 - 15,000 Americans have ALS, but there is no cure for it yet, only treatments to control its symptoms. There are two types of ALS: Sporadic ALS and Familial ALS. The majority of people with this disease have sporadic ALS, meaning the disease may have occurred randomly with no family history of the disease. Only 5%-10% of people with ALS are diagnosed with familial ALS, which means the individual inherited the disease from their parents. In the case of familial ALS, it has been identified by scientists from the National Institute of Neurological Disorders and Stroke (NINDS) that some familial ALS cases were associated with a mutation in the SOD1 gene. It is still unclear how this mutation led to the degeneration of motor neurons [National Institute of Neurological Disorders and Stroke (NINDS), 2013]. ALS is just one of many diseases that we still know very little about. Therefore, the need to find out more about diseases and how to cure them is a top priority for many researchers in this field.

In the remainder of this thesis, I will describe some of the biological terms, concepts and tools needed to understand the purpose and goals of the program. Secondly, I will go over the modules that were integrated to complete the application. Then, discussions of implementation will proceed. After that, testing will follow suit. And lastly, the conclusion and future work section will end the thesis.

2 Background and Related Work

2.1 Understanding the biology

Before developing this tool, there are some biology terms and concepts that I needed to learn and understand to develop this application.

Deoxyribonucleic acid (DNA) is a chain of nucleotides carrying genetic information. It is made up of two strands that are twisted together to form a helix. Each strand is a sequence of genes, and each gene is a sequence of nucleotides. One of the most important construction pieces of nucleotides are their nitrogenous bases.

There are four (4) types of nitrogenous bases found in DNA strands: adenine (A), cytosine (C), guanine (G), and thymine (T). Each strand contains the complementary genetic information of the other strand. In DNA, adenine pairs with thymine, and

cytosine pairs with guanine. Therefore, if one of the DNA strands contain bases 'CAGGTA', then the other strand should carry its complementary bases 'GTCCAT' [Genetics, Education, Discovery (GeneEd), 2018].

Genes are passed down by parents to children, and as stated earlier, they carry the instructions needed to assemble proteins. Proteins are large macromolecules that are major structural and functional components of cells. Each protein is constructed by a chain of amino acids, and an amino acid is the construction of three (3) nitrogenous bases. These three (3) nitrogenous bases are also called a codon.

		second base in codon				
		T	C	A	G	
T	first base in codon	TTT Phe	TCT Ser	TAT Tyr	TGT Cys	T C A G
		TTC Phe	TCC Ser	TAC Tyr	TGC Cys	
		TTA Leu	TCA Ser	TAA stop	TGA stop	
		TTG Leu	TCG Ser	TAG stop	TGG Trp	
C	first base in codon	CTT Leu	CCT Pro	CAT His	CGT Arg	T C A G
		CTC Leu	CCC Pro	CAC His	CGC Arg	
		CTA Leu	CCA Pro	CAA Gln	CGA Arg	
		CTG Leu	CCG Pro	CAG Gln	CGG Arg	
A	first base in codon	ATT Ile	ACT Thr	AAT Asn	AGT Ser	T C A G
		ATC Ile	ACC Thr	AAC Asn	AGC Ser	
		ATA Ile	ACA Thr	AAA Lys	AGA Arg	
		ATG Met	ACG Thr	AAG Lys	AGG Arg	
G	first base in codon	GTT Val	GCT Ala	GAT Asp	GGT Gly	T C A G
		GTC Val	GCC Ala	GAC Asp	GGC Gly	
		GTA Val	GCA Ala	GAA Glu	GGA Gly	
		GTG Val	GCG Ala	GAG Glu	GGG Gly	

Figure 3: Codon to Amino Acid Table

source: <https://www.chemguide.co.uk/organicprops/aminoacids/dna4.html>

There are twenty-two (22) different types of amino acids that can be constructed but only twenty (20) main ones are in use. Although it may seem that there should be

sixty-four (64) different amino acids, it is important to know that there exists more than one way to construct a single amino acid. This is seen in Figure 3. In Figure 3, Valine (Val) can be assembled from the nitrogenous bases 'GTT', 'GTC', 'GTA', or 'GTG'. Hence, there also exists a concept known as codon bias¹ where every type of organism has a preferred construction of each amino acid in their DNA.² This concept will be very important to the completion of the Humanizer. More information about this topic will be given in Chapter 3 and Chapter 4.

In Figure 3, to figure out which amino acid is created by each codon, find its first nitrogenous base on the left hand side. Then follow that row across until you find its second nitrogenous base, which is sorted by columns. Finally, you are left with four options, each with a different third nitrogenous base. The amino acid that is created by that codon will be listed in its abbreviated form to the codon's right. Let's say we are looking for the amino acid made from the codon 'CCT'. Its first nitrogenous base is 'C', so we will look at the second row. The next nitrogenous base is also 'C'. Therefore, we will stop at the second column of that row. The last nitrogenous base is a 'T', which is listed as the first codon within that section. Thus, the codon 'CCT' codes for the amino acid Proline (Pro). This figure also contains codons that code for 'start' and 'stop' codons. This is available because within genes exist an open reading frame. This open reading frame is actually where the instruction that code for a protein begins and ends. A gene does not typically contain only instructions that code for the protein. For a protein to be built, another protein called a promoter will read through the gene until it finds this 'start' codon to begin constructing the protein. It will know when to stop coding for the protein when it reads a 'stop' codon. In Figure 3, the codon 'ATG' which codes for Methionine (Met) is the start codon, and the codons 'TAA', 'TAG', and 'TGA' codes for a stop codon [Pevsner, 2015].

As a result of three (3) nitrogenous bases, a codon, coding for one (1) amino acid, and a chain of amino acids constructing a protein, if a nitrogenous base is switched out with another nitrogenous base, there is a huge chance that different amino acid will be constructed. In addition, imagine if there is an insertion of an extra nitrogenous base, a deletion of a nitrogenous base. This means the amino acid construction from that instance and ongoing will all be disrupted. This alteration of the nucleotide sequence will end up altering the composition of the protein. If more than one amino acid is changed, the construction of the whole protein will be flawed. This is what scientists call a mutation. Thus, the constructed protein may not perform as it should which may also lead to diseases [The Tech Museum of Innovation, 2018].

¹Codon bias is based on research done among a mass of DNA sequences from organisms to determine the most frequently used construction of each amino acid.

²This does not go to say that there does not exist a less frequently used codon within an organism's DNA.

2.2 Learning BioPython

To learn BioPython, the most useful tool was the online documentation found on BioPython's website [Chang et al., 2017]. This documentation provided information on some of the main modules supported, and examples of how to use them. I thought this documentation was best at providing a good foundation of BioPython. It was also fairly easy to follow for beginners.

When the information provided in the BioPython website was not enough, I found the Application Programming Interface (API) documentation helpful [International Association of Developers, 2017]. This document defines each of the Python objects within BioPython more specifically. A separate example for each explanation is provided and further links are provided as well to learn even more about each module. Because this API documentation includes so many other links, I think this was most helpful in understanding how to use the required modules. Using this documentation in combination with the source code for BioPython, I was able to understand what parameters were required, why they were required, optional parameters, return values, among other things.

2.3 Related bioinformatics tools

BLAST is a program that uses rigorous statistics to score sequences. The scores reveals related sequences present in the same organism or different organisms to show how closely related each result is to the query sequence. BLAST takes an input query sequence, and performs a pairwise alignment between the given sequence and a database. A pairwise alignment is an alignment of two sequences which determines their relatedness at a sequence level. So, all search results from a BLAST search are either highly related to the query sequence or marginally related. Related sequences may be homologous and have common functions.

The BLAST algorithm consists of three (3) phases: list, scan, and extend. The first phase of the algorithm compiles a list of words of a specific size. In protein searches, the default size of word pairs is three (3). These words are generated directly from the query sequence. For example, let's say the query sequence includes the following amino acid sequence "KVNALTVWG". Thus, the word pairs of size three (3) would be 'KVN', 'VNA', 'NAL', 'ALT', 'LTV', 'TVW', and 'VWG'. Then, with each word generated from the sequence, a list of similar words are produced. A couple of similar words that would be produced for the word 'KVN' would be 'KVL', 'KVA', 'KTN', and 'WVN'. Because there are twenty (20) different amino acids and each word size is three (3), then there are 8000 possible words. A threshold value, T, is established for the score of aligned words. If the threshold value is raised, the BLAST search takes less time, but the user will receive less results. These results will not include distantly related database matches. Thus, the opposite would occur if the threshold value is lowered. BLOSUM62, a common scoring matrices for amino acids, is used to score each pair of similar words. So, if any words from the list of

similar words are equal to or greater than the threshold value, then that word moves on to the next stage in the BLAST algorithm. If any words from the list of similar words are less than the threshold value, they are not pursued any longer.

The next phase in the blast algorithm is the scan stage. A scan of the database for word pairs that matches the pairs of similar words that had passed the threshold value from the previous step is executed. In the example above, let's assume that from the list of similar word pairs the words 'KVA' and 'KTN' passed the threshold value T . Then, in this step, a scan of the protein database for sequences containing the words 'KVA' or 'KTN' is performed. Notice that these word pairs are not exact: compare 'KVA' and 'KTN' to the word from the query sequence 'KVN'. This idea of incorporating a threshold allows the BLAST search to return exact sequences and non-exact but similar sequences. So, when a 'hit' is found, this is called a 'hit'. Each 'hit' is then extended for the rest of the sequence before and after the word match using gaps to create an alignment. During extension, a score is calculated using some sort of scoring matrices like BLOSUM62. The extension continues as long as the score continues to increase. Once it drops to a critical amount, this is called a "dropoff", and the 'hit' is no longer pursued. Meanwhile, any 'hit' whose score exceeds a particular cutoff score, S , is known as a high-scoring segment pair (HSP) and it returned as a BLAST result. During extension, to increase efficiency, insertions, deletions, and mismatches are not accounted for. This leads up to the third phase.

The last phase is a trace-back of the 'hit' sequence to locate insertions, deletions, and mismatches between the 'hit' and the query sequence that were not saved earlier.

There are five different BLAST algorithms, but the one used in this application is called BLASTX. BLASTX takes a DNA sequence and translates it into protein sequences using all the different reading frames. More information about reading frames will be provided in Section 3.1. After translating the DNA sequence, it then takes the protein sequences and performs a pairwise alignment against the protein database. This alignment determines the protein sequence record that is encoded by the DNA sequence.

The sequence inputted into the BLAST search in Figure 16 is the non-human DNA sequence record. It is translated into three (3) different protein sequences, and then taking those sequences a pairwise alignment is done against the protein database for the record that represents each sequence. The default database in BLASTX is the non-redundant (nr) database which contains sequences from Genbank, Protein Data Bank (PDB), SwissProt, PIR, and other data banks supported by NCBI. In total, there is approximately 65 million protein sequences in the non-redundant database. Therefore, it is important to keep in mind that when using the BLAST tool, it can take as long as a few seconds to a couple of hours.

Each record that has any sort of relationship with the DNA sequence will be returned in the results, and each record will have a score and an expect threshold value (E-value). Scores are attained by a scoring scheme which describes the relatedness between the query and each database hit. Scores are calculated from scoring matrices.

In this application BLOSUM62 is used. The greater the score, the more aligned it is with the DNA sequence. The E-value represents the number of alignments, within the search, whose scores are equal to or greater than its score that are expected to occur in a database search only by chance. This gives an estimate of the number of false positive results received from the search; the lower the E-value, the lower the probability that the sequence had occurred by chance. Because the score and E-value are inversely related, the higher the score, the lower the E-value [Pevsner, 2015].

3 Design

3.1 System Architecture

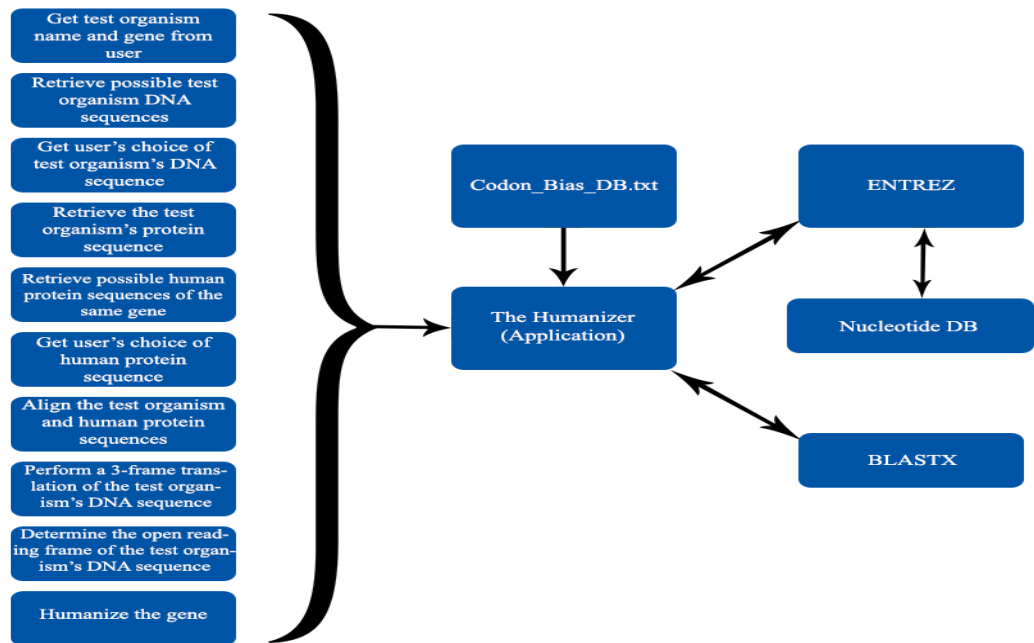


Figure 4: Architecture Diagram

As seen in Figure 4, there are fifteen (15) modules that make up the Humanizer, some that were written for this project and some external modules that needed to be integrated with it.

The Humanizer, itself, has ten (10) modules that perform the following tasks, in sequence:

- Get desired test organism name and gene from the user
- Retrieve possible test organism DNA sequences
- Get user's choice of test organism's DNA sequence
- Retrieve the test organism's protein sequence
- Retrieve possible human protein sequences of the same gene
- Get user's choice of human protein sequence
- Align the test organism and human protein sequences
- Perform a three-frame translation of the DNA sequence
- Determine the open reading frame of the DNA sequence
- Humanize the gene

The Humanizer (Application) This module contains the code for the whole application, and integrates all the following modules together.

- The Humanizer first prompts the user with the name of the model organism, or test organism, and the name of the gene which the user wishes to humanize. This information gets stored into the program for later usage throughout the application.
- Using the name of the test organism and gene, given by the user in the previous module, as parameters, the Humanizer connects to ENTREZ (explained in further detail below) and searches for DNA sequences that meet the parameters.
- A maximum of twenty (20) entries will be shown to the user. The entry with the highest scores will show first. At this point, it is up to the user to determine which record to use as the DNA sequence. It is important that the user makes this decision rather than the application because it would be near impossible to physically identify exactly what the user needs.
- Once the DNA sequence is selected, this selected record is used to find the protein sequence that is most related to it. In other words, the protein sequence that is encoded by the DNA sequence. This is performed in the BLAST tool.
- The DNA sequence is put into a BLAST search again to find records of protein sequences within humans, for the same gene. There may be more than one result because any record that is even slightly related to the DNA sequence will be returned. A maximum of twenty (20) entries will be shown to the user. Records with the highest scores populating first.

- It is now up to the user, again, to determine which record to use as the protein sequence for humans of the same gene. It is important that the user makes this decision rather than the application because it would be near impossible to physically identify exactly what the user needs.
- This step requires the alignment of the two protein sequences, the test organism's protein sequence and the human's protein sequence. The sequences are compared, one amino acid at a time, and aligned to ensure they are of equal length. Gaps are added into the sequences at this time for any sequences lacking an amino acid compared to the other sequence.
- A three-frame translation is then performed on the DNA sequence. The segment of DNA sequence acquired does not always begin with a complete codon. This step is required for the program to define where to start reading the DNA sequence. More information on this step is explained in Chapter 4.
- The open reading frame of a DNA sequence is the segment of instructions that actually code for the protein. As mentioned in Chapter 2 about the start and stop codons, the start and stop codon will determine where the instructions begin and end. Thus, this module reads each codon in the DNA sequence with the help of the reading frame, acquired in the previous module, to determine the coding region of the protein. The rest of the DNA sequence is ignored by the program, and this open reading frame becomes the new DNA sequence.
- Finally, using the two protein sequences, the application compares each amino acid, one by one. When it finds any discrepancies between the amino acids, a change in codon, in the newly updated dna sequence, occurs. More details on this step is provided in Chapter 4

CodonBiasDB.txt The Codon Bias Database is a file that contains the codon bias of all twenty (20) amino acids for numerous test organisms. During humanization of a gene, this module keeps the test organism's DNA sequence as close to its original as possible by allowing the system to take into account the most popular codon that codes for each amino acid. This is determined by looking at every gene in the genome, the complete set of genes in an organism, and determining which codon was used the most for every amino acid.

Within the application, when the protein sequences of both the test organism and humans are determined, each amino acid making up that protein is compared. Any differences between the amino acids, of the two sequences, that require a change in DNA with the test organism's protein sequence will look into this codon file. The application will search for the discrepant amino acid of the human protein sequence, find the codon bias for that amino acid for the

test organism, and then make the necessary changes to the test organism's DNA sequence.

End-users must maintain this file for the most accurate results. Only the individual users will know for certain what types of test organisms are used in their labs. Thus, if a test organism is being used and is not already included in the text file, it is of the responsibility of the user to ensure its information is entered in. Instructions of how to include and format the data is provided in the text file itself. Any test organism whose codon bias information is not documented in this file will result in the termination of the application, as this file is an integral part in humanizing genes.

ENTREZ To build the Humanizer, I had to make it work with several external modules provided by the National Center for Biotechnology Information (NCBI). NCBI is a huge resource that allows access to, not only genomic information, but also to tools that can turn the information retrieved from its databases into information that can be used to humanize genes. Some of the databases supported by NCBI includes the Gene database, Protein database, Nucleotide database, and PubMed. NCBI also supports a tool known as the Basic Local Alignment Search Tool (BLAST). Within BLAST exists sub-tools: BLASTP, BLASTN, BLASTX, TBLASTN, and TBLASTX. Each sub-tool has a different purpose.

Entrez is an integrated search engine used by NCBI to retrieve data from its many supported databases. Specifically, with connection to ENTREZ, the Humanizer can access databases such as the nucleotide database and modules such as ENTREZ's search, summary, and fetch modules. These modules allow the Humanizer access to do numerous things with the results generated from the search.

Nucleotide DB Entrez allows the application to connect to the Nucleotide Database which can retrieve nucleotide sequences, the complete strand of individual nitrogenous bases constructing a protein, or DNA sequence. Entrez inputs the name of an organism and the name of a gene into the database. In our case, the name of the test organism and the name of the gene we are researching would be entered.

Entrez then returns a list of records that match the input, back to the application. There may be more than one response, because all results that are only partially similar will also be returned. These results are then formatted and shown to the user, who will then select the record they wish to use.

BLASTX This module translates DNA sequences into protein sequences, and then compares them to other sequences in the protein database. This reveals related sequences present in the same organism or different organisms.

3.2 User interface design

For now, the Humanizer has a simple text-based command-line interface. Because of time constraints, my main priority was to ensure the usability and accuracy of the program before focusing on the cosmetics of the program. As a result, development of a more visual-friendly user interface will have to be delayed to future work.

4 Implementation

The Humanizer is completely written in Python, a high-level scripting language, along with the addition of the BioPython library. Although I have experience with Python from a previous course, there were still some things that needed to be learned to develop this program. In addition, BioPython was new to me so I experienced a learning curve with this library also. BioPython provides various tools which allow an application to connect to the National Center for Biological Information's ENTREZ search engine (described below) and gather biological information. Currently, the Humanizer is being executed through the python shell.

The Humanizer takes the DNA sequence, retrieved from the Nucleotide DB search, and runs it through BLASTX two times. First, with the Nucleotide DB result, the DNA sequence, and the test organism's name. This will translate the DNA sequence into multiple protein sequences, and save the sequence with the best match as the protein sequence for the test organism. The second BLAST will include the same Nucleotide DB result as a parameter, but will be run against *Homo Sapiens*, humans, instead of the test organism as in the previous BLAST. This will result in a protein sequence match of the same gene within humans. BLASTX will return a list of records with the matches, it will be formatted and shown to the user, who will then choose the match they wish to use. The selected record will now be known as the protein sequence of the test gene in humans.

These two protein sequences are required for comparison for the gene undergoing testing. Now, with the two protein sequences, an alignment is done to ensure the every amino acid from both strands are aligned. This is done by adding gaps into the sequence where there is a lacking amino acid.

Next, a 3-frame translation is done to the DNA sequence to find the correct reading frame of the strand. This step leads to the determination of the open reading frame within the DNA strand. Once this is completed, humanization can finally occur.

Further details and an explanation of each step is provided in the next section.

4.1 A Step-by-Step Walkthrough

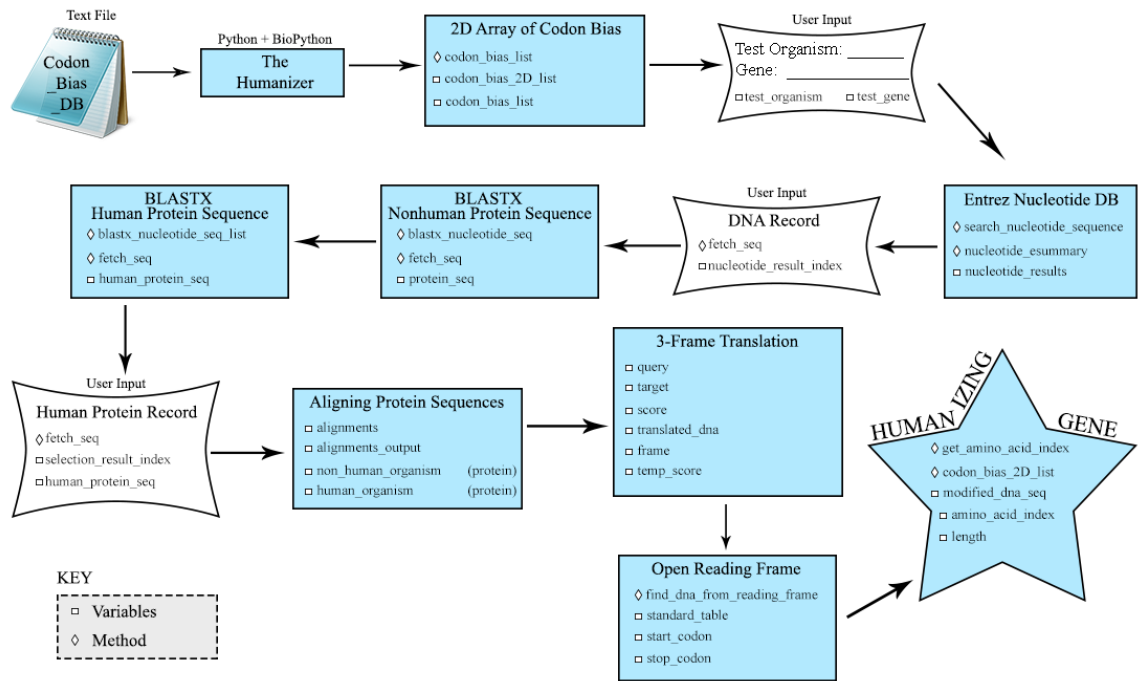


Figure 5: The Humanizer: Implementation Flow Chart

Figure 5 shows a complete flow chart of the implementation of the Humanizer. All modules from the design Section 3.1 are also included, as well as all the methods that are called within each module and the lower-level variables that appear in each.

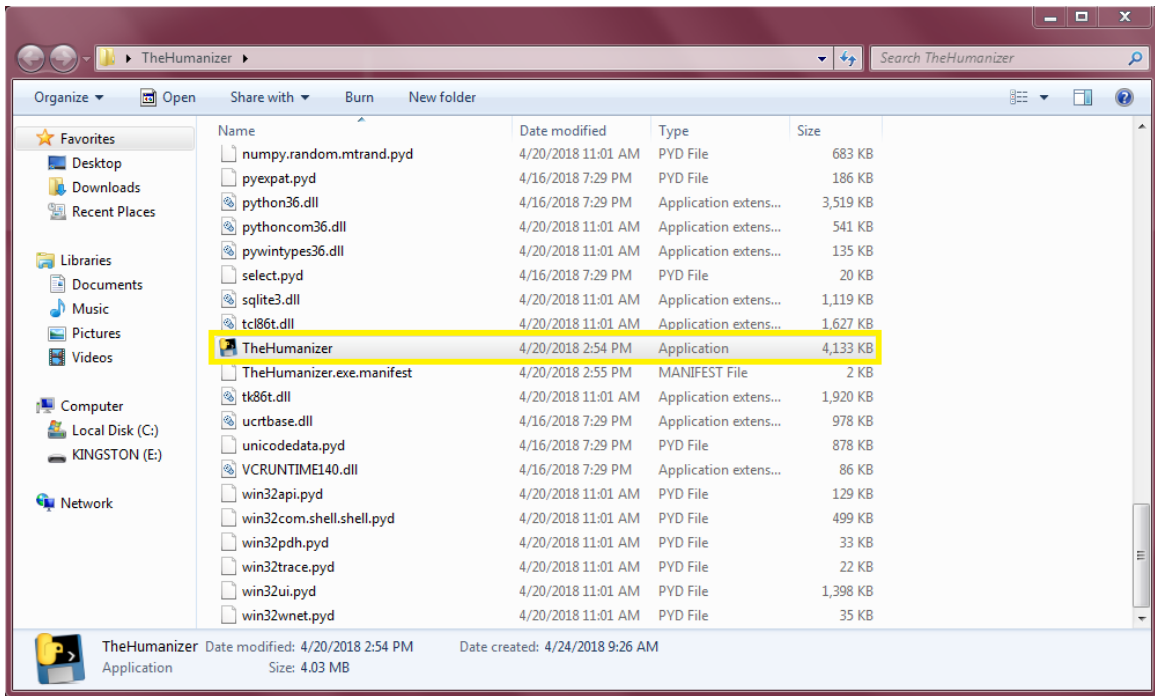


Figure 6: The Humanizer: Finding the Executable Application

To run the Humanizer, after successful installation, first the user locates and opens the 'TheHumanizer' folder. This folder contains all the files necessary to run the humanizer. Next, the user runs the 'TheHumanizer' executable application found within the opened folder. This is shown in Figure 6.

```

Codon_Bias_DB.txt - Notepad
File Edit Format View Help
#####
## Title: Codon_Bias_DB
## Description: This text file will contain the codon bias of all organisms whose gene(s) will be humanized. This data is required to humanize genes by referring to the codon bias of each
## particular protein of a specified organism. Codon bias for organisms must be manually entered in to this file!
##
## Codon Bias Database: http://www.kazusa.or.jp/codon/
##
## Instructions:
## To add a new organism and its codon bias, please follow the steps below.
## There are 20 main proteins:
##
##      1. Alanine      Ala      A
##      2. Arginine     Arg      R
##      3. Asparagine   Asn      N
##      4. Aspartic Acid Asp      D
##      5. Cysteine     Cys      C
##      6. Glutamic Acid Glu      E
##      7. Glutamine    Gln      Q
##      8. Glycine      Gly      G
##      9. Histidine    His      H
##     10. Isoleucine   Ile      I
##     11. Leucine      Leu      L
##     12. Lysine       Lys      K
##     13. Methionine   Met      M
##     14. Phenylalanine Phe      F
##     15. Proline      Pro      P
##     16. Serine       Ser      S
##     17. Threonine    Thr      T
##     18. Tryptophan  Trp      W
##     19. Tyrosine     Tyr      Y
##     20. Valine       Val      V
##
## Proteins that are not included due to them only being referred to in extremely rare cases:
##
##      1. Hydroxyproline Hyp      O
##      2. Pyroglutamic   Glp      U
##
## 1. Each organism requires three (3) lines
##    1. Number of the organism (starting from one (1))
##    ii. Latin name of the organism (use lower-case letters)
##    iii. Codon bias of each protein in the order specified above. Separate each codon with a comma (,) and encase the whole line in brackets ([]).
##
##    e.g. 1
##          Drosophila Melanogaster
##          [GCC, CGC, AAC, GAC, TGC, GAG, CAG, GGC, CAC, ATC, CTG, AAG, ATG, TTC, CCC, TCC, ACC, TGG, TAC, GTG]
##

```

Figure 7: Codon Bias Database: Instruction

```

Codon_Bias_DB.txt - Notepad
File Edit Format View Help
##
##    e.g. 1
##          Drosophila Melanogaster
##          [GCC, CGC, AAC, GAC, TGC, GAG, CAG, GGC, CAC, ATC, CTG, AAG, ATG, TTC, CCC, TCC, ACC, TGG, TAC, GTG]
##
## *****
## WARNINGS:
## 1. This file is required for the program to work. This file is referred to when humanizing the test organism's gene.
## 2. If an organism and its complete codon bias is not included in this file, the program will NOT WORK.
## 3. Please make sure the organism whose gene you are humanizing is included in this file.
## 4. Each organism and its codon bias MUST be written exactly as explained in the instructions. Otherwise, the program will NOT WORK.
## 5. If a codon is written incorrectly, results may be inaccurate.
## 6. Please ensure the data is entered in precisely.
## 7. Any modifications to this file will hinder the program from working correctly, unless it is also edited in the program source code.
## *****
1
drosophila melanogaster
[GCC, CGC, AAC, GAC, TGC, GAG, CAG, GGC, CAC, ATC, CTG, AAG, ATG, TTC, CCC, TCC, ACC, TGG, TAC, GTG]
2
mus musculus
[GCC, CGC, AAC, GAC, TGC, GAG, CAG, GGC, CAC, ATC, CTG, AAG, ATG, TTC, CCT, AGC, ACC, TGG, TAC, GTG]
3
danio rerio
[GCT, AGA, AAC, GAC, TGT, GAG, CAG, GGA, CAC, ATC, CTG, AAG, ATG, TTC, CCT, AGC, ACA, TGG, TAC, GTG]
4
xenopus laevis
[GCT, AGA, AAU, GAT, TGT, GAA, CAG, GGA, CAT, ATT, CTG, AAA, ATG, TTT, CCA, AGC, ACA, TGG, TAT, GTG]
5
caenorhabditis elegans
[GCT, AGA, AAT, GAT, TGT, GAA, CAA, GGA, CAT, ATT, CTT, AAA, ATG, TTC, CCA, AGT, ACA, TGG, TAT, GTT]
6
rattus rattus
[GCC, CGC, AAC, GAC, TGC, GAG, CAG, GGC, CAC, ATC, CTG, AAG, ATG, TTC, CCC, AGC, ACC, TGG, TAC, GTG]
7
saccharomyces cerevisiae
[GCT, CGT, AAT, GAT, TGT, GAA, CAA, GGT, CAT, ATA, TTG, AAA, ATG, TTT, CCA, AGT, ACT, TGG, TAT, GTT]

```

Figure 8: Codon Bias Database: Elements in database

As soon as the user runs the program, the application loads the codon bias database by opening up the “Codon_Bias_DB.txt” file and storing its data into

the program. This allows the application to use the information stored in the database. As seen in Figure 7, the `Codon_Bias_DB.txt` file first contains some meta-data about the file itself: its title, a description of what sort of data it holds, a reference link, and instructions on how to add to the database. Because this database requires maintenance from its users, instructions are necessary to keep the database formatted in the way it is read into the application. The text file's main purpose is to contain the name of a variety of test organisms, along with their codon bias of each of the main twenty (20) amino acids for each organism. Thus, after the instructions include an example of how each entry should look like, as well as some warning points to be aware of. These warning points emphasize the importance of the format each element in the database should follow. Figure 8, which is a continuation of Figure 7, shows some elements that are already in the database. As mentioned in the set of instructions, each element makes up a total of three (3) rows. The first row of each element is numbered, in numerical order, of which it appears in the database. The next row contains the name of the test organism, or non-human organism. The last row contains the list of codons that are favored by the organism for each of the twenty (20) amino acids.

The Humanizer reads in information from the text file line by line. It keeps track of the test organisms in the `Codon_bias_DB.txt` file by storing their names in a list called the `Codon_Bias_Directory`. To do this, the application uses each record's first line to indicate where to store the organism's name, which is given in the following line, in the `codon_bias_directory`. Then, it reads the next line, a long string of all the codon biases for that organism, and stores them in the codon bias table, `codon_bias_2D_list`. This repeats until the end of the file is reached. The `codon_bias_list` method is used primarily for this purpose. This process allows the program to verify whether a codon bias for a user-given test organism is provided before it continues further, and also allows the Humanizer to look up the row number in which a particular organism's information is stored.

Codon_Bias_2D_List

	0	1	2	...	18	19
0	A	R	N	...	Y	V
1	GCC	CGC	AAC	...	TAC	GTG
2	GCC	CGG	AAC	...	TAC	GTG
3	GCT	AGA	AAC	...	TAC	GTG

Codon_Bias_Directory

0	
1	drosophila melanogaster
2	mus musculus
...	...
7	saccharomyces cerevisiae

Figure 9: Codon_Bias_2D_List and Codon_Bias_Directory

Next, behind the scenes, the Humanizer generates a 2D-list of the codon bias table. The first row of the array is a list of the twenty (20) amino acids in their single-letter abbreviated form. The following rows each correspond to a different test organism. Each cell contains the codon bias for its row organism and amino-acid column. It also generates a separate list of the organism's names, where the location of each name is the same as the number of the row in which that organism's information is stored in the codon bias 2D-list. This list is called the codon_bias_directory. Examples of both lists are shown in Figure 9. This figure shows that the test organism 'drosophila melanogaster' is located in index one (1) of the codon bias directory. Thus, in row one (1) of the codon bias 2D list, shows the codon bias of each amino acid within the model organism drosophila melanogaster. For amino acid Alanine (A), drosophila melanogaster's codon bias is 'GCC', for Arginine (R) its codon bias is 'CGC', and so forth. A complete list of the full names of each amino acid and their one-letter abbreviation is provided in the 'Codon_Bias_DB.txt' file.

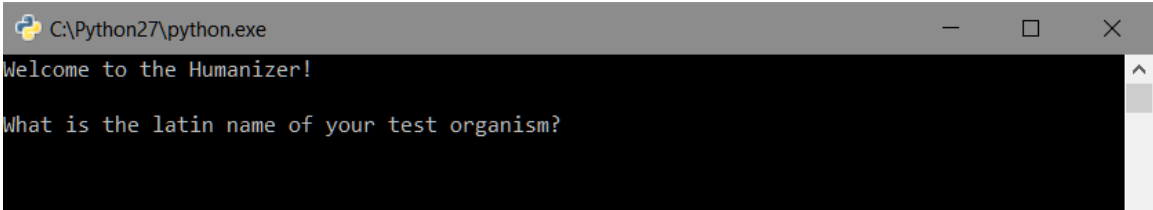


Figure 10: Welcome Greeting

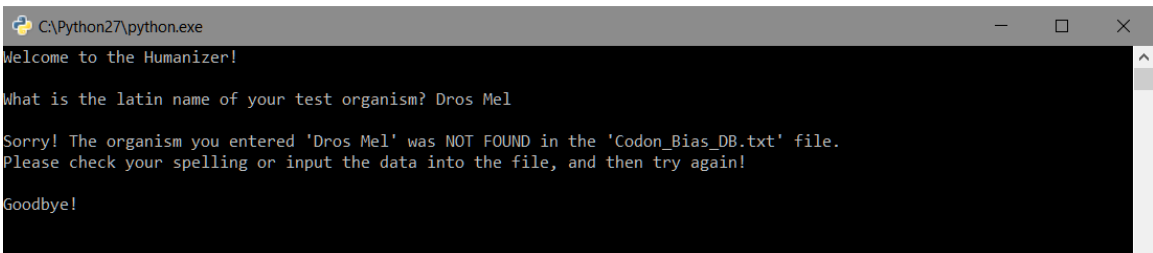
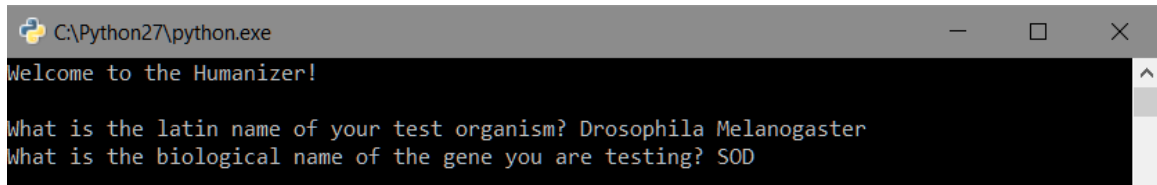


Figure 11: Test Organism Error

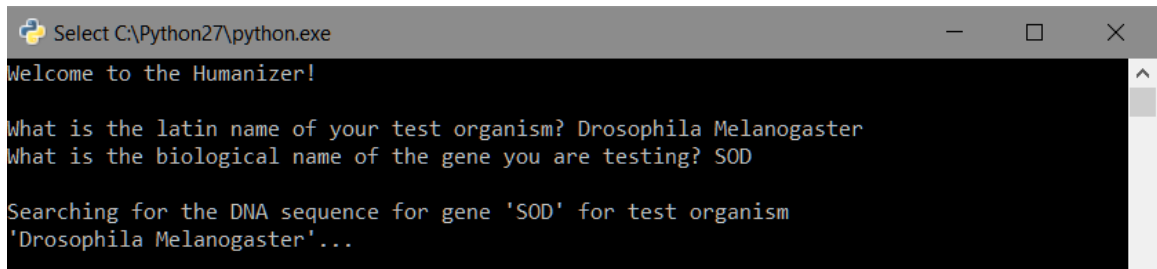
After the application loads the database into its memory, it then greets the user. This is the first thing the user sees. Displayed in Figure 10, the user is welcomed, and then the Humanizer immediately asks for the name of the test organism whose gene the user will be humanizing. Although it is not case-sensitive, the name must be spelled exactly how it appears in the "Codon_Bias_DB.txt" file. If not, the user will receive an error message as seen in Figure 11. This message will end the program after advising the user to check that the database is up to date and accurate.

After creating the 2D-list of the codon bias table, and the *codon bias directory* list, the program welcomes the user and asks for the Latin name of the test organism along with the name of the gene undergoing testing. It is at this time that, after input, the program checks in the *codon bias directory* to make sure the organism's codon bias is in the codon bias database. If it does, the program continues onto the next step. If not, an error message is displayed to the user, and the program ends.



```
C:\Python27\python.exe
Welcome to the Humanizer!
What is the latin name of your test organism? Drosophila Melanogaster
What is the biological name of the gene you are testing? SOD
```

Figure 12: Test Organism and Gene



```
Select C:\Python27\python.exe
Welcome to the Humanizer!
What is the latin name of your test organism? Drosophila Melanogaster
What is the biological name of the gene you are testing? SOD
Searching for the DNA sequence for gene 'SOD' for test organism
'Drosophila Melanogaster'...
```

Figure 13: Entrez Search

Next, the application will prompt the user for the name of the biological gene, which the user will be humanizing. An example of this screen is shown in Figure 12. After the user successfully enters in both of these fields, the Humanizer now searches for DNA sequences that match the organism and gene which the user had entered, as shown in Figure 13.

```
Select C:\Python27\python.exe
Welcome to the Humanizer!

What is the latin name of your test organism? Drosophila Melanogaster
What is the biological name of the gene you are testing? SOD

Searching for the DNA sequence for gene 'SOD' for test organism
'Drosophila Melanogaster'...

Results: 14

The results are printed below along with an INDEX number of each result.

INDEX: 1
ID: 671162317
gi|671162317|ref|NT_037436.4|[671162317]
Drosophila melanogaster chromosome 3L
Length: 28110227

INDEX: 2
ID: 671162315
gi|671162315|ref|NT_033778.4|[671162315]
Drosophila melanogaster chromosome 2R
```

Figure 14: Entrez Results

The program queries for this search with NCBI's ENTREZ tool which includes multiple modules for performing certain tasks. Using the 'search' module. This module allows for the application to make a search through any databases supported by NCBI. In this application, and in this search specifically, the query is bounded to the nucleotide database since we are looking for a DNA sequence. The method *search_nucleotide_seq* allows the program to perform this task. It uses the user-inputted organism and gene as parameters to search for DNA sequences that match the query.

Then, using the 'summary' module, ENTREZ allows the program to extract certain information found in each result to display to the user. This is done in the method *nucleotide_esummary*. Results are sorted with the best match first. Only the top twenty (20) entries will be displayed. A maximum has been set because of the high possibility that thousands of results may be returned. The number of results will be shown immediately after the search has completed, and the results will follow, shown in Figure 14 presents. An index is included with each search result for identification purposes within this application only. The following row of each result is a unique ID tag that identifies its record, while the next row includes various identifiers each result goes by within various NCBI databases. The fourth row is the name/description of each result, while the last row shows how many nucleotide bases long the gene is.

```
C:\Python27\python.exe
INDEX: 12
ID: 17946027
gi|17946027|gb|AY071435.1|[17946027]
Drosophila melanogaster RE52090 full length cDNA
Length: 747

INDEX: 13
ID: 220957721
gi|220957721|gb|FJ637145.1|[220957721]
Synthetic construct Drosophila melanogaster clone BS12339 encodes Sod-RA
Length: 494

INDEX: 14
ID: 7792
gi|7792|emb|Z19591.1|[7792]
D.melanogaster Cu-Zn superoxide dismutase gene
Length: 1844

Type in the INDEX of the DNA record you want: 11
```

Figure 15: Select Test-Organism DNA Sequence Record

After all the results are displayed, the next step prompts the user to carefully examine the results and choose the result that best match their needs. The user is prompted for the index of the record they wish to humanize. NCBI allows records to be manually added in by their users, thus, duplicates and errors are a possibility within the databases. Therefore, at this point, the expertise of the user is required to carefully identify and select a result that suits their needs. In Figure 15, record 11 is selected as the DNA sequence that will undergo humanization.

```
C:\Python27\python.exe
Type in the INDEX of the DNA record you want: 11

Returning non-human DNA sequence record for result id: 11

NON-HUMAN DNA SEQUENCE RECORD:
ID: M24421.1
Name: M24421.1
Description: M24421.1 Drosophila melanogaster Cu-Zn superoxide dismutase (SOD) gene, complete cds
Number of features: 0
Seq('ATGGTGTTAAAGCTGTCTGCGTAATTAACGGCGATGCCAAGGGCACGGTTTTC...TAA', SingleLetterAlphabet())

*****
**** BLASTING NON-HUMAN DNA SEQ TO RETRIEVE NON-HUMAN PROTEIN SEQ ****
*****

NON_HUMAN PROTEIN SEQUENCE RECORD:
ID: NP_476735.1
Name: NP_476735.1
Description: NP_476735.1 superoxide dismutase 1, isoform A [Drosophila melanogaster]
Number of features: 0
Seq('MVKAVCVINGDAKGTVFFEQESSGTPVKVSGEVCGLAKLGHGFHVHEFGDNTN...AKV', SingleLetterAlphabet())
```

Figure 16: BLASTing to Retrieve Test-Organism Protein Sequence

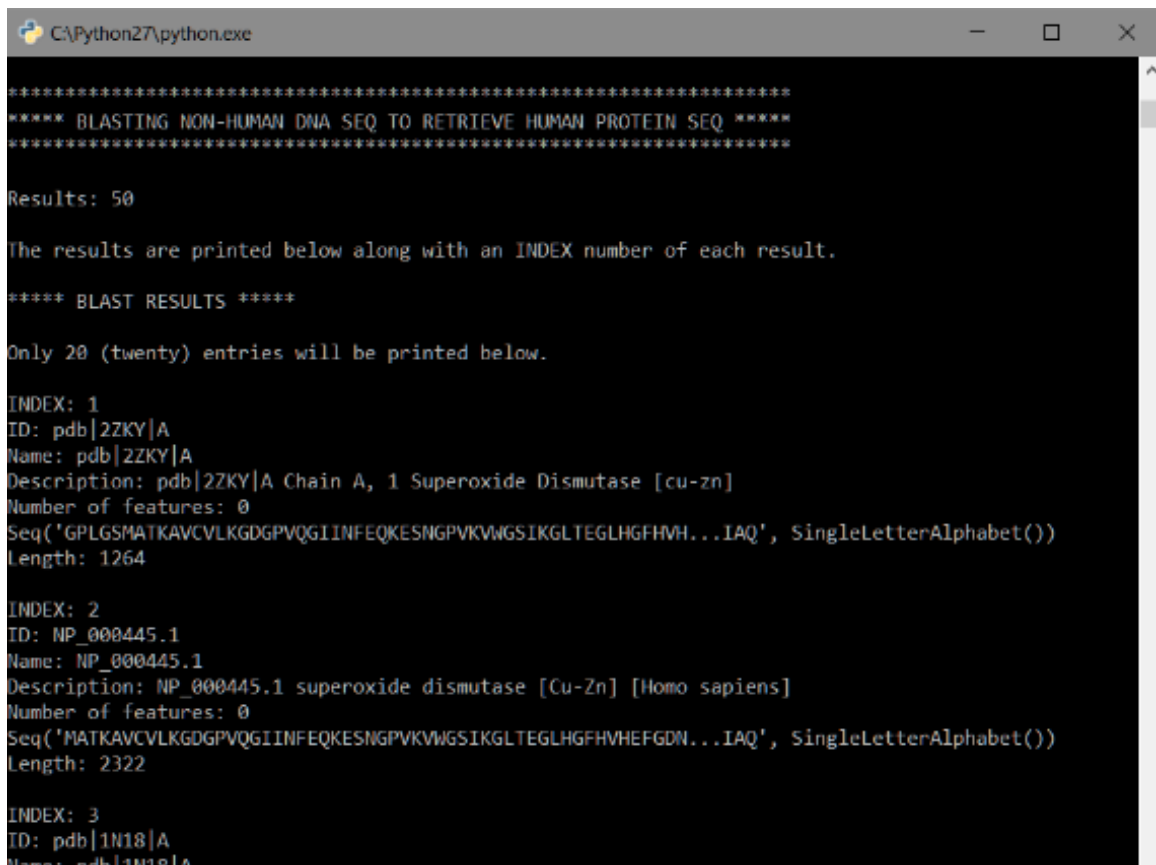
Once the DNA sequence is selected, it's index is used in the method *fetch seq* to grab the record's data, including its personal identification tag. The application returns this result, once again, to the user as a confirmation of what the user has chosen. In Figure 16, the record's ID, name, description, number of features, and a portion of the sequence appears. This returned format is called FASTA, a commonly used sequence format. Having the record formatted in FASTA allows it to be recognized as a sequence, and then used throughout the program.

The FASTA record is first put to use through the two BLASTX searches that follow. In the first BLAST search, the program runs the DNA sequence against the same test organism to get a complementary protein sequence for that test organism. In other words, this BLAST process determines the protein sequence record that is encoded by the DNA sequence selected earlier. The method *blastx.nucleotide_seq* will perform this task and return the first result, which is also the result with the closest match, to the user. Figure 16 shows this first BLAST result.

The sequence inputted into the BLAST search in Figure 16 is the non-human DNA sequence record. It is first translated into three (3) different protein sequences, one for each reading frame, which will be explained further in this section, and then taking those sequences a pairwise alignment is done against

the protein database for the protein sequence that represents each sequence. The default database in BLASTX is the non-redundant (nr) database which contains sequences from Genbank, Protein Data Bank (PDB), SwissProt, PIR, and other data banks supported by NCBI. In total, there is approximately 65 million protein sequences in the non-redundant database. Therefore, it is important to keep in mind that when using the BLAST tool, it can take as long as a few seconds to a couple of hours.

As the BLAST search concludes, only one record is displayed to the user: the first record, the record with the best score. Therefore, it concludes that the record shown in Figure 16 will represent the translated DNA sequence, the protein sequence of the test organism.



```
C:\Python27\python.exe
*****
***** BLASTING NON-HUMAN DNA SEQ TO RETRIEVE HUMAN PROTEIN SEQ *****
*****
Results: 50

The results are printed below along with an INDEX number of each result.

***** BLAST RESULTS *****

Only 20 (twenty) entries will be printed below.

INDEX: 1
ID: pdb|2ZKY|A
Name: pdb|2ZKY|A
Description: pdb|2ZKY|A Chain A, 1 Superoxide Dismutase [cu-zn]
Number of features: 0
Seq('GPLGSMATKAVCVLKGDPVQGIINFEQKESNGPVKVGSIKGLTEGLHGFHVH...IAQ', SingleLetterAlphabet())
Length: 1264

INDEX: 2
ID: NP_000445.1
Name: NP_000445.1
Description: NP_000445.1 superoxide dismutase [Cu-Zn] [Homo sapiens]
Number of features: 0
Seq('MATKAVCVLKGDPVQGIINFEQKESNGPVKVGSIKGLTEGLHGFHVHEFGDN...IAQ', SingleLetterAlphabet())
Length: 2322

INDEX: 3
ID: pdb|1N18|A
Name: pdb|1N18|A
```

Figure 17: BLASTing to Retrieve Human Protein Sequence

The second BLASTX search is performed right after the test organism's protein sequence is retrieved. The program is BLASTing the test organism's DNA sequence again, but this time it is limiting the search to protein sequences found in humans (homo sapiens) only. This limitation falls under one of the

many optional parameters allowed in a BLAST search. In the previous BLAST search, there were no organism limitation, thus the program only looked for protein sequences related to the DNA sequence within its own organism, the given test organism. A different method, *blastx_nucleotide_seq_list* will provide the user the results as a list. Having the results shown as a list will allow the user to choose the result they prefer because a number of records related to the DNA sequence will appear to the user as shown in Figure 17. The results, like in Figure 16, are returned in FASTA format using the method *fetch_seq* which connects to the fetch module in ENTREZ. A maximum of twenty (20) results will be displayed to the user. This maximum has been set because of the high possibility that thousands of results may be returned. Remember that the most likely aligned sequences are shown first.

```

C:\Python27\python.exe
INDEX: 19
ID: pdb|2WYZ|A
Name: pdb|2WYZ|A
Description: pdb|2WYZ|A Chain A, 1 Superoxide Dismutase [cu-zn]
Number of features: 0
Seq('ATKAVCVLKGDPVQGIINFEQKESNGPVKVGSIKGVTEGLHGFHVHEFGDNT...IAQ', SingleLetterAlphabet())
Length: 661

INDEX: 20
ID: pdb|2GBT|A
Name: pdb|2GBT|A
Description: pdb|2GBT|A Chain A, 1 Superoxide Dismutase [cu-zn]
Number of features: 0
Seq('ATKAVAVLKGDPVQGIINFEQKESNGPVKVGSIKGLTEGLHGFHVHEFGDNT...IAQ', SingleLetterAlphabet())
Length: 751

Type in the INDEX of the record you want: 3

```

Figure 18: Select Human Protein Sequence Record

After all the results are displayed, the user runs upon another inquiry which asks for the index of the human protein sequence they wish to use. Like the process that occurred in Figure 14 and Figure 15, an index number is included in the results for easier identification of each record. Again, this step would require the expertise of the user to identify and select the record that best meets their needs. In Figure 18, index 3 is inputted. Once the program receives the input, it prints out the record again to the screen in FASTA format using the method *fetch_seq*. This can be seen in the top half of Figure 18. At the end of these two blast algorithms, the program will have a DNA and protein sequence for the user-given gene in the user-given organism, as well as the protein sequence for the user-given gene in humans.

Reading frame #1

5'-AGUCUUACCGCAUUGUGG-3'
| | | | | |
Ser--Leu--Thr--Ala--Leu-Ser

Reading frame #2

5'-AGUCUUACCGCAUUGUGG-3'
| | | | | |
Val--Leu--Pro--His--Cys

Reading frame #3

5'-AGUCUUACCGCAUUGUGG-3'
| | | | | |
Ser--Tyr--Arg--Ile--Val

Figure 20: 3-Frame Translation

source: <https://classes.engineering.wustl.edu/cse131/extensions/frame.jpg>

After aligning the two sequences, the program performs a 3-frame translation on the DNA sequence acquired previously for the test organism. As stated prior about the construction of a codon, three (3) nucleotide bases code for an amino acid, and groups of amino acids create a protein. The gene provides instructions on how to create the protein which includes the individual nucleotides that code for it. Unfortunately, the DNA sequence acquired in Figure 16 does not always begin with a full codon. In other words, because a codon consists of three (3) nucleotide bases the DNA sequence from the ENTREZ search could have returned the DNA sequence starting with a full codon, at the second position of a codon, or even the last position of a codon. This totals up to three (3) different reading frames as seen in Figure 20. Therefore, a 3-frame translation is required to determine how the DNA sequence should be read.

Thus, the next step in the program is to determine which reading frame is the most accurate. The program uses a loop that runs three times, and basically reads the DNA sequence with a different starting point each time, either from the very beginning, from the second position, or the third position. Each time, it translates the whole DNA sequence. This gives us three (3) different pro-

tein sequences which are then put into a pairwise alignment, pairwise2, with the protein sequence acquired from Figure 16, the first blast result. Pairwise2 returns a score of each comparison, and the pair that returns the highest score is concluded as the correct reading frame.

With the correct reading frame determined, we now need to detect where in the DNA sequence the instructions for the protein begins and ends, because unfortunately the DNA sequence also does not begin at the start of the coding region. This is called the open reading frame. The DNA sequence can have nucleotide bases before the instructions begin and after the instructions end, both of which are not needed for humanization. In addition to the codons that code for amino acids, there are also codons that determine the start of a coding region and the end of a coding region. These are called start and stop codons. In Figure 3, the start codon that codes for Methionine (Met) is 'ATG', and the stop codons are marked as 'stop' in red. With this information, we can go through each codon in the DNA strand and find the one that codes for the start codon. All the codons before the start codon can then be extracted since we have no need for them. Also, keep in mind that we have the protein sequence which includes all the amino acids that code for the protein, thus we can get the length of the protein sequence and use it to determine the end of the DNA strand.

```

register_ncbi_table(name='Standard',
                  alt_name='SGC0', id=1,
                  table={
    'TTT': 'F', 'TTC': 'F', 'TTA': 'L', 'TTG': 'L', 'TCT': 'S',
    'TCC': 'S', 'TCA': 'S', 'TCG': 'S', 'TAT': 'Y', 'TAC': 'Y',
    'TGT': 'C', 'TGC': 'C', 'TGG': 'W', 'CTT': 'L', 'CTC': 'L',
    'CTA': 'L', 'CTG': 'L', 'CCT': 'P', 'CCC': 'P', 'CCA': 'P',
    'CCG': 'P', 'CAT': 'H', 'CAC': 'H', 'CAA': 'Q', 'CAG': 'Q',
    'CGT': 'R', 'CGC': 'R', 'CGA': 'R', 'CGG': 'R', 'ATT': 'I',
    'ATC': 'I', 'ATA': 'I', 'ATG': 'M', 'ACT': 'T', 'ACC': 'T',
    'ACA': 'T', 'ACG': 'T', 'AAT': 'N', 'AAC': 'N', 'AAA': 'K',
    'AAG': 'K', 'AGT': 'S', 'AGC': 'S', 'AGA': 'R', 'AGG': 'R',
    'GTT': 'V', 'GTC': 'V', 'GTA': 'V', 'GTG': 'V', 'GCT': 'A',
    'GCC': 'A', 'GCA': 'A', 'GCG': 'A', 'GAT': 'D', 'GAC': 'D',
    'GAA': 'E', 'GAG': 'E', 'GGT': 'G', 'GGC': 'G', 'GGA': 'G',
    'GGG': 'G', },
    stop_codons=['TAA', 'TAG', 'TGA'],
    start_codons=['TTG', 'CTG', 'ATG'])

```

Figure 21: BioPython: Codon Table

source:

<http://biopython.org/DIST/docs/api/Bio.Data.CodonTable-pysrc.html>

The program calls the *find_DNA_from_reading_frame* method to find the start and stop codons in the DNA strand. Then it extracts the codons that are not required. In BioPython, there are built in codon tables like the one shown in Figure 3. This codon table is shown in figure 21. This provides the program with lists of start and stop codons which tells the program what to look out for in the DNA sequence reading. As seen in Figure 21, there exists three (3) different start codons: 'TTG', 'CTG', and 'ATG'. Though, since it is rare for genes to have a start codon other than 'ATG', the Humanizer will only look for the start codon 'ATG' in DNA sequences. The list of start and stop codon gets sent as a parameter into the *find_DNA_from_reading_frame* method, as well as the reading frame indicator.

First, the method looks for a start codon by reading the DNA sequence in its correct reading frame and comparing it the start codon parameter. After finding the start codon in the DNA strand, all codons prior are extracted and the program now seeks for the stop codon by going through each subsequent codon. While doing so, it also keeps a count of how many codons it has looped through, until it finds a stop codon. Because there are three possible stop codons, the program will compare each codon in the DNA strand to the three (3) options until there is a match. When there is a match, the program will also make sure the count it kept track of earlier matches the length of the protein sequence. Once the stop codon is found, all codons that follow will also be extracted.

By this time, we will have an updated DNA sequence of only its open reading frame. This updated DNA sequence is stored in a variable called 'new_dna_seq' in the Humanizer. Now, we can finally humanize the gene!

Position	1	2	3	4	5	6	7												
Non-human DNA Sequence (test organism)	A	T	G	G	T	T	C	G	C	A	G	A	T	T	T	C	C	C	-
Non-human Protein Sequence (test organism)	M	V	R	R	F	P	-												
Human Protein Sequence	M	V	P	-	F	P	K												
			1	3			2												

Figure 22: The humanizing process

Finally, taking the two protein sequences, the Humanizer compares each individual amino acid of both sequences at each position until it reaches the ends of the sequences. Only when the application notices a discrepancy between the amino acids will it make any changes to the DNA sequence updated earlier, 'new_dna_seq'. These forthcoming changes to 'new_dna_seq' will be saved in a new variable called 'modified_dna_seq'. The 'modified_dna_seq' variable will represent the humanized DNA sequence. It starts off empty and fills up as codons are appended to it.

If amino acids in the test organism sequence and the amino acid sequence match, the codon that codes for the amino acid of the test organism will be appended to the variable 'modified_dna_seq'.

If the amino acid in the test organism sequence and the amino acid in the human sequence do not match, like in Figure 22 highlighted in red and marked 1, the application will change the codon in the DNA sequence and make it code for the amino acid found in the human sequence at that location. This is where the data from the "Codon_Bias_DB" comes in.

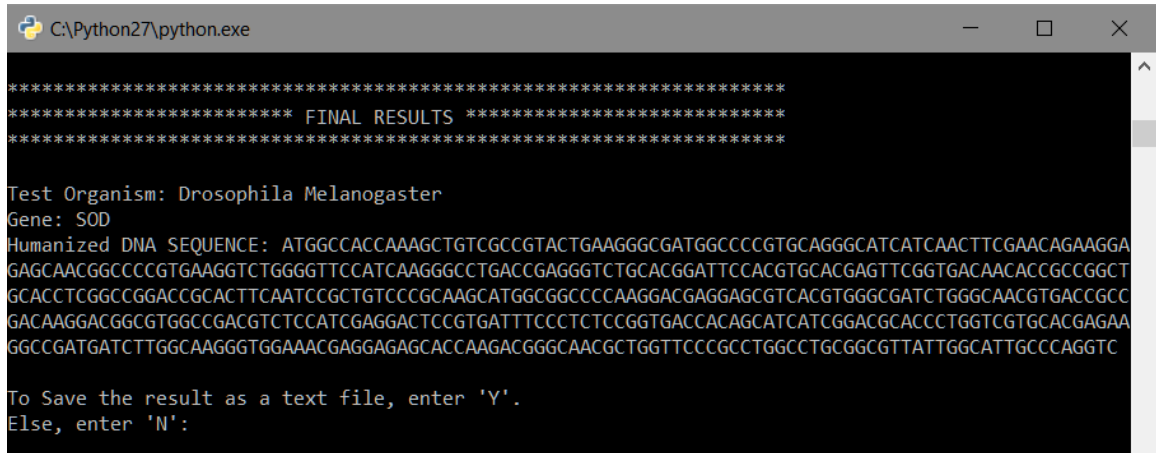
In Figure 22, the non-human protein sequence has a 'R', for Arginine, and the human protein sequence has a 'P', for Proline, in position 3. The application then calls the *get_amino_acid_index* method and puts the amino acid 'P' (Proline) in the parameter. It then finds the row of the test organism in the "Codon_Bias_DB" list, that stores the test organism's codon bias metadata, and grab the codon bias that codes for amino acid 'P' (Proline). An image of the "Codon_Bias_DB" list from Figure 8, shows that the row for test organism *Drosophila Melanogaster* is the first row. The test organism, *Drosophila Melanogaster*'s, codon bias for Proline (P) is 'CCC'. Thus, instead of the codon 'CGC' that codes for Arginine (R) being added to the 'modified_dna_seq' variable, its codon bias for Proline (P), 'CCC', will be appended.

If the non-human organism has a gap as highlighted in Figure 22 and marked 2, the program will append the codon that the non-human organism favors for the amino acid 'K', Lysine, found in the human protein sequence at that same position. It does this by calling the *get_amino_acid_index* method, goes into the correct row in the Codon_Bias_DB table, and finding the codon bias for Lysine (K).

Lastly, if the human protein sequence is the sequence with a gap in any of its positions, marked 3 in Figure 22, nothing will be appended to 'modified_dna_seq'. We only want to add to the DNA sequence if something is missing or the amino acids do not match. Other than that, we will not be appending any codons to the humanized sequence.

Although there is more than one way to construct many of the amino acids, we keep to the organism's codon bias, the preferred codon, that codes for the

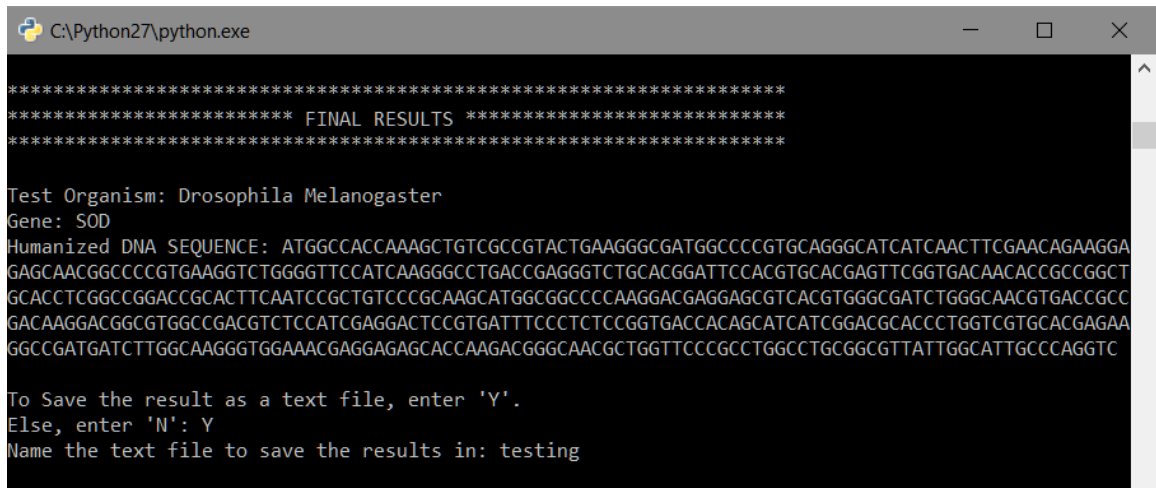
amino acid in the case the chemical properties present in the codon is required for the organism's natural bodily functions or prevents it. Because, as stated earlier, we do not want to disrupt any organism's natural way of functioning.



```
C:\Python27\python.exe
*****
***** FINAL RESULTS *****
*****
Test Organism: Drosophila Melanogaster
Gene: SOD
Humanized DNA SEQUENCE: ATGGCCACCAAAGCTGTCGCCGTAAGGGCGATGGCCCGTGCAGGGCATCATCAACTTCGAACAGAAGGA
GAGCAACGGCCCGTGAAGGTCTGGGGTTCATCAAGGGCCTGACCGAGGGTCTGCACGGATTCCACGTGCACGAGTTCGGTGACAACACCGCCGGCT
GCACCTCGGCCGGACCGCACTTCAATCCGCTGTCCGCAAGCATGGCGGCCCAAGGACGAGGAGCGTCACGTGGGCGATCTGGCAACGTGACCGCC
GACAAGGACGGCGTGGCCGACGTCTCCATCGAGGACTCCGTGATTTCCCTCCTCCGGTGACCACAGCATCATCGGACGCACCCTGGTCTGCACGAGAA
GGCCGATGATCTTGGCAAGGGTGAAACGAGGAGAGCACCAGACGGGCAACGCTGGTTCCCGCTGGCTGCGGCCGTTATTGGCATTGCCAGGTC

To Save the result as a text file, enter 'Y'.
Else, enter 'N':
```

Figure 23: Returns Humanized DNA Sequence and Prompts to Save Sequence



```
C:\Python27\python.exe
*****
***** FINAL RESULTS *****
*****
Test Organism: Drosophila Melanogaster
Gene: SOD
Humanized DNA SEQUENCE: ATGGCCACCAAAGCTGTCGCCGTAAGGGCGATGGCCCGTGCAGGGCATCATCAACTTCGAACAGAAGGA
GAGCAACGGCCCGTGAAGGTCTGGGGTTCATCAAGGGCCTGACCGAGGGTCTGCACGGATTCCACGTGCACGAGTTCGGTGACAACACCGCCGGCT
GCACCTCGGCCGGACCGCACTTCAATCCGCTGTCCGCAAGCATGGCGGCCCAAGGACGAGGAGCGTCACGTGGGCGATCTGGCAACGTGACCGCC
GACAAGGACGGCGTGGCCGACGTCTCCATCGAGGACTCCGTGATTTCCCTCCTCCGGTGACCACAGCATCATCGGACGCACCCTGGTCTGCACGAGAA
GGCCGATGATCTTGGCAAGGGTGAAACGAGGAGAGCACCAGACGGGCAACGCTGGTTCCCGCTGGCTGCGGCCGTTATTGGCATTGCCAGGTC

To Save the result as a text file, enter 'Y'.
Else, enter 'N': Y
Name the text file to save the results in: testing
```

Figure 24: Saving results in a text file on user's desktop

```

C:\Python27\python.exe
*****
***** FINAL RESULTS *****
*****
Test Organism: Drosophila Melanogaster
Gene: SOD
Humanized DNA SEQUENCE: ATGGCCACCAAAGCTGTCGCCGTAAGGGCGATGGCCCCGTGCAGGGCATCATCAACTTCGAACAGAAGGA
GAGCAACGGCCCCGTGAAGGTCTGGGGTTCATCAAGGGCCTGACCGAGGGTCTGCACGGATTCCACGTGCACGAGTTCGGTGACAACACCGCCGGCT
GCACCTCGGCCGGACCGCACTTCAATCCGCTGTCCCGAAGCATGGCGGCCCAAGGACGAGGAGCGTCACGTGGGCGATCTGGGCAACGTGACCGCC
GACAAGGACGGCGTGGCCGACGCTCCATCGAGGACTCCGTGATTCCTCTCCGGTGACACAGCATCATCGGACGCACCCTGGTCTGCACGAGAA
GGCCGATGATCTTGGCAAGGGTGGAAACGAGGAGAGCACCAAGACGGCAACGCTGTTCCCGCTGGCTGCGGCCGTTATTGGCATTGCCAGGTC

To Save the result as a text file, enter 'Y'.
Else, enter 'N': Y
Name the text file to save the results in: testing

The result has been saved in the 'testing.txt' file on your desktop.
Thank you for using 'The Humanizer'. Goodbye!

```

Figure 25: End of Program

This humanizing step continues until the program reaches the end of the protein sequences. Once that is completed, the gene has been humanized and the user sees a screen similar to that shown in Figure 23. The name of the test organism and the gene, both given by the user at the beginning of the program, and the humanized DNA sequence are all printed to the screen. In addition, the user sees the last prompt of the program, seen in Figure 23. The user has a choice of saving the results into a text file. If the user enters 'N', for no, the program immediately ends after printing out a farewell. If the user enter 'Y', for yes, the user is asked to type in the name of the text file they wish to save the results in. In Figure 24, 'testing' is entered. Finally, the program tells the user that the file has been saved to their desktop, prints out a farewell, and the program ends, shown in Figure 25.

```

testing.txt - Notepad
File Edit Format View Help
Test Organism: Drosophila Melanogaster
Gene: SOD
Humanized DNA SEQUENCE:
ATGGCCACCAAAGCTGTCGCCGTAAGGGCGATGGCCCCGTGCAGGGCATCATCAACTTCGAACAGAAGGAGAGCAACGGCCCCGTGAAGGTCTGGGGTTCATCAAGGGCCTGACCGAGGGTCTGCACG
GATTCCACGTGCACGAGTTCGGTGACAACACCGCCGGCTGCACCTCGGCCGGACCGCACTTCAATCCGCTGTCCCGAAGCATGGCGGCCCAAGGACGAGGAGCGTCACGTGGGCGATCTGGGCAACGTGAC
CGCCGACAAGGACGGCGTGGCCGACGCTCCATCGAGGACTCCGTGATTCCTCTCCGGTGACACAGCATCATCGGACGCACCCTGGTCTGTGCACGAGAAGGGCGATGATCTTGGCAAGGGTGGAAACGAG
GAGAGCACCAAGACGGGCAACGCTGTTCCCGCTGGCTGCGGCCGTTATTGGCATTGCCAGGTC

```

Figure 26: Text file of saved results

When the user goes to their desktop, the text file would have been saved with the name the user had inputted. If the file does not already exist, the program

will create that file for the user, if it does exist, the file will be saved over the previous file. Figure 26 shows the record that was saved which includes the name of the test organism, the name of the gene, and the humanized DNA sequence.

5 Testing

5.1 Functional testing

Dr. Stilwell and I ran the Humanizer using three sets of different organisms and genes to ensure the functionality of each module within the system. Although its interface is quite simple, at this moment, for its users, its functionality is more important.

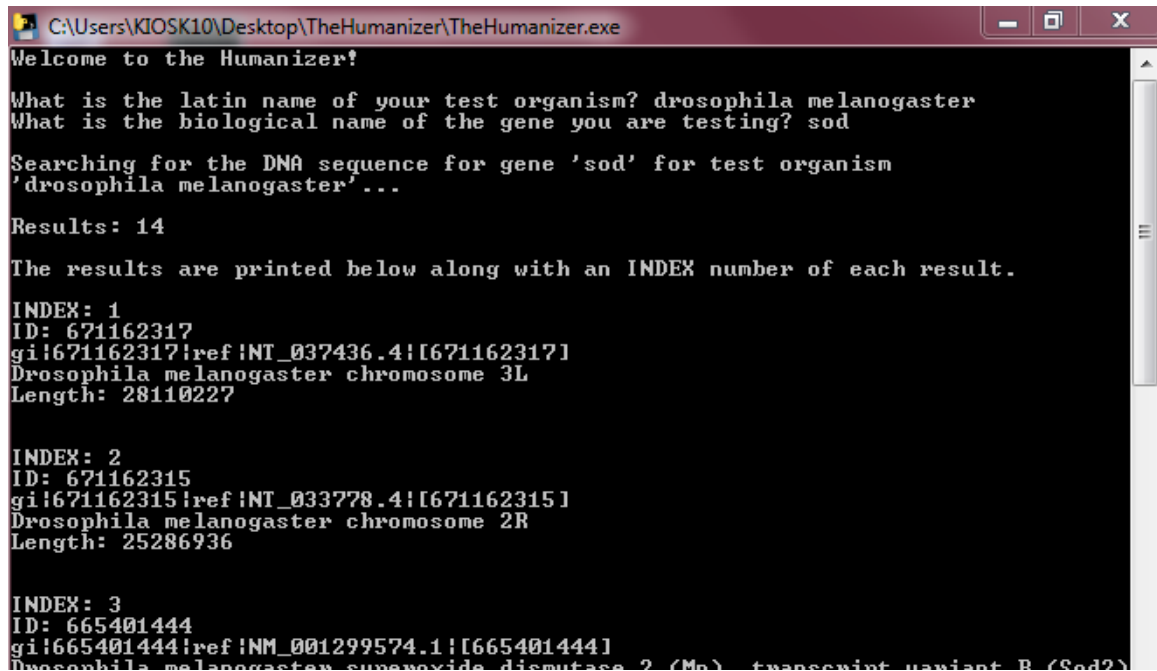


Figure 27: Automatic Nucleotide Search with the Humanizer: finding the test organism's DNA sequence

The screenshot shows the NCBI Nucleotide search interface. The search bar contains the query: "(Drosophila Melanogaster[Organism]) AND SOD[Gene Name]". The search results are displayed in a list format, showing 14 items. The first item is "Drosophila melanogaster chromosome 3L", which is a 28,110,227 bp linear DNA sequence. The second item is "Drosophila melanogaster chromosome 2R", a 25,286,936 bp linear DNA sequence. The third item is "Drosophila melanogaster superoxide dismutase 2 (Mn), transcript variant B (Sod2), mRNA", a 1,015 bp linear mRNA sequence. The fourth item is "Drosophila melanogaster superoxide dismutase 2 (Mn), transcript variant A (Sod2), mRNA", an 884 bp linear mRNA sequence. The fifth item is "Drosophila melanogaster superoxide dismutase 1, transcript variant D (Sod1), mRNA". The interface includes various filters and options on the left side, such as "Species", "Molecule types", "Source databases", "Sequence length", "Release date", and "Revision date". On the right side, there are sections for "Results by taxon", "Analyze these sequences", "Find related data", "Search details", and "Recent activity".

Figure 28: Manual Nucleotide Search: finding the test organism’s DNA sequence

One of the tests we ran uses the model organism 'drosophila melanogaster', also known as fruit flies, and the gene 'SOD' which is the gene that causes ALS and also the gene Dr. Stilwell is currently studying.

First, as discussed in Section 4.1, we entered the name of the test organism that will host the humanized gene, and the name of the gene we are studying. Thus, we entered in 'Drosophila Melanogaster' and 'SOD' respectively. This query is then searched against NCBI’s nucleotide database through ENTREZ, and we got the same fourteen (14) results as we did if we performed this step by hand. Figure 27 show the results from the Humanizer application which performed this automatically, and Figure 28 show the results from the manual NCBI nucleotide search, which we also ran.


```

C:\Users\KIOSK10\Desktop\TheHumanizer\TheHumanizer.exe
Type in the INDEX of the DNA record you want: 11

Returning non-human DNA sequence record for result id: 11

NON-HUMAN DNA SEQUENCE RECORD:
ID: M24421.1
Name: M24421.1
Description: M24421.1 Drosophila melanogaster Cu-Zn superoxide dismutase (SOD) gene, complete cds
Number of features: 0
Seq(<'ATGGTGGTTAAAGCTGCTGCCTAATTACGGCGATGCCAAGGGCACGGTTTTC...TAA', SingleLetterAlphabet(>>)

*****
***** BLASTING NON-HUMAN DNA SEQ TO RETRIEVE NON-HUMAN PROTEIN SEQ *****
*****

NON_HUMAN PROTEIN SEQUENCE RECORD:
ID: NP_476735.1
Name: NP_476735.1
Description: NP_476735.1 superoxide dismutase 1, isoform A [Drosophila melanogaster]
Number of features: 0
Seq(<'MUUKAUCUINGDARGTUFFEQESSGTPUKUSGEUCGLAKGLHGFHUHEFGDNTN...AKU', SingleLetterAlphabet(>>)

```

Figure 29: Automatic BLASTX Search: finding the test organism’s protein sequence

Sequences producing significant alignments:

Select: All None Selected: 0

Alignments Download GenPlot Graphics

Description	Max score	Total score	Query cover	E value	Ident	Accession
<input type="checkbox"/> superoxide dismutase 1, isoform A [Drosophila melanogaster]	307	307	99%	7e-106	100%	NP_476735.1
<input type="checkbox"/> Cu-Zn superoxide dismutase [Drosophila melanogaster]	305	305	99%	5e-105	99%	CAA35210.1
<input type="checkbox"/> RecName: Full=Superoxide dismutase [Cu-Zn]	303	303	99%	3e-104	99%	Q8L4X2.3
<input type="checkbox"/> RecName: Full=Superoxide dismutase [Cu-Zn]	301	301	99%	2e-103	98%	Q8L4X3.3
<input type="checkbox"/> RecName: Full=Superoxide dismutase [Cu-Zn]	301	301	99%	4e-103	97%	Q8L4X5.3
<input type="checkbox"/> Sod [Drosophila yakuba]	300	300	99%	7e-103	97%	XP_002094375.1
<input type="checkbox"/> Sod [Drosophila erecta]	299	299	99%	2e-102	97%	XP_001972349.1
<input type="checkbox"/> superoxide dismutase 1, isoform D [Drosophila melanogaster]	298	298	99%	4e-102	92%	NP_001261700.1

Figure 30: Manual BLASTX Search: finding the test organism’s protein sequence

The next step, after we selected the nucleotide sequence we preferred, the first BLAST search is done. We selected result number eleven (11). In the automatic process with the Humanizer, the best result, which is also the first result, has an ID of 'NP_476735.1', as seen in Figure 29. In the manual process, the first result in Figure 30 also displays the same record whose ID is 'NP_476735.1', after we BLASTed record number 11.

```

C:\Users\KIOSK10\Desktop\TheHumanizer\TheHumanizer.exe
***** BLASTING NON-HUMAN DNA SEQ TO RETRIEVE HUMAN PROTEIN SEQ *****
***** BLAST RESULTS *****

Results: 50

The results are printed below along with an INDEX number of each result.

***** BLAST RESULTS *****

Only 20 entries will be printed below.

INDEX: 1
ID: pdb:1ZKY:A
Name: pdb:1ZKY:A
Description: pdb:1ZKY:A Chain A, 1 Superoxide Dismutase [cu-zn]
Number of features: 0
Seq<' GPLGSMATKAUCULKGDGPUQGI INFEQKESNGPUKUWGS IKGLTEGLHGFHVV...IAQ', SingleLetter
Alphabet(>>)
Length: 1264

INDEX: 2
ID: NP_000445.1
Name: NP_000445.1
Description: NP_000445.1 superoxide dismutase [Cu-Zn] [Homo sapiens]
Number of features: 0
Seq<' MATKAUCULKGDGPUQGI INFEQKESNGPUKUWGS IKGLTEGLHGFHVFHEFGDN...IAQ', SingleLetter

```

Figure 31: Automatic BLASTX Search with the Humanizer: finding the human protein sequence

Sequences producing significant alignments:

Select: All None Selected: 0

Alignments Download GenPept Graphics

Description	Max score	Total score	Query cover	E value	Ident	Accession
<input type="checkbox"/> Chain A, Crystal Structure Of Human Cu-Zn Superoxide Dismutase Mutant G93a	186	186	98%	2e-60	61%	2ZKY_A
<input type="checkbox"/> superoxide dismutase [Cu-Zn] [Homo sapiens]	186	186	98%	3e-60	61%	NP_000445.1
<input type="checkbox"/> Chain A, Thermostable Mutant Of Human Superoxide Dismutase, C6a_C111s	186	186	98%	3e-60	61%	1N18_A
<input type="checkbox"/> Chain F, Structure Of Metal Loaded Pathogenic Sod1 Mutant G93a	185	185	97%	5e-60	61%	2WK0_F
<input type="checkbox"/> Chain A, Human Sod1 G93a Metal-Free Variant	184	184	97%	9e-60	61%	3GZP_A
<input type="checkbox"/> Chain A, Human Sod1 G93a Variant	184	184	97%	1e-59	61%	3GZQ_A

Figure 32: Manual BLASTX Search: finding the human protein sequence

Then the second BLAST search is performed. This comparison is limited to the gene within the organism homo sapiens as mentioned in Section 4.1. The automatic process had a total of fifty (50) results, despite only showing twenty (20). We compared the first couple of results from the automatic process, seen in Figure 31, to the results from the manual process in Figure 32. You can see that the results are the same by comparing the ID's from Figure 31 to the accession numbers in Figure 31. Since the results were the same for both processes, we moved on to the next step.

search to only homo sapiens. This will translate out humanized DNA sequence into a protein sequence. If the humanized form is accurate, this should return a protein sequence, within humans, that matches our humanized DNA sequence with 100% accuracy. Indeed, Figure 34, which displays our BLASTX results, shows that the first record has a 100% identity with our query sequence, the humanized form. In fact, it is the exact protein sequence we had selected previously after the second BLAST.

In addition to the organism 'Drosophila Melanogaster' and 'SOD', we ran tests with 'Danio Rerio' (zebrafish) and the gene 'OPTN', short for optineurin, and 'Mus Musculus' (rat) with gene 'TARDBP'. Similar results were obtained for both separate test cases.

Thus, we have concluded that these tests supports our conclusion that the functionality of the Humanizer is indeed accurate. From the tests we ran, we tested on three (3) different model organisms and three (3) different genes. The DNA sequences we chose ranged from hundreds to almost 3,000 base pairs, taking into consideration the humanization of longer genes. Since the test cases were all random and expected results were obtained, it must be safe to assume that the Humanizer will work for any sequence users would choose in the future.

5.2 Usability testing

While the immediate users of the Humanizer, Dr. Stilwell and the members of his lab, have tried this application and found it usable, before we publish this tool, we intend to test it more systematically on a broader set of users. Accordingly, I have completed the CITI training required by RIC's Institutional Review Board (IRB), and we plan to submit a proposal for a usability experiment on the Humanizer to the IRB before the end of the semester.

6 Conclusions and Future Work

Developed using Python and the BioPython library, the *Humanizer* is able to connect numerous sources together to humanize genes. The manual process normally requires the usage of many different sources like searches within NCBI's nucleotide and protein databases, BLAST, and Emboss Water just to humanize a gene. In addition, the manual process also requires the user to personally humanize the result at the end. The amount of wait time and high possibility of errors from the common manual process is greatly diminished with the development of this tool, making it a huge stepping stone in the research and experimentation of human diseases on animal models.

After completing this project, I realized I have come a long way to understand the work the *Humanizer* must perform. A great bulk of my time with this project required learning numerous biology concepts. Although I had a bit of understanding

from previous biology courses I had taken in the past, humanizing genes went into much greater depth. In addition to biology concepts, I spent a lot of time learning about the tools I needed the *Humanizer* to connect to. NCBI's Entrez search tool alone supports so many different features to query through all the databases supported by NCBI. Not only that, but the results retrieved from these sources needed to be understood too, in order for me to use them. The BLAST tool is another tool I had to learn about, as well as its algorithms which allows it to perform such a useful task. There were also many surprises along the way that forced me to learn even more about biology. Then, I had to figure out how to translate that into the code. An example of one of these surprises were the result of the addition of the 3-frame translation. When I got to this part in the application, the results were not matching up. No matter how I edited the code, it was still not coming up correctly. The reason was because of a small piece of biology that I had overlooked. Aside from the biology, I found that learning BioPython was a bit of a complication. Although the documentation that came with it was easy to follow, especially for a beginner like me, it did not go into as much depth as I had needed it too. One of the challenges were trying to find other sources that would help me learn what I needed my program to do. There were not many outside sources other than those found throughout its website and the API documentation. Thus, I found myself going directly into the source code of BioPython to see what was really happening behind the scenes. Overall, the development of the *Humanizer* entailed learning from day one to the very end of the semester. While the development of the *Humanizer* is reaching the end of its first run, I feel most proud of all the biology I learned and using that to create an application that will be of use to other people at our college, and hopefully even beyond that.

For future runs of the *Humanizer*, I would suggest other developers to first focus on understanding the biology that is being demonstrated by the program already, before building on it. Next, to build on this project, the developer must know the tools that are used. Personally humanizing a gene by hand would really help with the understanding of both of these, and show how the program works as well.

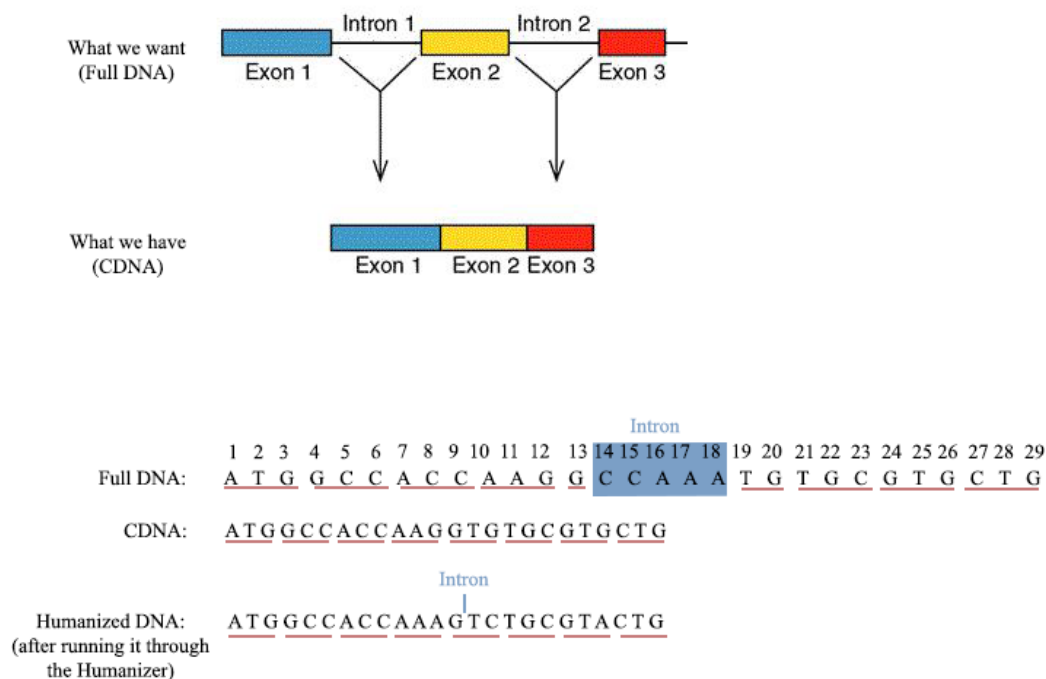


Figure 35: Flowchart of next steps

From a biological point of view, an important update to the program should include a step further after humanization. Humanization, which is explained in Section 4.1, returns a humanized DNA sequence that only includes the exons of the gene, the coding sequence (CDS). As stated in Chapter 1, there also exists introns that do not code for proteins, but also hold a very important value. Although they do not code for proteins, they may code for the functionality of the organism that without it may cause complications. Figure 35 displays this concept and shows that the next step of development should include the insertions of introns back into the genome.

Some other features a developer may want to pick up is adding a more user-friendly interface to the program: adding a menu to show the user what types of model organisms are in the `codon.bias.2d.list`. This may also extend to allow the user to enter in abbreviations of model organism names, or take away the option of typing in the organism name altogether to avoid any errors. Another feature can be to allow the program to show more than twenty (20) entries at a time after searching through the nucleotide database for DNA sequences, or after BLAST searches. With the help of developers and biologists, the *Humanizer* can reach places further than Rhode Island College. Thus, I am excited and honored to have been able to present the first development of the *Humanizer*, and I look forward to seeing where it leads.

References

- [Chang et al., 2017] Chang, J., Chapman, B., Friedberg, I., Hamelryck, T., de Hoon, M., Cock, P., Antao, T., Talevich, E., and Wilczynski, B. (2017). Biopython tutorial and cookbook. <http://biopython.org/DIST/docs/tutorial/Tutorial.html>.
- [Genetics, Education, Discovery (GeneEd), 2018] Genetics, Education, Discovery (GeneEd) (2018). Genetic code. https://geneed.nlm.nih.gov/topic_subtopic.php?tid=15&sid=19.
- [International Association of Developers, 2017] International Association of Developers (2017). Package bio. <http://biopython.org/DIST/docs/api/Bio-module.html>.
- [National Institute of General Medical Sciences (NIGMS), 2017] National Institute of General Medical Sciences (NIGMS) (2017). Using research organisms to study health and disease. https://www.nigms.nih.gov/Education/Pages/modelorg_factsheet.aspx.
- [National Institute of Neurological Disorders and Stroke (NINDS), 2013] National Institute of Neurological Disorders and Stroke (NINDS) (2013). Amyotrophic lateral sclerosis (als) fact sheet. <https://www.ninds.nih.gov/Disorders/Patient-Caregiver-Education/Fact-Sheets/Amyotrophic-Lateral-Sclerosis-ALS-Fact-Sheet>.
- [Pevsner, 2015] Pevsner, J. (2015). *Bioinformatics and Functional Genomics*. Wiley-Blackwell, United Kingdom.
- [Than, 2018] Than, K. (2018). What is darwin’s theory of evolution. <https://www.livescience.com/474-controversy-evolution-works.html>.
- [The Tech Museum of Innovation, 2018] The Tech Museum of Innovation (2018). Mutations and disease. <http://genetics.thetech.org/about-genetics/mutations-and-disease>.

A Program Code

```
#####  
## PROGRAMMER: Stacy Vang  
## DESCRIPTION:  
## This program will take a user-input biological organism  
## name and gene and run it against  
## the NCBI Nucleotide database to return a DNA sequence. The  
## resulting sequence is then BLASTed to  
## retrieve its Protein sequence, and then blasted again to  
## retrieve a Human Protein sequence. The program  
## then runs the Pairwise2 function from the BioPython  
## package to align the two protein sequences. A  
## 3-frame translation is run to find the correct reading-  
## frame for the test organism. All nucleotides  
## before the start codon (of the test organism nucleotide  
## sequence), and all nucleotides after the stop  
## codon is extracted. This results to the determination of  
## the open reading frame of the DNA sequence.  
## The remaining nucleotide bases in the DNA sequence will be  
## modified. Any disimilarities  
## in amino acids between the test organism and the human  
## protein sequence will result in a change of  
## amino acids at the nucleotide level.  
##  
## PURPOSE: The purpose of this program is to aid its users  
## in humanizing genes. The results of this  
## program will present the user with a humanized gene of the  
## user-inputted gene and organism.  
#####  
  
#####  
# IMPORTING NECESSARY LIBRARIES  
#####  
import sys      # Required to exit the program  
import os       # Required to get user's environment  
import Bio      # Required to run Biopython Modules  
  
# SEE: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf (  
# page 125/335, chap 9)  
# SEE: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf (  
# page 72/335, chap 6)
```



```

from Bio import Entrez, SeqIO, AlignIO
# SEE: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf (
    page 96/335, chap 7)
from Bio.Blast import NCBIWWW, NCBIXML
from Bio.Seq import translate
# SEE: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf (
    page 94/335, sec 6.4.6)
from Bio import pairwise2
from Bio.SubsMat.MatrixInfo import blosum62
# SEE: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf (
    page 30/335, sec 3.10)
from Bio.Data import CodonTable

```

```

#####
# Adding global email variable according to the 'NCBI's
# Entrez User Requirements'
#####
# REF: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf (
# page 126/335, sec 9.1)
Entrez.email = "gstilwell@ric.edu"

```

```

#####
# METHODS
#
# codon_bias_list                Creates a list of
#                               each organism (in the Codon_Bias_DB file)'s set of codon
#                               bias
# search_nucleotide_seq         Queries through the
#                               NCBI Nucleotide Database for the user-given test organism
#                               and gene
# nucleotide_esummary           Grabs detail
#                               description of a particular sequence record from the NCBI
#                               Database.
# fetch_seq                     Fetches nucleotide/
#                               protein sequence from the NCBI Nucleotide and Protein
#                               Databases given the index/id of a record
# blastx_nucleotide_seq         Blasts the nucleotide
#                               sequence of a user-given organism to get a Protein
#                               Sequence
# blastx_nucleotide_seq_list    Blasts the nucleotide

```

```

    sequence of a user-given organism to get Protein
    Sequences from Homo Sapiens
# find_DNA_from_reading_frame          Extracts the
    nucleotides before the start codon and from the stop codon
    beyond from the DNA
# get_amino_acid_index                 Retrieve the index of
    the amino acid abbr letter from the 'amino_acids_letter'
    list that matches the amino acid given
# export_results                       Export results into a
    text file
#####

def codon_bias_list(codon_bias_string):
    """
    RETURNS the list of each individual organism's set of
        codon bias.
    Will eventually be appended into a 2D list.
    """
    # 'codon' will temporarily store each codon(3 nucleotides
    )
    codon = ""
    # Declare the codon list variable
    c_list = []
    # Starting at index 1, because index 0 is the "["
    character that we do not need (see the '
    codon_bias_table.txt' file for format)
    # Ending at (len(codon_bias_string)-1), because the last
    character is "]" that we do not need (see the '
    codon_bias_table.txt' file for format)
    for i in range (1, (len(codon_bias_string)-1), 1):
        # If next element is a nucleotide, it is part of a
        codon.
        # Store it in the codon STRING.
        if (codon_bias_string[i].isalpha()):
            codon = codon + codon_bias_string[i]
            # Every codon contains 3 nucleotides.
            # If the codon string contains 3 nucleotides,
            append it to the codon LIST.
            if (len(codon) == 3):
                c_list.append(codon)
                # Refresh the codon variable for the next
                codon

```

```

        codon = ""
        # If next element is not a nucleotide, append an
        # empty string to the list
        # This will act as a place-holder for the empty codon
        bias
        elif ((codon_bias_string[i] == ",") & (
            codon_bias_string[i-2] == ",")):
            c_list.append(" ")
    return c_list

def search_nucleotide_seq(organism, gene):
    """
    RETURNS a list of records that match the query
    """
    handle = Entrez.esearch(db = "nucleotide", term =
        organism + "[Orgn] AND " + gene + "[Gene]")
    record = Entrez.read(handle)
    handle.close()
    return record

def nucleotide_esummary(record_id):
    """
    Prints the information of a particular record from the
    Nucleotide DB.

    This is needed to help end-users decide which Nucleotide
    record they want to use
    """
    handle = Entrez.esummary(db = "nucleotide", id =
        record_id)
    record = Entrez.read(handle)
    handle.close()
    # Printing out the index: adding 1 so as not to confuse
    # users
    print ("INDEX: " + str(i + 1));
    # Grabbing record[0] because there is only one record
    # Prints the ID for emphasis
    print ("ID: " + str(record[0]["Id"]));
    # Prints GI, Accession Version, among other
    # identification info

```

```

    print (record[0]["Extra"]);
    # Prints record Title
    print (record[0]["Title"]);
    # Printes record Length
    print ("Length: " + str(record[0]["Length"]));
    print ("\n");

def fetch_seq(database, index):
    """
    RETURNS the record as a SeqRecord; need this to translate
        the dna later
    """
    handle = Entrez.efetch(db = database, id = index, rettype
        = "fasta")
    results = SeqIO.read(handle, "fasta")
    handle.close()
    return results

def blastx_nucleotide_seq(fasta_string, organism_query):
    """
    RETURNS the Protein Sequence for the appointed organism.
    Results (50 entries MAX) are sorted in order by score.
    Will return the 1st result (the one with the best match).
    """
    handle = NCBIWWW.qblast("blastx", "nr", fasta_string,
        entrez_query = organism_query + " [Orgn]", format_type
        = "text")
    #returns the results as a SeqRecord
    result = next(SeqIO.parse(handle, "fasta"))
    handle.close()
    return result

# REFERENCE: CHAP 7 sec 7.1 (pg 96/335) and sec 7.3 (pg
    99/335) - http://biopython.org/DIST/docs/tutorial/Tutorial.pdf
def blastx_nucleotide_seq_list(fasta_string):
    """
    Records are held in a list to enable access to any record
        later in the program
    """

```

```

RETURNS records of Human Protein Sequences for the given
    organism. Results (50 entries MAX) are sorted in order
    by score.
Will return a list of the results.
"""
handle = NCBIWWW.qblast("blastx", "nr", fasta_string ,
    entrez_query = "Homo Sapiens [Orgn]", format_type = "
    text")
records = list(SeqIO.parse(handle, "fasta"))
handle.close()
return records

def find_DNA_from_reading_frame(reading_frame, query,
start_codon, stop_codon, protein_seq):
    """
    RETURNS the new dna sequence without the DNA before the
        start codon and from the stop codon beyond
    """
    # Loop through the length of the non-human organism dna,
        CONSIDER its reading_frame
    # Start at the reading frame index
    # End 3 positions before the end of the length, since we
        are looping by threes
    dna_seq = []          # will store the new dna sequence

    for i in range(reading_frame, len(query) - 3, 3):
        # Looping for start_codon
        # For each pair of codon, it will loop through the
            length of start_codon
        # BEWARE: We are only considering the start codon '
            ATG', which occurs 99.9% of the time

        #for j in range (0, 1, 1):
            codon = query[i] + query[i+1] + query[i+2]
            if (codon == 'ATG'):
                # Add codon to dna_seq - RESET IT
                dna_seq = [codon]

            # Now looking for stop codon!
            # Begin loop where it ended in the codon
            # Start 3 indexes after i, because

```

```

start_codon was the previous 3 indexes
for k in range(i + 3, len(query), 3):
    # Try-Except clause to catch IndexError
    that may occur
    try:
        codon = query[k] + query[k+1] + query
            [k+2]

        # Looping for stop codon
        # For each pair of codon after the
        start_codon, loop through length
        of stop_codon
        for l in range(0, len(stop_codon),
            1):
            # If match and len(dna_seq) =
            original protein length (w/o
            gaps)
            if ((codon == stop_codon[l]) & (
                len(dna_seq) == len(
                protein_seq))):
                # Appending STOP codon, but
                it will not be
                incorporated into the
                SEQUENCE
                dna_seq.append(codon)
                return dna_seq
        # If an IndexError occurs, ignore this
        and go back to find the correct Start
        Codon
    except IndexError:
        break;

    # Append codon to dna seq
    dna_seq.append(codon)

return -1

```

```

def get_amino_acid_index(amino_acid):
    """
    RETURNS the index of the protein letter, if matched. Else
    , returns -1 (as an error)
    """

```

```

This is needed in order to get into the 2D codon bias
list and humanize the non-human DNA seq
"""
for j in range(0, len(amino_acids_letter), 1):
    if (amino_acid == amino_acids_letter[j]):
        return j
return -1

```

```

def export_results(organism, gene, dna):
    """
    Creates/Rewrites a file with the results (given through
    param) and saves onto user's desktop
    """
    # Getting the user's home environment path
    users_env = os.getenv("USERPROFILE")
    path = users_env

    # Getting the name of the file the user wishes to save
    the file as
    new_file = input("Name the text file to save the results
    in: ")
    file_name = new_file + ".txt"

    # Combining path and file name
    file_path = path + "\\Desktop\\" + file_name

    # Creating/Opening the file to write in
    f = open(file_path, 'w+')
    f.write("\nTest Organism: " + organism)
    f.write("\nGene: " + gene)
    f.write("\nHumanized DNA SEQUENCE: " + dna)
    print("\nThe result has been saved in the '" + new_file
    + ".txt' file on your desktop.");
    f.close()

```

```

#####
# MAIN PROGRAM

```

```

#####

#####
# CREATE a 2D array of the Codon BIAS Table using the '
  Codon_Bias_DB.txt' FILE
#####
amino_acids_letter = ["A", "R", "N", "D", "C", "E", "Q", "G",
  "H", "I", "L", "K", "M", "F", "P", "S", "T", "W", "Y", "V
  "]

codon_bias_2D_list = [amino_acids_letter]
# This list will be used to determine which organism the user
  is using from the txt file
codon_bias_directory = ["N/A"]

# File will be properly closed upon completion when using the
  'with' keyword
with open("Codon_Bias_DB.txt") as f:
  # Read each line in the file
  for line in f:
    count = 0;
    # This is the start of a new organism's codon bias
    # If the line starts with a digit, it is NOT part of
      a comment or an empty line
    # (see the "Codon_Bias_DB.txt" file for reference)
    if (line[0].isdigit()):
      # Organism Number
      number = line
      # Organism Name
      line = next(f)
      name = line
      # Organism Codon Bias
      line = next(f)
      codon_bias = line

    # Call the subroutine to turn this into a list
    # List is appended to the 2D-Codon-list
    codon_bias_2D_list.append(codon_bias_list(
      codon_bias))
    # WARNING: "N/A" will be stored in index 0 of
      this list because we want to keep
    # the 2D list and the directory complementary

```



```

codon_bias_directory.append(name.strip('\n'))

print ("Welcome to the Humanizer!\n");
#####
# User-Input: User enters in the latin organism name and gene
they want to humanize
#####
test_organism = input("What is the latin name of your test
organism? ")

# Matching the user-inputed organism with the 'codon_bias_DB.
txt' file
# If this data is not found, the program cannot execute:
program exits
exists = False
for i in range (1, len(codon_bias_directory), 1):
    if (codon_bias_directory[i] == test_organism.lower()):
        test_organism_index = i # This var will later be
used to connect to the codon bias 2D-list
        exists = True
        break
if (exists == False):
    print ("\nSorry! The organism you entered "" +
test_organism + "" was NOT FOUND in the 'Codon-Bias-DB
.txt' file.");
    print ("Please check your spelling or input the data into
the file , and then try again!\n");
    exitPrgm = input("Goodbye!")
    sys.exit() # Exits the program

test_gene = input("What is the biological name of the gene
you are testing? ")

#####
# Searching for non-human nucleotide (DNA) sequence
#####
print ("\nSearching for the DNA sequence for gene "" +
test_gene + "" for test organism\n"" + test_organism +

```

```

    " '...'");
nucleotide_results = search_nucleotide_seq(test_organism ,
    test_gene)
total_entries = int(nucleotide_results["Count"])

if (total_entries == 0):
    print ("\nNo results have been found for your query
        sequence of '" + test_organism + "' and '" + test_gene
        + "'.");
    exitPrgm = input("\nGoodbye!")
    sys.exit() # Exits the program

print ("\nResults: " + nucleotide_results["Count"]);
print ("\nThe results are printed below along with an INDEX
    number of each result.\n");

# Looping through the results to provide a short summary of
    each record
# Capping the number of entries at 20
max_entries = 20

if(total_entries < max_entries):
    for i in range (0, total_entries , 1):
        result_id = nucleotide_results ["IdList"][i]
        nucleotide_esummary(result_id)
        num_entries = total_entries
else:
    print ("\nOnly " + str(max_entries) + " entries will be
        printed below.\n")
    for i in range (0, max_entries , 1):
        result_id = nucleotide_results ["IdList"][i]
        nucleotide_esummary(result_id)
        num_entries = max_entries

#####
# User-Input: User selects the non-human nucleotide(DNA)
    sequence record they want
#####
while True:
    try:

```

```

nucleotide_result_index = str(input("Type in the
INDEX of the DNA record you want: "))

if (0 < int(nucleotide_result_index) <= num_entries):
    break
print ("Sorry , invalid input. Try Again!")
except ValueError:
    print ("Sorry , invalid input. Try Again!")

print ("\n\nReturning non-human DNA sequence record for
result id: " + nucleotide_result_index)
# Fetching the nucleotide id:
# The user-inputted index needs to be casted as an int
# Subtracting 1 to the index to balance out the addition
from 'nucleotide_esummary()'
int_index = int(nucleotide_result_index) - 1
result_index = nucleotide_results ["IdList"][int_index]

# Fetching the nucleotide sequence
nucleotide_seq = fetch_seq("nucleotide", result_index)
print ("\nNON-HUMAN DNA SEQUENCE RECORD:");
print (nucleotide_seq);

#####
# Searching for non-human protein sequence
# BLASTing nucleotide(DNA) seq against test organism to get
the protein seq of that organism
#####
print
("\n\n
*****");
print ("***** BLASTING NON-HUMAN DNA SEQ TO RETRIEVE NON-
HUMAN PROTEIN SEQ *****");
print
("*****\
n");
nucleotide_id = nucleotide_seq.id
blast_result = blastx_nucleotide_seq(nucleotide_id ,
test_organism)

```

```

total_result = len(blast_result)

if (total_result == 0):
    print ("\nNo Non-Human Protein Sequence have been found
          for your query sequence, the non-human DNA sequence
          record selected previously.\n");
    print (nucleotide_seq);
    exitPrgm = input("\nGoodbye!")
    sys.exit() # Exits the program

#Return the first result bc normally that's the one with the
    greatest match
result_id = blast_result.id
protein_seq = fetch_seq("protein", result_id)
print ("NONHUMAN PROTEIN SEQUENCE RECORD:");
print (protein_seq);

#####
# Searching for human protein sequence
# BLASTing the nucleotide(DNA) seq for the complimentary
    human protein seq for the given gene
#####
print
("\n\n
    *****");
print ("***** BLASTING NON-HUMAN DNA SEQ TO RETRIEVE HUMAN
    PROTEIN SEQ *****");
print
("*****\n");
#human_organism = "Homo Sapiens" #Comparing seq with Homo
    Sapiens (Humans)
blast_results = blastx_nucleotide_seq_list(nucleotide_id)
total_results = len(blast_results)

if (total_results == 0):
    print ("\nNo Human Protein Sequence have been found for
          your query sequence, the non-human DNA sequence record
          selected previously.\n");
    print (nucleotide_seq);

```

```

exitPrgm = input("\nGoodbye!")
sys.exit() # Exits the program

print ("Results: " + str(total_results));
print ("\nThe results are printed below along with an INDEX
number of each result.\n");

# There should be 50 max entries , capping the entries at 20
# Looping through the results to provide a short summary of
each record
# Capping the number of entries at 20
max_blast_results = 20

print ("***** BLAST RESULTS *****");
if (len(blast_results) < max_blast_results):
    for i in range (0, total_results , 1):
        print ("INDEX: " + str(i + 1));
        # Running each result's id into the fetch_seq method
        to fetch the protein seq
        human_protein_seq = fetch_seq("protein",
            blast_results[i].id)
        print (human_protein_seq);
        print ("Length: " + str(len(blast_results[i])));
        print ("");
        tot_blast_results = blast_results

else:
    print ("\nOnly " + str(max_blast_results) + " entries
will be printed below.\n");
    for i in range (0, max_blast_results , 1):
        print ("INDEX: " + str(i + 1));
        # Running each result's id into the fetch_seq method
        to fetch the protein seq
        human_protein_seq = fetch_seq("protein",
            blast_results[i].id)
        print (human_protein_seq);
        print ("Length: " + str(len(blast_results[i])));
        print ("");
        tot_blast_results = max_blast_results

```

```

#####
# User-Input: User selects the human record they want to
# compare with
#####
while True:
    try:
        selection_result_index = str(input("Type in the INDEX
            of the record you want: "))

        if (0 < int(selection_result_index) <=
            tot_blast_results):
            break
        print ("Sorry, invalid input. Try Again!");
    except ValueError:
        print ("Sorry, invalid input. Try Again!")

print ("\nReturning human protein sequence for result id: " +
    selection_result_index);

# Fetching the protein id from the blast results:
# The user-inputted index needs to be casted as an int
# Subtracting 1 to the index to balance out the addition
# from earlier
int_index = int(selection_result_index) - 1
human_protein_result_id = blast_results[int_index].id

# Fetching the human protein sequence with the protein id
human_protein_seq = fetch_seq("protein",
    human_protein_result_id)

print ("\nHUMAN PROTEIN SEQUENCE RECORD:");
print (human_protein_seq);

# This result contains an unidentified amino acid ('X') in
# its protein sequence
# The application does not yet support unidentified amino
# acids
unidentified_amino_acids_count = human_protein_seq.seq.count
("X")
if (unidentified_amino_acids_count > 0):
    print ("\nSorry, the selected human protein sequence

```

```

        record contains an unidentified amino acid.");
print (" Unidentified amino acids are not yet supported by
        the Humanizer.");
print (" Please try again using a different human protein
        sequence selection.");
exitPrgm = input("\nGoodbye!")
sys.exit() # Exits the program

```

```

#####
# ALIGNING the non-human and human protein sequences using
  BioPython's Pairwise2
#####
print ("\n\n
        *****");
print ("***** ALIGNING '" + test_organism + "' AND 'Homo
        Spaiens' *****");
print
  ("*****\n
  ");
alignments = pairwise2.align.localds(protein_seq.seq,
  human_protein_seq.seq, blosum62, -10, -0.5)
print (pairwise2.format_alignment(*alignments[0]));

# The result (alignments) is a list containing seqA, seqB,
  score, begin index and end index
# The data listed above are all stored in the first index of
  alignments: alignments[0]
# Storing alignments[0] into a new variable to extract the
  individual items within this list
alignment_output = alignments[0]
non_human_organism = alignment_output[0] # non-human
  protein sequence (includes GAPS)
human_organism = alignment_output[1] # human
  protein sequence (includes GAPS)

#####
# Updates the non-human protein sequence (which does not
  include gaps) to get a length to get the open reading

```

```

    frame later
# STARTING NON-HUMAN PROTEIN SEQUENCES AT 'M', or 'ATG'
  BECAUSE THAT IS WHERE THE OPEN-READING FRAME OF THE DNA (
    new_dna_seq) WILL START
# Thus, we are ignoring the encodement of any amino acids
  before the start codon
#####
updated_protein_seq = ""
protein_sequence = protein_seq.seq
for i in range (0, len(protein_sequence), 1):
    if (protein_sequence[i] == "M"):
        for j in range (i, len(protein_sequence), 1):
            updated_protein_seq += protein_sequence[j]
        break

#####
# Retrieving the 3-frame translation of the non-human
  nucleotide(DNA) seq to get the
# correct READING-FRAME (DNA seq translates to its Protein
  seq)
#####
query = nucleotide_seq.seq      # non-human DNA seq
target = non_human_organism     # non-human PROTEIN seq
score = 0.0
translated_dna = ""
reading_frame = 0

# FIGURING OUT WHICH READING FRAME WITHIN THE DNA MATCHES THE
  PROTEIN SEQ
for frame_start in range(3):
    frame = translate(query[frame_start:])
    temp_score = pairwise2.align.localxx(frame, target,
        score_only = True)

    # This will get the translated dna with the best score
    if (temp_score > score):
        reading_frame = frame_start
        score = temp_score
        translated_dna = frame

```



```

#####
# Extracting DNA before the start codon, and after the stop
# codon from the non-human DNA sequence
# Determining the OPEN READING FRAME of the DNA sequence
# See: http://biopython.org/DIST/docs/api/Bio.Data.CodonTable
# -pysrc.html
#####
standard_table = CodonTable.unambiguous_dna_by_id[1]
start_codon = standard_table.start_codons
stop_codon = standard_table.stop_codons

# Find the open reading frame of the dna sequence using the
# reading frame
# new_dna_seq = the dna for the non-human protein sequence: '
# protein_seq'
new_dna_seq = find_DNA_from_reading_frame(reading_frame,
# query, start_codon, stop_codon, updated_protein_seq)

if (new_dna_seq == -1):
    print ("There has been an error finding the open reading
# frame of the nucleotide sequence.");
    print ("Please try again.");
    exitPrgm = input("Goodbye!")
    sys.exit() # Exits the program

#####
# HUMANIZING THE GENE
# Modifying the updated non-human DNA sequence ('new_dna_seq
# ') to get a DNA sequence more
# aligned with the human DNA sequence for the user-inputted
# gene
#####
modified_dna_seq = ""
start = False #Boolean to track the start codon
#This is required in case there are gaps
#BEFORE the start codon,
#thus, the new_dna_seq (nucleotide seq of
# the open reading frame)

```

```

                                #needs to be updated to add the gaps
                                before the start codon

# non_human_organism = protein sequence (seq includes the
  gaps)
# human_organism = protein sequence (seq includes the gaps)
if (len(non_human_organism) == len(human_organism)):
    for i in range(0, len(non_human_organism), 1):
        # Comparing each character within the sequences
        if (non_human_organism[i] != human_organism[i]):
            if (non_human_organism[i] == "-"):
                # get the protein from the human seq (col)
                amino_acid_index = get_amino_acid_index(
                    human_organism[i])
                # IF: There is an error
                if (amino_acid_index == -1):
                    print ("Sorry. There has been an error
                        matching an amino acid from the human
                        protein sequence to the proteins
                        listed in the amino acids list ('
                        amino_acids_letter ').");
                    print ("Error occurs at index " + str(i)
                        + " of the human protein sequence");
                    print ("Human amino acid: " +
                        human_organism[i]);
                    print ("Non-human amino acid: " +
                        non_human_organism[i]);
                    exitPrgm = input("PGoodbye!")
                    sys.exit() # Exits the program
                # ELSE: Add the human protein
                # get the codon from the organism (row =
                test_organism_index)
                add = codon_bias_2D_list[test_organism_index
                    ][amino_acid_index]
                modified_dna_seq += add
                # Start codon has been added
                if (add == "ATG"):
                    start = True

# inserting a gap into the new_dna_seq (non-
  human dna seq)
# because there's a gap in its protein

```

```

        sequence: this is required
    # to make sure the new_dna_seq is translated
    # as closely as possible
    # new_dna_seq.insert(12, '-')
    new_dna_seq.insert(i, '-')

elif (human_organism[i] == "-"):
# Delete the non-human organism protein:
# We do not want the model organism's gene
# integrated with the human organism's gene
# To Do This: Just don't add anything to the
# modified_dna_seq and don't do anything
# to new_dna_seq (which holds the nucleotide
# seq for the non-human orgn), else this
# will mess up the index of reading in the
# new_dna_seq.

# If human_organism[i] == "-" and "-" is
# located before the start codon in the
# non_human_organism protein seq
# THEN, ignore the amino acid (of the
# non_human_organism) at the same location
# as "-"
# BUT, don't ignore the dna sequence (
# new_dna_seq) at that position!
# So increase the length of new_dna_seq
# at the begining!
if ((new_dna_seq[i] == "ATG") & (start ==
False)):
    new_dna_seq.insert(i, '-')
    modified_dna_seq = modified_dna_seq

# non-human[i] and human[i] are not the same
# protein
else:
    # get the protein from the human seq (col)
    amino_acid_index = get_amino_acid_index(
        human_organism[i])
    # IF: There is an error in the human protein
    # sequence (containing an ambiguous amino
    # acid)
    if (amino_acid_index == -1):

```

```

        print (" Sorry. There has been an error
              matching an amino acid from the human
              protein sequence to the proteins
              listed in the database.");
        print (" Error occurs at index " + str(i)
              + " of the human protein sequence
              selected from the BLAST results.");
        print (" Human amino acid: " +
              human_organism[i]);
        print (" Non-human amino acid: " +
              non_human_organism[i]);
        print ("\nGoodbye!");
        sys.exit();
# ELSE: Replace the non-human protein with
      the human protein
# get the codon from the organism (row =
      test_organism_index)
add = codon_bias_2D_list[test_organism_index
                        ][amino_acid_index]
modified_dna_seq += add
# Start codon has been added
if (add == "ATG"):
    start = True

# Non-human and human protein are the same
else:
    # Keep the non-human organism protein
    add = new_dna_seq[i]
    modified_dna_seq += add
    # Start codon has been added
    if (add == "ATG"):
        start = True

# This should never happen since the pairwise alignment
      always makes sure the two protein sequences are of the
      same length
# But this is just a precaution.
else:
    print ("Length of test organism protein sequence and
          human protein sequence do not match!");
    print ("Cannot humanize the gene at this time. Please try
          again.");

```

```

exitPrgm = input(" Goodbye!")
sys.exit() # Exits the program

print
("\n\n
*****")
;
print ("***** FINAL RESULTS
*****");
print
("*****\
n");
# Print the humanized dna sequence
print ("Test Organism: " + test_organism);
print ("Gene: " + test_gene);
print ("Humanized DNA SEQUENCE: " + modified_dna_seq);

do_not_save = True;
while(do_not_save):
    save = input ("\nTo Save the result as a text file , enter
    'Y'.\nElse , enter 'N': ")
    if save.lower() == 'y':
        export_results(test_organism , test_gene ,
            modified_dna_seq);
        do_not_save = False;
    elif save.lower() == 'n':
        do_not_save = False;

exitPrgm = input ("Thank you for using 'The Humanizer'.
    Goodbye!")
# Exit program
sys.exit()

```