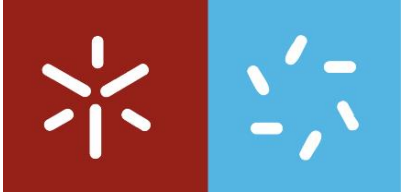


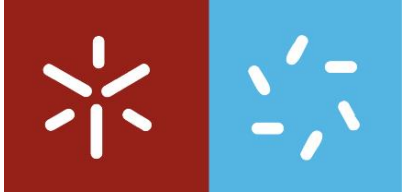
University of Minho
School of Sciences

Maria Adília Balacó Chócha Pessoa Monteiro

A user-friendly interface to GenBank using Biopython

Bachelor Degree Project in Applied Biology

June 2019



University of Minho
School of Sciences

Maria Adília Balacó Chócha Pessoa Monteiro

A user-friendly interface to GenBank using Biopython

Department of Biology, School of Sciences, University of Minho
Project executed under the supervision of
Assistant professor Björn Frederik Johansson

June 2019

Acknowledgments

I would like to offer my special thanks to my supervisor Björn Johansson, for his greatly appreciated feedback and patience.

I would also like to express my appreciation to the Course Management, for the big learning opportunity that was this project.

Finally, I would like to thank the online Stack Overflow & Biostars communities for their valuable help.

Index

Sumário

O pacote de software Biopython permite, por exemplo, interação com o GenBank, um repositório de informação de sequências biológicas, através das E-utilities, um conjunto de programas *server-side*. Infelizmente, o Biopython segue a interface do GenBank, que por necessidade é bastante complexa. Neste relatório foram criadas cinco funções para facilitar interação com o GenBank, usando o Biopython e as E-utilities, abrangendo pesquisas de proteínas para DNA no GenBank (find_gene & batch_find_gene), BLAST proteico para encontrar sequências semelhantes e homólogos (find_homologues), proteínas idênticas (find_ipg) e o cálculo do *codon adaptation index,* que pode ser usado para minimizar o *codon bias* em expressão heteróloga (choose_gene).

Abstract

The Biopython software package enables interaction with GenBank, a repository for biological sequence information, by means of the E-utilities, a set of server-side programs. Unfortunately, Biopython follows the GenBank interface, which is by necessity very complex. Five functions were developed in this work to ease interaction with GenBank using Biopython and the E-utilities, involving protein to DNA searches in GenBank (find_gene & batch_find_gene), protein BLAST for finding similar sequences and homologues (find_homologues), identical proteins (find_ipg) and calculating the codon adaptation index, which can be used to help minimize codon bias in heterologous expression (choose_gene).

1. Introduction

Manually analysing one or a couple of sequences in a web browser is feasible, but analysing thousands of sequences, without errors, risks becoming a formidable undertaking. Instead, biologists process biological data, in various types of file formats, using a programming language. The Python programming language is widely used, free, open source and popular in bioinformatics both for the simplicity of its syntax as well as its utility, having many library modules, dynamic typing and object-oriented capabilities (Oliphant, 2007).

1.1 GenBank

One of the main sources of biological data is GenBank, an extensive database containing publicly available nucleotide sequences. It is one of the online services provided by the National Center for Biotechnology Information (NCBI), part of the United States based National Library of Medicine. By means of NCBI's Entrez retrieval system, GenBank combines data from other databases, like protein sequences and structures. Entrez is simultaneously an indexing (or content classifying, to make it more easily accessible) and a retrieval system for NCBI's major databases (Ostell, 2014).

As a partner in the International Nucleotide Sequence Database Collaboration, GenBank collaborates with other databases, for instance the European Molecular Biology Laboratory Nucleotide Sequence Database, to assemble a thorough repository accessible worldwide. Practically all GenBank records are submissions, mostly done using programs such as BankIt, a web-based sequence submission tool (Benson *et al.,* 2008). Anyone is free to submit sequence data to GenBank.

The feature section of a GenBank file is loaded with metadata (information about, or that contains, other data), including coding regions (the coding sequence subsection, or CDS), protein translations and cross references to other databases. A GenBank file also includes the header and the sequence sections, with the first containing, for example, the sequence length, an unchanging unique identifier called an accession number (AN), followed by the version number that tracks any changes to the sequence, a description (dubbed definition) and the source organism (Benson *et al.,* 2008).

1.2 Biopython and the E-utilities

In association to the Entrez system, the Entrez Programming Utilities (E-utilities) are a set of server-side programs (dealing with the generation of content of a web page) that provide

a fixed URL (an online page address) syntax that takes a set of parameters to search for and retrieve the requested data from NCBI databases (Sayers, 2009).

Biopython provides an open source application programming interface with software libraries that include access to NCBI's online services and ExPASy, a bioinformatics resource portal. Biopython also includes parsers specific to several file formats, for instance FASTA and GenBank, as well as a standard sequence class, with an alphabet attribute that differentiates between types of sequences, and common operations like translation and transcription (Chang *et al*., 2010).

The Bio.Entrez module permits programmatic access to Entrez, applying the E-utilities using functions, which translate the parameters given into the fixed URL syntax used by the E-utilities. For example: ESearch for text searches and AN retrieval; EFetch, for retrieval of a full record from an Entrez database in a specified format, like FASTA; or ESummary, for retrieval of document summaries (Sayers, 2009).

Unfortunately, using the E-utilities (even through Bio.Entrez) can be complex because they follow the database's format by necessity. For example, take the *Saccharomyces cerevisiae* Homeobox transcription factor with the GenBank protein AN AAA34866. Finding corresponding genes may involve a handful of trial and error attempts until suitable results are found. A potential NCBI Nucleotide query could be "PHO2 protein[All fields] AND Saccharomyces cerevisiae[Organism] AND (biomol_genomic[PROP] AND ("0"[SLEN] : "20000"[SLEN]))". The first field is based on the protein description, the second restricts the data source to *S. cerevisiae*, the third specifies the type of results as genomic DNA/RNA and the last field sets a maximum sequence length of 20000 base pairs, to avoid encountering entire chromosomes. While this search only yields 2 results (accessed on 4/5/2019), another protein could return numerous results.

1.3 BLAST and homologues

Another online service of the NCBI is the Basic Local Alignment Search Tool – BLAST, a sequence similarity search tool which compares a query sequence with an online database. There are 4 main types of BLAST according to the combination of sequence types: Nucleotide (blastn), Protein (blastp), Translated (tblastx/blastx) and Genome, with the Protein's default database being "nr", standing for non-redundant (McGinnis & Madden, 2004).

A BLAST search returns the ANs of the sequences matched and statistical information, primarily the e-value (expect value, the false-positive rate, set to 10 by default) and percent

sequence identity, the similarity score between two sequences, in order of highest to lowest similarity. BLAST searches present the e-value instead of P values because they do not concern a single pairwise alignment, but many. In other words, the e-value is the expected number of times a BLAST score would happen by chance, after many searches. On the other hand, while BLAST minimizes false positives, false negatives may happen. Changing BLAST parameters is usually a trade-off between speed and sensitivity and the default settings usually allow for the best results in general (McGinnis & Madden, 2004; Pearson, 2013).

When two sequences are more similar than expected by chance, it is deduced that they are homologous, or sharing a common origin. Similar form, or sequence homology, is often a sign of similar function, thus searching online databases is very useful for determining the function of a newly sequenced gene. One way to detect functional similarity is focusing on orthologs, genes originating from the last common ancestor (a speciation event), as opposed to paralogs, genes produced by gene-duplication events, which may not retain the original function (Ashburner *et al.*, 2000; Koonin, 2005; Pearson, 2013).

1.4 Codon bias and heterologous expression

Heterologous protein expression resulting from genetic, protein, and metabolic engineering advances, such as recombinant DNA, has created an alternative to natural sources and lead to high level production (Mattanovich *et al*., 2012).

One of the many factors that affect heterologous protein expression is codon bias towards the protein's corresponding gene. Codon bias is the frequency of or preference for certain codons and varies by organism, proposed to be determined by natural selection and mutation related to translational efficiency. Most amino acids are coded by several codons and the available amino acid codons are used according to each organism's codon bias. The codon bias of messenger RNA is also mirrored in transfer RNA (tRNA) (Angov *et al*., 2008; Sahdev *et al.*, 2008; Sharp & Li, 1987; Terpe, 2006).

In particular, codon bias has a greater effect on the elongation step of prokaryote translation than differences in the Shine-Dalgarno sequence (a short sequence upstream of the start codon) in the initiation of translation or the stop codon in termination (Lithwick & Margalit, 2003).

Prokaryotes are often the chosen hosts for heterologous expression. Prokaryotes are usually inexpensive in their carbon source demands, accumulate biomass quickly, allow for high-cell density fermentation and are relatively easy to genetically manipulate. However, the

lack of post-translational machinery like glycosylation (without genetic engineering), formation of inclusion bodies and the accumulation of the lipopolysaccharide endotoxin, in the cases of gram-negative bacteria (predominantly *Escherichia coli*), may prove disadvantageous (Sahdev *et al.*, 2008; Terpe, 2006).

The overexpression of heterologous genes in *E. coli* with rare codons, which then requires rare tRNAs, may impede translation. This is especially the case if these codons are present in clusters in the N-terminus of the coding sequence, leading to reduced protein synthesis. If the coding sequence is rich in a specific amino acid, issues with codon bias in translation will also increase (Sahdev *et al.*, 2008; Terpe, 2006). Nonetheless, "codon harmonization", or substitution of the original codons with synonymous ones with comparable usage frequencies, appears to improve heterologous protein expression in *E. coli*. (Angov *et al*., 2008).

As eukaryotic host systems, yeasts are capable of protein processing typical of other eukaryotic organisms, without endotoxins. In particular, the baker's yeast *S. cerevisiae* is classified by the American Food and Drug Administration as an organism generally regarded as safe and has long been applied in the production of consumable goods. Its long history and the availability of the complete genome, culminating in the existence of the Saccharomyces Genome Database (Cherry *et al*., 2011), allowed for *S. cerevisiae* to become well established in large scale production and metabolic engineering. However, *S. cerevisiae* shows a strong fermentative metabolism, proteins produced are often hyper-glycosylated and retained in the periplasmatic space, consequently leading to partial degradation (Mattanovich *et al*., 2012; Ostergaard *et al*., 2000).

In *S. cerevisiae*, highly expressed genes seem to have the greatest degree of codon bias, accompanied by the preference for codons that are translated by dominant isoacceptor tRNAs (Percudani, 1997; Sharp & Li, 1987).

Codon bias is only one of many variables involved in heterologous expression. Even so, strain developers adopt the use of codon optimized genes to remain competitive and the cost of producing synthetic genes, or optimizing them, continually decreases. The quantity of products obtained by codon optimized genes has also been steadily increasing, including biofuels, such as ethanol and biodiesel, pharmaceuticals and industrial enzymes (Elena *et al*., 2014).

Methods that attempt to quantify codon bias are either null hypothesis based ("H_0" based methods), which measure the divergence from equal use of synonymous codons, or alternative hypothesis based ("H_1" based methods), which measure the frequency of supposed translationally optimal codons. Methods belonging to this second group are more likely to

detect codon bias due to translational selection and include the codon adaptation index (CAI) (Coghlan & Wolfe, 2000). The CAI can help predict levels of expression, by comparing gene candidates (for example, non-redundant genes that code identical proteins) to highly expressed genes and scoring them, allowing for codon optimization and therefore improving heterologous expression by minimizing codon bias (Lanza *et al*., 2014; Sharp & Li, 1987).

The objective of this project is to write Python functions to facilitate user interaction with GenBank, using Biopython. These functions involve protein to DNA GenBank searches (find_gene & batch_find_gene), protein BLAST (find_homologues), identical proteins (find_ipg) and calculating the codon adaptation index (choose_gene).

2. Materials and methods

The functions presented in this work depend on Biopython, the CAI module and an internet connection. They were implemented in Python.

2.1 Programmes used

Biopython version 1.73: https://biopython.org/
CAI module (Lee, 2018) used instead of Biopython's SeqUtils.CodonUsage: https://github.com/Benjamin-Lee/CodonAdaptationIndex
PlanetB for code formatting: http://www.planetb.ca/syntax-highlight-word
Python version 3.7: https://www.python.org
Pytest 4.6 module was used for basic testing: https://github.com/pytest-dev/pytest/
Spyder 3.3 IDE (Integrated Development Environment) used as part of the Anaconda distribution: https://www.anaconda.com/distribution/

Biopython's SeqUtils.CodonUsage module does not implement the codon adaptation index correctly, as stated in this post from Biostars, a bioinformatics website (last accessed on 24/5/2019): https://www.biostars.org/p/290485/

2.2 Codon adaptation index

The codon adaptation index (Sharp & Li, 1987) is calculated using a reference table of relative synonymous codon usage (RSCU) values from highly expressed genes. A codon's RSCU value is the observed frequency of that codon divided by the expected frequency of equal usage (based on the number of alternative codons).

The weight, or relative adaptiveness of a codon, is the frequency of use of a codon (its RSCU value) divided by the frequency of use of the optimal (most frequently used) codon for that amino acid (maximum RSCU value).

Finally, the index is the average mean of the relative adaptiveness of all the codons in the gene:

$$CAI = \frac{1}{n} * \sum_{i=1}^{n} \frac{RSCU_i}{\max RSCU_i}$$

n = the number of codons of a gene

The index should then fall between 0 and 1. As the CAI function was designed based of its original implementation, the original *S. cerevisae* (referred to as yeast) and *E. coli* RSCU values were used to ensure accuracy (Sharp & Li, 1987).

2.3 Entrez's Usage Guidelines and Requirements

NCBI requires that no more than one request be made every 3 seconds. Biopython's Bio.Entrez module automatically enforces that limit internally, with functions such as EFetch. The sleep function from the time module is used to pause between online queries using the functions (ESearch or EFetch) to respect NCBI's requirements.

An email was provided under the variable *Entrez.email*, but is not shown in this work. This email is used with each online request to Entrez, required since 2010 (Chang *et al*., 2010). Please provide one when calling any of the functions below, after importing Bio.Entrez:

```
1. from Bio import Entrez
2. from batch_find_gene import batch_find_gene
3. Entrez.email = "example@email.com"
4.
5. proteins = ["AAA34866", "GAX67478"]
6. print(batch_find_gene(proteins))
```

3. Results

This work presents a collection of functions that aims to simplify user interaction with GenBank using Biopython and the E-utilities, from searching for a corresponding protein's DNA accession number and a basic protein BLAST to retrieving identical proteins and calculating the codon adaptation index of genes. In this work, accession number will be interchangeably referred to as "AN" and codon adaptation index as "CAI".

3.1 Find_gene function

In its first version (source file "early_find_gene.py"), the "early_find_gene" function accepted a protein AN and an optional argument, the maximum number of results (set to 10 by default). The function returned a list with the protein description, the total number of results, a list with their corresponding ANs and, if found, the AN of the matching gene.

First, early_find_gene used EFetch, to retrieve the protein file of the AN passed as an argument, parsed it using Biopython's SeqIO module and saved it under the variable *record* (after a delay of 1 second). The protein description was then added to the results list. The variable *search_term* took in account the protein description and the source organism, along with the type of result and maximum length as mentioned in the Introduction. This is not ideal as the format of the fields is not guaranteed.

```
1.  from Bio import SeqIO, Entrez
2.  from Bio.Seq import translate
3.  from time import sleep
4.
5.  def early_find_gene(prot_acc,max_number=10):
6.
7.      results = []
8.      prot_fetch = Entrez.efetch(db="protein", id=prot_acc, rettype="gb",
9.                                 retmode="text")
10.     sleep(1)
11.     record = SeqIO.read(prot_fetch, "gb")
12.     results.append(f"Protein description: {record.description}")
13.
14.     start_index = record.description.index("[")
15.     query =  record.description[0:(start_index-1)] + "[All fields]"
16.     organism = record.description[(start_index+1):record.description.index("]")] + "
    [Organism]"
17.     search_term = query + " AND " + organism + ' AND (biomol_genomic[PROP] AND ("0"[
    SLEN] : "20000"[SLEN]))'
```

For this reason, a search for the corresponding gene directly in the GenBank file's coding sequence feature was later added.

```
18.     for feat in record.features:
19.         if feat.type == "CDS" and 'coded_by' in feat.qualifiers.keys():
20.             gene_acc = str(feat.qualifiers['coded_by']).split(":")
21.             gene_acc[0] = gene_acc[0].strip("[").strip("'")
22.             results.append(f"DNA AN: {gene_acc[0]}")
```

The function used *search_term* as the term argument for ESearch and the optional maximum number of results. If the resulting search yielded no results, early_find_gene exited (early_find_gene code line 33). If it did, it added the number of results found and their ANs to the results list.

Afterwards, early_find_gene used the saved gene ANs to retrieve the GenBank files in a single batch using EFetch (preceded by a delay of 1 second). It then checked if any of the coding sequences corresponded to the protein after translation. Since all DNA searches made restrict to genomic DNA/RNA, the mitochondrial codon table was unused (standard table 1,

early_find_gene code line 44). If a match was found, the gene's AN is added to the results list (this may be the same as the gene saved in the CDS metadata).

```
23.     dna_search = Entrez.esearch(db="nucleotide", term=search_term,
24.                                 idtype="acc", retmax=max_number)
25.     dna_search_record = Entrez.read(dna_search)
26.     sleep(1)
27.
28.     if int(dna_search_record['Count']) != 0:
29.         results.append(f"Result Count: {dna_search_record['Count']}")
30.         results.append(f"Accession numbers: {dna_search_record['IdList']}")
31.     else:
32.         results.append("No DNA results")
33.         return results
34.
35.     dna_acc = " ".join(dna_search_record["IdList"])
36.     dna_fetch = Entrez.efetch(db="nucleotide", id=dna_acc, rettype="gb",
37.                               retmode="text")
38.     dna_fetch_record = list(SeqIO.parse(dna_fetch, "gb"))
39.     print(len(dna_fetch_record[0].seq))
40.
41.     for feat in dna_fetch_record[0].features:
42.         if feat.type == "CDS":
43.             cds_location = dna_fetch_record[0].seq[int(feat.location.start):int(feat
    .location.end)]
44.             if translate(cds_location, table=1, to_stop=True) == record.seq:
45.                 results.append(f"Match found: {dna_fetch_record[0].id}")
46.     return results
```

Taking the example in the introduction with the *S. cerevisiae* Homeobox transcription factor AAA34866, the output of "early_find_gene" (last called on 1/6/19):

```
['Protein description: PHO2 protein [Saccharomyces cerevisiae]', 'DNA AN:
M22259.1', 'Result Count: 2', "Accession numbers: ['M22259.1',
'X05062.1']", 'Match found: M22259.1']
```

Apart from the unpredictability of the protein description field, early_find_gene may find no results if the associated DNA file is very large, which was the case with the protein NP_014996, a serine/threonine kinase from *S. cerevisiae*. The output of early_find_gene is as follows:

```
['Protein description: serine/threonine protein kinase MEK1 [Saccharomyces
cerevisiae S288C]', 'DNA AN: NM_001183771.3', 'No DNA results']
```

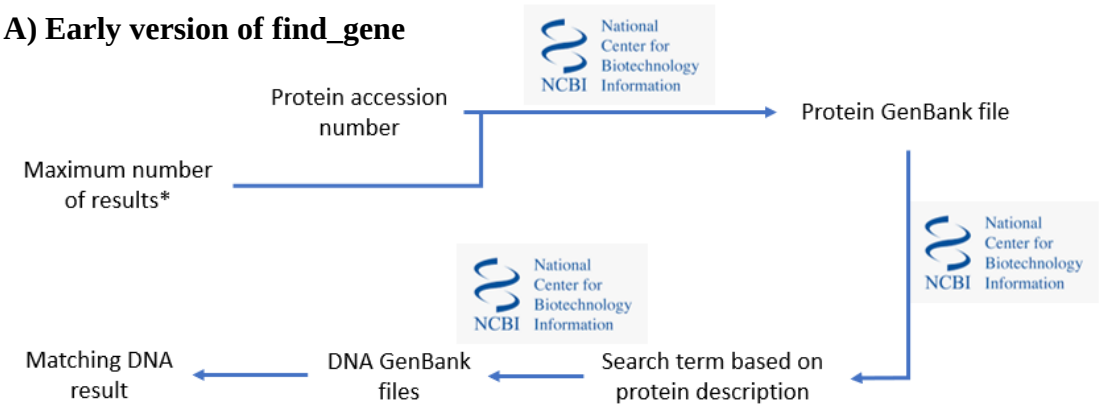
As this process was inefficient, the final version of find_gene depends only on the file metadata. This final version takes only one argument, a protein AN, and returns the corresponding DNA AN (source file "find_gene.py"). Find_gene comprises essentially of find_gene code lines 7-11, retrieving the protein file without the need for a pause with sleep (as it only makes one online request), and lines 18-22, for the metadata search that returns the protein's corresponding DNA accession number. The output of the final version of find_gene, called (twice) for the ANs above:

```
['DNA AN: M22259.1']
```

```
['DNA AN: NM_001183771.3']
```

In summary, the process of both versions of the find_gene function is as follows (Figure 1):

A) Early version of find_gene



B) Final version of find_gene

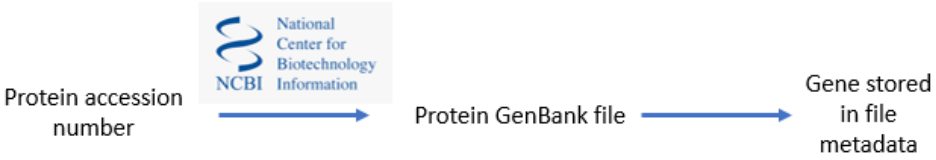


Figure 1 – Simplified process of the find_gene function. The early version (A) of the function accepted a protein AN as a required argument and an optional argument (*), the maximum number of results (set to 10 by default). It returned a list containing the protein description, the total number of results, a list with their corresponding ANs and, if found, the AN of the matching gene. Three separate online requests to Entrez were made in total. The final version (B) takes only a protein AN as an argument and returns the corresponding DNA AN saved in the protein file's metadata. Only one online request is made.

3.2 Find_homologues function

The find_homologues function accepts 3 arguments, the protein AN for the BLAST, an optional max number of results, set to 10 by default (to reduce testing time) and an optional name for the XML file (extensible markup language), set to "blast.xml" by default. The function creates an XML file, the default file format for saving BLAST search results, and returns a list of proteins ANs (source file "find_homologues.py").

First, find_homologues uses the Bio.Blast.NCBIWWW module's qblast function to do a protein BLAST in the non-redundant protein database ("nr"), using the default parameters (e-value = 10).

```
1.  from Bio.Blast import NCBIWWW, NCBIXML
2.  from time import sleep
3.
4.  def find_homologues(prot_acc, max_number=10, filename="blast.xml"):
5.      result_handle = NCBIWWW.qblast("blastp","nr",prot_acc,hitlist_size=max_number)
6.      with open(filename, "w") as out_handle:
7.          out_handle.write(result_handle.read())
```

The function then parses the resulting XML file after downloading it, using the Bio.Blast.NCBIXML module, retrieving the ANs of the proteins and appending them to the *protACC* list.

```
8.      with open(filename) as file:
9.          blast_record = NCBIXML.read(file)
10.         protACC = []
11.         for rec in blast_record.alignments:
12.             protACC.append(rec.accession)
13.     return protACC
```

Even when the max number of results is set to 10, the function is very slow due to BLAST. In early development, a 10 result BLAST of *S. cerevisiae* protein AAA34866, calculated using the time function from the time module, took over 20 seconds. An issue worth mentioning is that weeks later, after several uses, the same 10 result BLAST took over 2 minutes (143 seconds). The output is as follows (last called on 20/6/19):

```
['NP_010177', 'AJU66839', 'AJU64042', 'AJU65439', 'AJU57760', 'EDV08376',
'AJU73189', 'AJU59861', 'AJU63321', 'AHY74907']
```

In summary, the process of the find_homologues function is as follows (Figure 2):

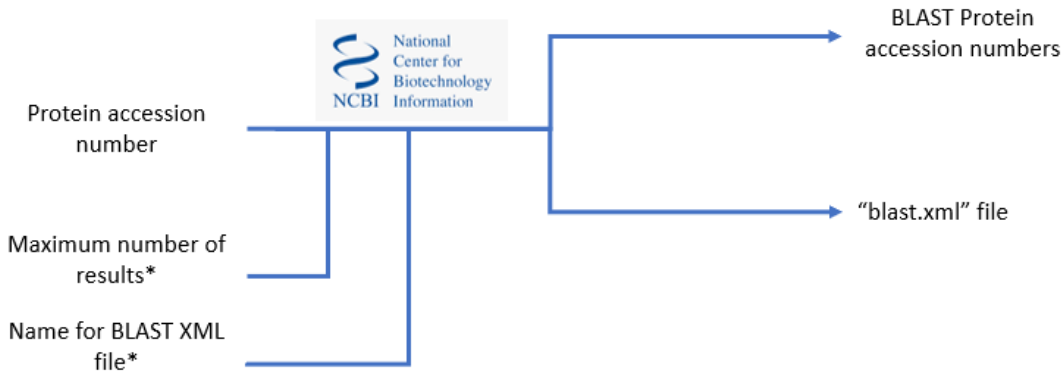


Figure 2 - Simplified process of the find_homologues function. The find_homologues function accepts 3 arguments, the required protein AN for the BLAST, an optional (*) max number of results (set to 10 by default) and an optional (*) name for the XML file, set to "blast.xml" by default. The function returns a list of the proteins ANs from the BLAST and saves the entire BLAST result in an XML file. Only one online request to Entrez is made.

3.3 Batch_find_gene function

As the original find_gene function was limited to only one protein AN at a time, the batch_find_gene function was created. The batch_find_gene function accepts only one argument, a list of protein ANs, and returns a list with the corresponding DNA ANs, namely the genes saved in the protein GenBank files. The function starts by joining the list into a single

string for a batch query using EFetch. It parses the result using the SeqIO module and saves it under the variable *prot_fetch_record*.

```
1.  from Bio import Entrez, SeqIO
2.
3.  def batch_find_gene(protACC_list):
4.
5.      protACC = " ".join(protACC_list)
6.      dna_acc = []
7.      prot_fetch = Entrez.efetch(db="protein", id=protACC, rettype="gb",
8.                                 retmode="text")
9.      prot_fetch_record = list(SeqIO.parse(prot_fetch, "gb"))
```

Finally, the batch_find_gene function iterates through all the proteins' CDS features in *prot_fetch_record*, processes the "coded by" metadata and appends the gene ANs to the *dna_acc* list. As the function does not check for size, whole chromosomes may be included in the list. In some cases, the protein is coded by the complement of the strand on file and is handled accordingly (batch_find_gene code lines 15-18).

```
10.     i = 0
11.     while i < len(prot_fetch_record):
12.         for feat in prot_fetch_record[i].features:
13.             if feat.type == "CDS" and 'coded_by' in feat.qualifiers.keys():
14.                 gene_acc = str(feat.qualifiers['coded_by']).split(":")
15.                 if "comp" not in gene_acc:
16.                     gene_acc[0] = gene_acc[0].strip("[").strip("'")
17.                 else:
18.                     gene_acc[0] = gene_acc[0].strip("[").strip("]")[11:]
19.                 if gene_acc not in dna_acc:
20.                     dna_acc.append(gene_acc[0])
21.         i += 1
22.     return dna_acc
```

The output of the batch_find_gene function called using a list with two *S. cerevisiae* homeobox transcription factors, AAA34866 and GAX67478:

```
['M22259.1', 'BEGW01000001.1']
```

A check for repeat DNA ANs (batch_find_gene code lines 19-20), as well as the AN processing first included in an early version of choose_gene was later included (source file "batch_find_gene.py").

In summary, the process of the batch_find_gene function is as follows (Figure 3):

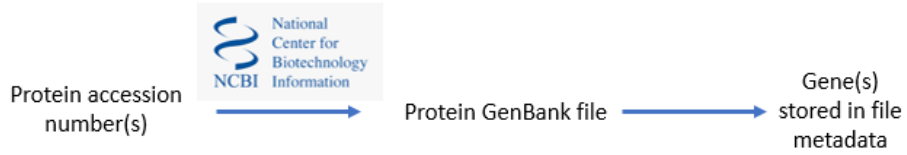


Figure 3 - Simplified process of the batch_find_gene function. The batch_find_gene function accepts only one argument, a list of protein ANs (which may only have one element) and returns a list with the corresponding DNA ANs (genes saved in the protein GenBank files). Only one online request to Entrez is made.

3.4 Find_ipg function

After executing a non-redundant BLAST to find similar proteins and homologues, finding identical proteins coded by different genes can help discern even more options for heterologous expression, as they may have different CAI values.

In early development, avoiding encumbering GenBank's servers by chaining several functions and their results together internally was the main concern. After retrieving the protein ANs as seen in find_homologues (find_homologues code lines 8-13), the proteins identical to those obtained from BLAST were found and their ANs retrieved using EFetch (*rettype* set to ipg, meaning identical protein groups). This identical protein EFetch was saved in a text file (referred to as "ipg.txt").

Finally, for testing purposes, batch_find_gene was used to find the corresponding DNA ANs. This last step clarified an issue with batch_find_gene, which returned redundant DNA ANs for the identical proteins (fixed in the final version, now including a check for repeat DNA).

The "ipg.txt" file includes, for example, the DNA ANs corresponding to the identical proteins, as well as the strand they are coded by, the start and the stop coordinates (Table 1):

Table 1 – Summary of the first two results obtained by an EFetch identical protein search (retrieval type set to ipg or identical protein groups), originally saved in a text file, based on the protein BLAST of the *S. cerevisiae* Homeobox transcription factor AAA34866 ("AAA34866blast.xml" annexed). The file contains, for example, the DNA (Nucleotide) ANs, start position, stop position and the protein's AN.

Nucleotide Accession	Start	Stop	Strand	Protein
NC_001136.10	270222	271901	-	NP_010177.1
NM_001180165.1	1	1680	+	NP_010177.1

As an identical protein EFetch returned the necessary information for choose_gene (DNA accession numbers, start positions, stop positions and strands) and bypassed the issues with batch_find_gene, another function named find_ipg was made specifically to process it.

Find_ipg accepts two arguments, a protein AN list and the name for the comma separated values (csv) file in which to save the EFetch results, set to "ipg.csv" by default. Find_ipg saves the results in a csv file rather than the text file mentioned previously, as it can be conveniently opened in Microsoft Office Excel (and other such programs). It otherwise follows the same format of "ipg.txt" and has the same information as seen in Table 1.

The function returns a dictionary with the keys being the corresponding DNA ANs and the values being tuples with the start position, stop position and coding strand (plus = 1, minus = 2), in that order. These values are already prepared for input in EFetch (*rettype* set to "gbwithparts"), allowing for specification of the start and stop positions as well as the strand.

First find_ipg uses EFetch (*rettype* set to ipg) to retrieve the identical proteins, then creates a csv file with the name supplied or defaults to "ipg.csv".

```
1.  import csv
2.  from Bio import Entrez
3.
4.  def find_ipg(protACC_list, ipgfilename="ipg.csv"):
5.
6.      protACC = " ".join(protACC_list)
7.      ipg = Entrez.efetch(db="protein", id=protACC, rettype='ipg',
8.                          retmode='text')
9.
10.     fetch_reader = csv.reader(ipg, delimiter="\t")
11.     with open(ipgfilename, "w", newline='') as out_handle:
12.         writer = csv.writer(out_handle)
13.         writer.writerows(fetch_reader)
```

Next the function opens the file it just created, iterates through it and saves the DNA ANs, start position, stop position and coding strand in their respective lists (*dna*, *start*, *stop* and *strand*) when a DNA AN is available (find_ipg code line 25). Finally, it returns these lists organized in a dictionary with the DNA ANs as keys and a tuple with the start position, stop position and coding strand as values.

```
14.     with open(ipgfilename) as ipgfile:
15.         reader = csv.reader(ipgfile)
16.         rows = list(reader)
17.         rows = rows[1:]
18.
19.     dna = []
20.     strand = []
21.     start = []
22.     stop = []
23.
24.     for col in rows:
25.         if len(col[2]) > 3:
26.             dna.append(col[2])
27.             start.append(col[3])
28.             stop.append(col[4])
29.             if col[5] == "+":
30.                 strand.append(1)
31.             else:
32.                 strand.append(2)
33.
34.     values = zip(start, stop, strand)
35.     return dict(zip(dna, values))
```

Part of the output ("{DNA: (start, stop, strand)}") of the find_ipg function, called using a list with the proteins from the find_homologues example ("AAA34866ipg.csv" annexed):

```
{'NC_001136.10': ('270222', '271901', 2), 'NM_001180165.1': ('1', '1680',
1), 'M22259.1': ('387', '2066', 1), 'X05062.1': ('391', '2070', 1),
'X95644.1': ('8447', '10126', 2)}
```

In summary, the process of the find_ipg function is as follows (Figure 4):

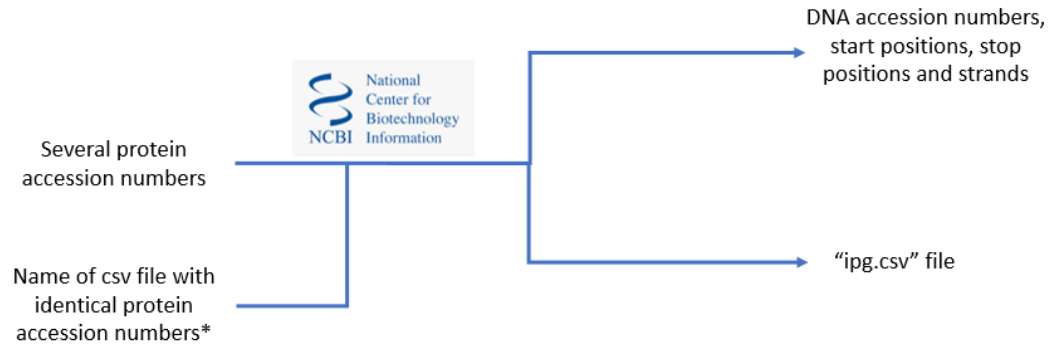


Figure 4 - Simplified process of the find_ipg function. The find_ipg function accepts two arguments, a required protein AN list and and optional (*) name for the csv file in which to save the EFetch results, set to "ipg.csv" by default. The function returns the DNA ANs, start positions, stop positions and strands in a dictionary ("{DNA: (start, stop, strand)}") and creates a file with the EFetch results. Only one online request to Entrez is made.

3.5 Choose_gene function

The choose_gene function is meant to assist heterologous protein expression using the codon adaptation index, to minimize codon bias. As the start and stop translation coordinates in the DNA are required for an accurate calculation of the CAI (using only the coding sequence) choose_gene uses the "ipg.csv" file created by find_ipg directly.

The proteins may also be coded by the complement of the strand on the GenBank file, affecting the calculation of the CAI: the RSCU *E. coli* values for the AAA and TTT codons differ, for example (cf. sharp_ecoli.txt annexed).

Choose_gene accepts two arguments, the name of the file containing RSCU values and the name of a csv file with the result of an identical protein EFetch (default is "ipg.csv"). The function returns a dictionary with the protein ANs as keys and tuples with the corresponding DNA ANs, CAI (rounded to 3 decimal places), start positions and stop positions as values. The format expected for the RSCU file is a text file with a header and two columns, the first column with codons and the second with RSCU values, delimited by a single space.

First, a dictionary with the codons and their corresponding RSCU yeast values is saved under the variable *RSCU_sharp*, to use the CAI function from the CAI module. The original *E. coli* RSCU values (which did not include stop codons) are used (Sharp & Li, 1987). These were

previously saved in a file named “sharp_ecoli.txt”, which is opened and parsed using the csv module.

```
1.  from CAI import CAI
2.  from Bio import Entrez, SeqIO
3.  from time import sleep
4.  import csv
5.
6.  def choose_gene(CAI_reference_table, ipgfile="ipg.csv"):
7.
8.      with open(CAI_reference_table) as file:
9.          reader = csv.reader(file, delimiter=" ")
10.         rows = list(reader)
11.         rows = rows[1:]
12.
13.         tri = []
14.         RSCU = []
15.         for col in rows:
16.             tri.append(col[0])
17.             RSCU.append(float(col[1]))
18.         RSCU_sharp = dict(zip(tri, RSCU))
```

The identical protein csv file is opened and processed, following the same method as seen in the find_ipg function. DNA ANs are saved in the *dnaACC* list as well as the start position, stop position and the strand, in the *start*, *stop* and *strand* lists, respectively.

```
19.     with open(ipgfile) as ipgfile:
20.         reader = csv.reader(ipgfile)
21.         ipgrows = list(reader)[1:5]
22.
23.     dnaACC = []
24.     protACC = []
25.     strand = []
26.     start = []
27.     stop = []
28.     for col in ipgrows:
29.         if len(col[2]) > 3:
30.             dnaACC.append(col[2])
31.             protACC.append(col[6])
32.             start.append(col[3])
33.             stop.append(col[4])
34.             if col[5] == "+":
35.                 strand.append(1)
36.             else:
37.                 strand.append(2)
```

A batch EFetch using *rettype* set to “gbwithparts” (specifying the start and stop positions as well as the strand) did not seem to retrieve the specified sequences correctly. As such, individual EFetch requests are made for each DNA accession number (each followed by a 1 second delay). This is impractical, as online requests to Entrez should be minimized.

```
38.     all_dna = []
39.     i = 0
40.     while i < len(dnaACC):
41.         dna_fetch = Entrez.efetch(db="nucleotide", id=dnaACC[i],
42.                                   rettype='gbwithparts', retmode='text',
43.                                   seq_start=start[i], seq_stop=stop[i],
44.                                   strand=strand[i])
45.         sleep(1)
```

```
46.         seq_obj = SeqIO.read(dna_fetch, "gb")
47.         i += 1
```

Finally, the function calculates the CAI for all the sequences and returns them in a dictionary with the protein ANs as keys and tuples with the DNA ANs, CAIs, start and stop positions as values. If a DNA sequence is not a multiple of 3, as required by the CAI function, choose_gene prints out the AN with a warning message (choose_gene code line 62).

```
48.     CAI_list = []
49.     final_dnaACC = []
50.     final_protACC = []
51.     final_start = []
52.     final_stop = []
53.     i = 0
54.     while i < len(all_dna):
55.         if len(all_dna[i]) % 3 == 0:
56.             CAI_list.append(round(CAI(all_dna[i], RSCUs=RSCU_sharp), 3))
57.             final_dnaACC.append(dnaACC[i])
58.             final_protACC.append(protACC[i])
59.             final_start.append(start[i])
60.             final_stop.append(stop[i])
61.         else:
62.             print(f"Not a multiple of 3: {dnaACC[i]}")
63.         i += 1
64.
65.     values = zip(final_dnaACC, CAI_list, final_start, final_stop)
66.     CAI_dict = dict(zip(final_protACC, values))
67.     return CAI_dict
```

The output of choose_gene ("{Protein: DNA, start, stop, strand}") using two *S. cerevisiae* proteins with available DNA ANs from "AAA34866ipg.csv" and the *E. coli* RSCU file named "sharp_ecoli.txt" (Sharp & Li, 1987):

```
{'NP_010177.1': ('NM_001180165.1', 0.173, '1', '1680'), 'AAA34866.1':
('M22259.1', 0.173, '387', '2066')}
```

Both genes, in the DNA GenBank files NM_001180165 and M22259, have the same CAI value (0,173) in *E. coli*.

In summary, the process of the choose_gene function is as follows (Figure 5):

Figure 5 - Simplified process of the choose_gene function. The choose_gene function accepts two arguments, the name of a csv file containing the result of an identical protein EFetch (default is "ipg.csv") and the name of the text file containing RSCU values. The function returns a dictionary with

the protein ANs, DNA ANs, codon adaptation index, start and stop positions ("{Protein: DNA, start, stop, strand}"). Several online requests are made, depending on the size of the identical protein file.

3.6 Chaining example

Despite the burden on NCBI's servers (requiring the use of the sleep function to pause between online requests), chaining find_homologues, find_ipg and choose_gene is a useful method for preliminary screening of potential gene candidates for heterologous expression.

Take the AN NP_036345.2, a recently submitted (8/6/2019) human malonyl CoA decarboxylase protein sequence. The malonyl-CoA (coenzyme A) decarboxylase catalyses the conversion of malonyl-CoA to acetyl-CoA. It is mostly located in mitochondria of animal tissues, being involved in fatty-acid metabolism and malonyl CoA decarboxylase deficiency in humans (Gao *et al*., 1999; Zhou *et al*., 2004).

First find_homologues is called (as no maximum number of results is provided, the function defaults to 10 result protein BLAST) with the XML file name "chain_blast.xml" and the returned protein AN list is saved in *protACC*, which is used as find_ipg's argument along with the name "chain_ipg.csv" for the identical protein csv file.

Finally, choose_gene is called using the "sharp_yeast.txt" yeast RSCU file and the "chain_ipg.csv" file. The functions were separated by 1 second delays using sleep.

```
1.  from time import sleep
2.  from find_homologues import find_homologues
3.  from find_ipg import find_ipg
4.  from choose_gene import choose_gene
5.
6.  protACC = find_homologues("NP_036345.2", filename="chain_blast.xml")
7.  sleep(1)
8.  find_ipg(protACC, ipgfilename="chain_ipg.csv")
9.  sleep(1)
10. print(choose_gene("sharp_yeast.txt", ipgfile="chain_ipg.csv"))
```

The output of the above code, excluding NW_011999143.1 from the "chain_ipg.csv" file:

```
Not a multiple of 3: NC_000016.10
Not a multiple of 3: NG_009079.1
Not a multiple of 3: CH471114.2
Not a multiple of 3: CM000267.1
Not a multiple of 3: NC_018440.2
Not a multiple of 3: NC_027884.1
Not a multiple of 3: NC_019817.1
Not a multiple of 3: NW_004087880.1
{'NP_036345.2': ('NM_012213.3', 0.04, '31', '1512'), 'AAH52592.1':
('BC052592.1', 0.04, '21', '1502'), 'AAD48994.1': ('AF090834.1', 0.04,
'1', '1482'), 'XP_523518.3': ('XM_523518.7', 0.04, '33', '1514'),
'JAA02619.1': ('GABC01008719.1', 0.04, '14', '1495'), 'XP_004058114.1':
('XM_004058066.2', 0.042, '32', '1513'), 'XP_003820926.1':
('XM_003820878.3', 0.041, '1', '1485'), 'XP_003272531.1':
('XM_003272483.3', 0.04, '16', '1497'), 'XP_007992389.1':
```

```
('XM_007994198.1', 0.04, '1', '1566'), 'XP_025226707.1':
('XM_025370922.1', 0.039, '35', '1516'), 'XP_003917292.1':
('XM_003917243.4', 0.04, '13', '1494')}, 'XP_011941035.1':
('XM_012085645.1', 0.04, '42', '1523')}
```

Of the 27 identical proteins retrieved by find_ipg, only one had no associated DNA AN (cf. "chain_ipg.csv" annexed). Secondly, when passed the AN NW_011999143.1, a *Cercocebus atys* whole genome shotgun sequence, the CAI function returned an error (KeyError: 'Bad weights dictionary passed: missing weight for codon').

It does not seem to be a matter of size, as EFetch (*rettype* set to "gbwithparts") specifies the coding sequence (16835 bp) and other genome shotgun sequences, coding genes of similar size, were accepted by the function. As the error pertains to CAI, the cause of it may be due to the sequence having letters other than "c", "t", "a" or "g". Unfortunately, the true cause of the error remains unknown.

Of the other 25 DNA ANs, 8 sequences were not multiples of 3. The remaining 17 DNA ANs all have very low CAI values (0,03 < CAI < 0,08), making it clear that trying to express these DNA sequences will probably not be very successful in *S. cerevisiae* without codon substitution.

If choose_gene is called using the same "chain_ipg.csv" file but with *E. coli* RSCU values, the CAI output (only multiples of 3 shown) values are higher (0,24 < CAI < 0,26):

```
{'NP_036345.2': ('NM_012213.3', 0.251, '31', '1512'), 'AAH52592.1':
('BC052592.1', 0.251, '21', '1502'), 'AAD48994.1': ('AF090834.1', 0.248,
'1', '1482'), 'XP_523518.3': ('XM_523518.7', 0.248, '33', '1514'),
'JAA02619.1': ('GABC01008719.1', 0.248, '14', '1495'), 'XP_004058114.1':
('XM_004058066.2', 0.252, '32', '1513'), 'XP_003820926.1':
('XM_003820878.3', 0.243, '1', '1485'), 'XP_003272531.1':
('XM_003272483.3', 0.243, '16', '1497'), 'XP_007992389.1':
('XM_007994198.1', 0.245, '1', '1566'), 'XP_025226707.1':
('XM_025370922.1', 0.25, '35', '1516'), 'XP_003917292.1':
('XM_003917243.4', 0.256, '13', '1494')} , 'XP_011941035.1':
('XM_012085645.1', 0.251, '42', '1523')}
```

Of course, other factors would have to be considered, such as post-translational machinery, for example.

3.7 Testing with Pytest

All functions were tested using Pytest and all passed (tests in source file "test_all.py"). The tests were run one by one, except for test_find_gene and test_batch_find_gene, as they concern the same objective (Figure 6). The tests were called through the Anaconda prompt window, by changing to the directory with "test_all.py" and inputting the "pytest test_all.py" command.

A. test_find_gene & test_batch_find_gene

```
(base) C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project>pytest test_all.py
============================================= test session starts =============================================
platform win32 -- Python 3.7.1, pytest-4.6.3, py-1.7.0, pluggy-0.12.0
rootdir: C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project
plugins: arraydiff-0.3, doctestplus-0.2.0, openfiles-0.3.1, remotedata-0.3.1
collected 2 items

test_all.py ..                                                                                       [100%]

========================================== 2 passed in 3.67 seconds ===========================================
```

B. test_find_ipg

```
(base) C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project>pytest test_all.py
============================================= test session starts =============================================
platform win32 -- Python 3.7.1, pytest-4.6.3, py-1.7.0, pluggy-0.12.0
rootdir: C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project
plugins: arraydiff-0.3, doctestplus-0.2.0, openfiles-0.3.1, remotedata-0.3.1
collected 1 item

test_all.py .                                                                                        [100%]

========================================== 1 passed in 7.26 seconds ===========================================
```

C. test_find_homologues

```
(base) C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project>pytest test_all.py
============================================= test session starts =============================================
platform win32 -- Python 3.7.1, pytest-4.6.3, py-1.7.0, pluggy-0.12.0
rootdir: C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project
plugins: arraydiff-0.3, doctestplus-0.2.0, openfiles-0.3.1, remotedata-0.3.1
collected 1 item

test_all.py .                                                                                        [100%]

========================================== 1 passed in 144.89 seconds =========================================
```

D. test_choose_gene

```
(base) C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project>pytest test_all.py
============================================= test session starts =============================================
platform win32 -- Python 3.7.1, pytest-4.6.3, py-1.7.0, pluggy-0.12.0
rootdir: C:\Users\liamo\Desktop\Python\PROJ\MM_A81190_Project
plugins: arraydiff-0.3, doctestplus-0.2.0, openfiles-0.3.1, remotedata-0.3.1
collected 1 item

test_all.py .                                                                                        [100%]

========================================== 1 passed in 5.78 seconds ===========================================
```

Figure 6 – Screenshots of the output of the Pytests (in "test_all.py") run for each of the functions in this work: (A) test_find_gene & test_batch_find_gene; (B) test_find_ipg; (C) test_find_homologues, with the time difference reflecting the function's slowness after multiple uses; (D) test_choose_gene. The test percentage refers to the number of tests passed, a progress indicator.

4. Discussion

A great deal of python packages related to the E_utilities, have already been created – a search in the journal Bioinformatics' online site, using only the keywords "e-utilities python" returns 463 results. In contrast, a search using the keyword "biopython" returns 105 results and a search using both keywords "e-utilities biopython", returns no results (last accessed on 29/5/19). One such package is "Entrezpy", which focuses on the query and download process of interacting with E-utilities and does not require Biopython. As stated by the authors, Biopython leaves the implementation of the interaction with E-utilities to the user (Buchmann & Holmes, 2019).

As mentioned in the introduction, the Bio.Entrez module applies the E-utilities using functions, translating the parameters given into the fixed URL syntax used by the E-utilities. Even so, its implementation may not be very intuitive.

A simple search for the corresponding DNA of a given protein initially required several lines of code because GenBank file sections seemingly have no specified format, rendering it inefficient. The early_find_gene function used three separate online requests to do so, whereas find_gene performs more cleanly and with less than half the code length of early_find_gene, as it only relies on the "coded by" metadata (DNA ANs saved in the protein GenBank files). However, its usage is made obsolete by batch_find_gene, which also operates when given a list with a single element.

As part of the aim of the find_homologues function is to be able to chain with find_ipg and choose_gene, it only returns the (usually 10) protein ANs and therefore is less informative than a regular BLAST search. While this is mitigated by the full result being saved in an XML file, the file can be difficult to process due to its abundance in information.

A potential future change could be including the option for more statistical information, or options for more BLAST parameters, such as specifying the e-value. Also, sequences available in GenBank with few homologues (or information in general) may affect the result of further processing, such as using find_ipg.

In the choose_gene function, direct filtering of the retrieved DNA GenBank files, while using a batch EFetch with *rettype* set to "gbwithparts" (specifying the seq_start, seq_stop and strand parameters) for a more economical approach would be ideal. Unfortunately, strings containing these parameters, as used to input ANs for batch requests, do not seem to be recognized.

Information regarding genomes has vastly improved from the date of publication of the codon adaptation index in 1987. A future change to the choose_gene function could be adapting it to use more thorough codon usage values (based on more genes), such as those from the Codon Usage Tabulated from GenBank (Nakamura *et al*., 2000).

Furthermore, the codon adaptation index is only meant to provide an estimation for the likelihood of a successful heterologous protein expression, as it only measures the codon usage within a gene. Still, the index may advise in the choice between choosing a gene in its present condition or trying to replace certain codons (Sharp & Li, 1987).

As the examples showed, especially the chaining example, these functions can be a useful tool, fulfilling their purpose in facilitating user interaction with GenBank and the use of its information by overlaying a different interface over Biopython's use of the E-utilities.

References

Angov, E., Hillier, C. J., Kincaid, R. L., & Lyon, J. A. 2008. Heterologous protein expression is enhanced by harmonizing the codon usage frequencies of the target gene with those of the expression host. *Public Library of Science one*, 3: e2189.

Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., ... & Harris, M. A. 2000. Gene ontology: tool for the unification of biology. *Nature genetics*, 25: 25.

Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., & Sayers, E. W. 2008. GenBank. *Nucleic acids research*, 37: D26-D31.

Buchmann, J. P., & Holmes, E. C. 2019. Entrezpy: A Python library to dynamically interact with the NCBI Entrez databases. *Bioinformatics*.

Chang, J., Chapman, B., Friedberg, I., Hamelryck, T., De Hoon, M., Cock, P., … & Talevich, E. 2010. Biopython Tutorial and Cookbook.

Cherry, J. M., Hong, E. L., Amundsen, C., Balakrishnan, R., Binkley, G., Chan, E. T., ... & Fisk, D. G. 2011. Saccharomyces Genome Database: the genomics resource of budding yeast. *Nucleic acids research*, 40: D700-D705.

Coghlan, A., & Wolfe, K. H. 2000. Relationship of codon bias to mRNA concentration and protein length in *Saccharomyces cerevisiae*. *Yeast*, 16: 1131-1145.

Elena, C., Ravasi, P., Castelli, M. E., Peirú, S., & Menzella, H. G. 2014. Expression of codon optimized genes in microbial systems: current industrial applications and perspectives. *Frontiers in Microbiology*, 5: 21.

Gao, J., Waber, L., Bennett, M. J., Gibson, K. M., & Cohen, J. C. 1999. Cloning and mutational analysis of human malonyl-coenzyme A decarboxylase. *Journal of lipid research*, 40: 178-182.

Koonin, E. V. 2005. Orthologs, paralogs, and evolutionary genomics. *Annual Review of Genetics*, 39: 309-338.

Lanza, A. M., Curran, K. A., Rey, L. G., & Alper, H. S. 2014. A condition-specific codon optimization approach for improved heterologous gene expression in *Saccharomyces cerevisia*e. *BMC systems biology*, 8: 33.

Lee, B. D. 2018. Python Implementation of Codon Adaptation Index. *Journal of Open Source Software* 3: 905.

Lithwick, G., & Margalit, H. 2003. Hierarchy of sequence-dependent features associated with prokaryotic translation. *Genome research* 13: 2665-2673.

Mattanovich, D., Branduardi, P., Dato, L., Gasser, B., Sauer, M., & Porro, D. 2012. Recombinant protein production in yeasts. *In* Recombinant gene expression. Humana Press, Totowa, NJ. pp 329-358.

McGinnis, S. & Madden, T. L. 2004. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic acids research* 32: W20-W25.

Nakamura, Yasukazu, Takashi Gojobori, and Toshimichi Ikemura. 2000. Codon usage tabulated from international DNA sequence databases: status for the year 2000. *Nucleic acids research* 28: 292-292.

Oliphant, T. E. 2007. Python for scientific computing. *Computing in Science & Engineering* 9: 10-20.

Ostell, J. 2014. The Entrez search and retrieval system. *In* The NCBI Handbook [Internet]. *2nd edition*. National Center for Biotechnology Information, US. pp 439-444.

Ostergaard, S., Olsson, L., & Nielsen, J. 2000. Metabolic engineering of *Saccharomyces cerevisiae*. *Microbiology and Molecular Biology Reviews*, 64: 34-50.

Percudani, R., Pavesi, A., & Ottonello, S. 1997. Transfer RNA gene redundancy and translational selection in *Saccharomyces cerevisiae*. *Journal of molecular biology*, 268: 322-330.

Pearson, W. R. 2013. An introduction to sequence similarity ("homology") searching. *Current protocols in bioinformatics*, 42*:* 3-1.

Sahdev, S., Khattar, S. K., & Saini, K. S. 2008. Production of active eukaryotic proteins through bacterial expression systems: a review of the existing biotechnology strategies. *Molecular and cellular biochemistry*, 307: 249-264.

Sayers, E. 2009. The E-utilities in-depth: parameters, syntax and more. *Entrez Programming Utilities Help [Internet]*.

Sharp, P. M., & Li, W. H. 1987. The codon adaptation index-a measure of directional synonymous codon usage bias, and its potential applications. *Nucleic acids research*, 15: 1281-1295.

Terpe, K. 2006. Overview of bacterial expression systems for heterologous protein production: from molecular and biochemical fundamentals to commercial systems. *Applied microbiology and biotechnology*, 72: 211.

Zhou, D., Yuen, P., Chu, D., Thon, V., McConnell, S., Brown, S., ... & Nadzan, A. M. 2004. Expression, purification, and characterization of human malonyl-CoA decarboxylase. *Protein expression and purification*, 34: 261-269.

Annexes

Python files

- batch_find_gene.py
- choose_gene.py
- early_find_gene.py (not annexed directly in this document)
- find_gene.py
- find_homologues.py
- find_ipg.py
- test_all.py

Other files (not annexed directly in this document):

- AAA34866blast.xml
- AAA34866ipg.csv
- chain_blast.xml
- chain_ipg.csv
- sharp_yeast.txt
- sharp_ecoli.txt

```python
# -*- coding: utf-8 -*-
"""
Created on Mon May 13 18:07:11 2019

@author: Maria Pessoa Monteiro
"""
from Bio import Entrez, SeqIO

def batch_find_gene(protACC_list):
    """
    Batch_find_gene takes a protein accession number list as the only
    argument and returns the corresponding DNA accession number in a list.
    """
    protACC = " ".join(protACC_list)
    dna_acc = []
    prot_fetch = Entrez.efetch(db="protein", id=protACC, rettype="gb",
                               retmode="text")
    prot_fetch_record = list(SeqIO.parse(prot_fetch, "gb"))

    i = 0
    while i < len(prot_fetch_record):
        for feat in prot_fetch_record[i].features:
            if feat.type == "CDS" and 'coded_by' in feat.qualifiers.keys():
                gene_acc = str(feat.qualifiers['coded_by']).split(":")
                if "comp" not in gene_acc:
                    gene_acc[0] = gene_acc[0].strip("[").strip("'")
                else:
                    gene_acc[0] = gene_acc[0].strip("[").strip("]")[11:]
                if gene_acc not in dna_acc:
                    dna_acc.append(gene_acc[0])
        i += 1
    return dna_acc
```

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 24 11:01:40 2019

@author: Maria Pessoa Monteiro
"""
from CAI import CAI
from Bio import Entrez, SeqIO
from time import sleep
import csv

def choose_gene(CAI_reference_table, ipgfile="ipg.csv"):
    """
    Choose_gene accepts two arguments, the name of the file containing RSCU
    values and the name of a csv file containing the result of an identical
    protein EFetch.
    The function returns a dictionary with the protein ANs as
    keys and a tuple containing the corresponding DNA ANs, CAI, start
    positions and stop positions as values.
    The format expected for the RSCU file is a text file with a header and
    two columns, the first column with codons and the second with RSCU values,
    delimited by a single space.
    """
    with open(CAI_reference_table) as file:
        reader = csv.reader(file, delimiter=" ")
        rows = list(reader)
        rows = rows[1:]

        tri = []
        RSCU = []
        for col in rows:
            tri.append(col[0])
            RSCU.append(float(col[1]))
        RSCU_sharp = dict(zip(tri, RSCU))

    with open(ipgfile) as ipgfile:
        reader = csv.reader(ipgfile)
        ipgrows = list(reader)[1:]

    dnaACC = []
    protACC = []
    strand = []
    start = []
    stop = []
    for col in ipgrows:
        if len(col[2]) > 3:
            dnaACC.append(col[2])
            protACC.append(col[6])
            start.append(col[3])
            stop.append(col[4])
            if col[5] == "+":
```

```
            strand.append(1)
        else:
            strand.append(2)
all_dna = []
i = 0
while i < len(dnaACC):
    dna_fetch = Entrez.efetch(db="nucleotide", id=dnaACC[i],
                              rettype='gbwithparts', retmode='text',
                              seq_start=start[i], seq_stop=stop[i],
                              strand=strand[i])
    sleep(1)
    seq_obj = SeqIO.read(dna_fetch, "gb")
    all_dna.append(str(seq_obj.seq))
    i += 1

CAI_list = []
final_dnaACC = []
final_protACC = []
final_start = []
final_stop = []
i = 0
while i < len(all_dna):
    if len(all_dna[i]) % 3 == 0:
        CAI_list.append(round(CAI(all_dna[i], RSCUs=RSCU_sharp), 3))
        final_dnaACC.append(dnaACC[i])
        final_protACC.append(protACC[i])
        final_start.append(start[i])
        final_stop.append(stop[i])
    else:
        print(f"Not a multiple of 3: {dnaACC[i]}")
    i += 1
values = zip(final_dnaACC, CAI_list, final_start, final_stop)
CAI_dict = dict(zip(final_protACC, values))
return CAI_dict
```

```python
# -*- coding: utf-8 -*-
"""
Created on Wed May  8 17:19:43 2019

@author: Maria Pessoa Monteiro
"""
from Bio import SeqIO, Entrez

def find_gene(prot_acc):
    """
    Find_gene accepts a protein AN as an argument. The function
    returns the AN of the matching gene.
    """
    results = []
    prot_fetch = Entrez.efetch(db="protein", id=prot_acc, rettype="gb",
                               retmode="text")
    record = SeqIO.read(prot_fetch, "gb")

    for feat in record.features:
        if feat.type == "CDS" and 'coded_by' in feat.qualifiers.keys():
            gene_acc = str(feat.qualifiers['coded_by']).split(":")
            gene_acc[0] = gene_acc[0].strip("[").strip("'")
            results.append(f"DNA AN: {gene_acc[0]}")

    return results
```

```python
# -*- coding: utf-8 -*-
"""
Created on Mon May 13 17:08:57 2019

@author: Maria Pessoa Monteiro
"""
from Bio.Blast import NCBIWWW, NCBIXML

def find_homologues(protACC, max_number=10, filename="blast.xml"):
    """
    Find_homologues takes a protein accession number as required argument,
    and an optional max_number of results argument, default set to 10,
    and does a protein BLAST. The function returns the accession numbers
    of the BLAST proteins.
    """
    result_handle = NCBIWWW.qblast("blastp","nr", protACC,
                                   hitlist_size=max_number)
    with open(filename, "w") as out_handle:
        out_handle.write(result_handle.read())
    with open(filename) as file:
        blast_record = NCBIXML.read(file)
        protACC = []
        for rec in blast_record.alignments:
            protACC.append(rec.accession)
    return protACC
```

```python
# -*- coding: utf-8 -*-
"""
Created on Fri May 24 11:01:40 2019

@author: Maria Pessoa Monteiro
"""
import csv
from Bio import Entrez

def find_ipg(protACC_list, ipgfilename="ipg.csv"):
    """
    Find_ipg accepts a protein accession number list as an argument and uses
    EFetch to retrieve identical proteins. A csv file, by default named
    ipg.csv, is saved with the results.
    The function returns a dictionary with the keys being the corresponding
    DNA ANs and the values a tuple with the start position, stop position
    and coding strand (plus = 2, minus = 1), in that order.
    """
    protACC = " ".join(protACC_list)
    ipg = Entrez.efetch(db="protein", id=protACC, rettype='ipg',
                        retmode='text')

    fetch_reader = csv.reader(ipg, delimiter="\t")
    with open(ipgfilename, "w", newline='') as out_handle:
        writer = csv.writer(out_handle)
        writer.writerows(fetch_reader)

    with open(ipgfilename) as ipgfile:
        reader = csv.reader(ipgfile)
        rows = list(reader)
        rows = rows[1:]

    dna = []
    strand = []
    start = []
    stop = []

    for col in rows:
        if len(col[2]) > 3:
            dna.append(col[2])
            start.append(col[3])
            stop.append(col[4])
            if col[5] == "+":
                strand.append(1)
            else:
                strand.append(2)
    values = zip(start, stop, strand)

    return dict(zip(dna, values))
```

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jun  1 14:19:15 2019

@author: Maria Pessoa Monteiro
"""
from find_gene import find_gene
from batch_find_gene import batch_find_gene
from find_homologues import find_homologues
from find_ipg import find_ipg
from choose_gene import choose_gene

def test_find_gene():
    """
    test_early_find_gene should return the matching DNA accession
    number (M22259.1) of protein AAA34866.
    """
    expect_gene = find_gene("AAA34866")
    assert expect_gene == ["DNA AN: M22259.1"]

def test_batch_find_gene():
    """
    test_batch_find_gene should return both gene's corresponding DNA accession
    numbers.
    """
    expect_batch = batch_find_gene(["AAA34866", "GAX67478"])
    assert expect_batch == ['M22259.1', 'BEGW01000001.1']

def test_find_ipg():
    """
    test_find_ipg should return the identical proteins of both proteins
    provided. GenBank may update, possibly leading to different identical
    protein results.
    """
    expect_ipg = find_ipg(["AAA34866", "GAX67478"], ipgfilename="testipg.csv")
    assert expect_ipg == {'M22259.1': ('387', '2066', 1),
                          'NC_001136.10': ('270222', '271901', 2),
                          'NM_001180165.1': ('1', '1680', 1),
                          'X05062.1': ('391', '2070', 1),
                          'X95644.1': ('8447', '10126', 2),
                          'Z74154.1': ('246', '1925', 2),
                          'BK006938.2': ('270222', '271901', 2),
                          'CM004297.1': ('288824', '290503', 2),
                          'LBMA01000004.1': ('288824', '290503', 2),
                          'HB870545.1': ('387', '2066', 1),
                          'HC927954.1': ('387', '2066', 1),
                          'BEGW01000001.1': ('1223469', '1225145', 1),
                          'CP004724.2': ('264356', '266032', 2),
                          'CP004740.2': ('253800', '255476', 2),
                          'CP004750.2': ('264165', '265841', 2),
                          'DG000040.1': ('258839', '260515', 2)}
```

```
def test_find_homologues():
    """
    test_find_homologues should return only the 3 results, as specified.
    GenBank may update, possibly leading to different BLAST results.
    """
    expect_blast1 = find_homologues("AAA34866",max_number=3,
                                    filename="testblast.xml")
    assert expect_blast1 == ['NP_010177', 'AJU66839', 'AJU64042']

def test_choose_gene():
    """
    test_choose_gene should return the DNA accession numbers, codon adaptation
    indexes, start positions and stop positions of the corresponding first two
    proteins in testipg.csv with available DNA accession numbers.
    """
    expect_CAI = choose_gene("sharp_yeast.txt",ipgfile="testipg.csv")
    assert expect_CAI == {'AAA34866.1': ('M22259.1', 0.148, '387', '2066'),
                          'NP_010177.1': ('NM_001180165.1', 0.148, '1', '1680')}
```