Stockholms universitet

Making GenFamClust user friendly in Python

Göra GenFamClust användarvänligt i Python

Jesper Holm

Supervisor: Lars Arvestad
Examinator: Marc Hellmuth
Date: May 26, 2022

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

Genes evolve by duplication and mutation of already existing genes. Genes that share a common ancestor are therefore often very similar to each other in both structure and function. This is one reason why most of the homology inference algorithms mainly use sequence similarity. Unfortunately, determining homology is not that simple; divergent families, convergent evolution, and multidomain proteins are all reasons why sequence similarity does not tell the whole story. A method called GenFamClust, consisting of several modules, has been shown to perform better than other homology inferring methods. However the GenFamClust program is not very user friendly. The aim of this project is to make a new GenFamClust program in Python that solves the usability issues from the predecessor. The resulting Python program consists of the same module structure but reduces the amount of commands required to run the program.

Sammanfattning

Gener utvecklas genom duplicering och mutation av en existerande gen. Detta betyder att gener med en gemensam förfader ofta liknar varandra i både genstrukturen och funktionaliteten. Detta är en anledning till att de flesta algoritmer som bestämmer huruvida två gener är homologa använder sig främst av geners sekvenslikhet. Olyckligtvis är det inte så enkelt att bestämma om två gener är homologa; divergerande genfamiljer, konvergent evolution och multidomän-proteiner är anledningar till detta. En metod som kallas GenFamClust, som är uppbyggd av olika moduler, har visat bra resultat jämfört andra metoder för att bestämma homologi av gener. Men GenFamClust-programmet är uppbyggt på ett sätt som kräver att användaren manuellt startar delberäkningar. Målet för detta projekt är att utveckla ett nytt program i Python som löser sin företrädares problem. Resultatet blev ett program med liknande modulstruktur men som kräver färre kommandon för att köra programmet.

1 Introduction

In this project we use the terms *homology* and *homologs* often. In our context, genes/proteins that are homologous to each other, simply means they share a common ancestor. So by saying "determine homology between genes/proteins" as an example, we mean to determine if those genes/proteins are homologous to each other, ie sharing a common ancestor. Homologs are interesting to determine, since there are many uses for them, a few examples are: inferring phylogeny, predict protein structure and function prediction [2]. Another important term in this project is *synteny* and this is essentially the conservation of order for genes on a chromosome. In this project we also use the term protein domain, which are what gives proteins their physical structure, but also more importantly, gives the proteins their function(s) [5]. Protein domains are important to us since they carry out protein functions which we either want to identify or infer homology on. We also use the term *sequence similarity* and this refers to sequences of genes/proteins can be aligned to some extent and have similar sequences from that alignment. One way of calculating the similarity score is by doing sequence alignments to find regions of similarity and scoring these and this is what the method BLAST does. BLAST or *Basic Local Alignment Search Tool* is an algorithm that finds sequence similarities between genes/proteins against a database [10]. On NCBI's website [10], BLAST calculates several important measurements such as, E-value, score and percent identity. However, in this project we are only interested in the score, which is the highest alignment score, bit score, between a query sequence and a database match. The bit score is a measure of sequence similarity that is $\log_2$ scaled and normalized. It also is independent of the database size which makes it suitable for comparing matches between searches. A similarity measure gives us a good idea whether two

genes/proteins are homologs, however due to problems such as divergent gene families, convergent evolution and multidomain proteins [2] makes determining homology limited purely based on sequence similarity. A sequence similarity network is a weighted network consisting of vertices that represent gene/protein sequences and two sequences that share a significant similarity, shares an edge weighted proportionate to their similarity. The neighborhood of a sequence in the network is a set of other sequences who shares an edge over a certain threshold. Multidomain proteins are simply proteins consisting of multiple domains. Unfortunately it is common for multidomain proteins to share domains [2], which can lead to strong local similarity for two otherwise unrelated proteins. This is why GFC not only uses sequence similarity but also synteny to help determine homologs.

The GenFamClust method builds upon *Neighborhood Correlation* [11], or NC for short, and the purpose of NC is to identify homologs in complex multidomain families with varying domain architectures. The reason NC is used as its similarity score is because of its great accuracy for inferring homology on multidomain proteins compared to other methods such as BLAST [11]. NC achieves this by creating a sequence similarity network and determine homologs by looking at local topology in the network [11]. NC requires an input file containing all-versus-all BLAST scores for pairwise genes/proteins to run. BLAST can be either run through the website or downloaded as a program on your computer. To run BLAST requires two things, firstly it requires some query data, a sequence or sequences in FASTA format [10]. FASTA is a text format that represents gene/protein sequences. Secondly, BLAST needs to know what reference data you want to use, this takes form of a database. All-Versus-All BLAST uses the query data as a reference database.

GenFamClust, or GFC, is a program suggested by Ali et al. [2] whose purpose is to determine homology between genes or proteins. GFC is based on another homology inference method called *Neighborhood Correlation*, and the GFC program is structured into different modules; Raw Similarity

Evaluator, NC, SyS, SyC, Homology Inference and Gene Family Clustering. GFC utilizes a sequence similarity score together with a synteny correlation score to infer homology [2]. The original code for GFC is publicly available on BitBucket [3], but unfortunately the modules' implementations are structured in such a way that they are not user friendly. They require the user to manually run one module and afterward use that output to manually set it as input for the next module. This process makes the program tedious to use and not very user friendly. The program does state in which order the modules should be executed, but there are no clear instructions on how to execute them. This project aims to investigate the GFC code and then make it user friendly with Python and investigate the following questions:

- How to best simplify the use of GenFamClust?
- To what extent can the GFC code be reused and how to best structure the program in Python?
- How can we integrate the usage of Neighborhood Correlation into this Python program?

2 Methods

GenFamClust and the original source code for GFC are publicly available on Bitbucket [3]. Figure 2.1 shows the structure and workflow of GFC.

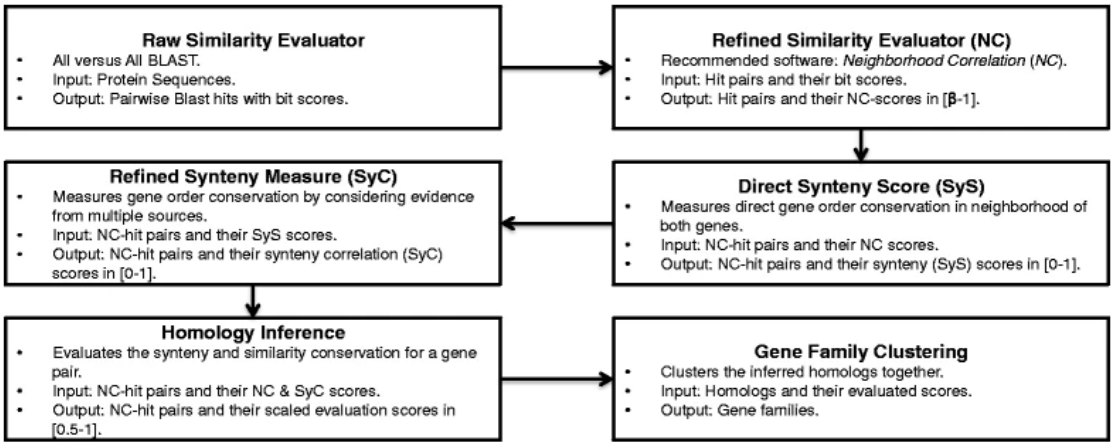

Figure 2.1: Diagram showing all modules and workflow of GFC and comes from Ali et al., 2016 [2], licenced under CC-BY 4.0.

It was decided in this project to not include the first, Raw Similarity Evaluator, and last module, Gene Family Clustering, in the GFC program. The main reasons to exclude the first and last module is because of time constraints, but also because those modules are commonly used and exist in other programs and libraries. So the GFC Python program in this project will consist of the four main modules: NC, SyS, SyC and Homology-Inference module. The NC module calculates a robust sequence similarity score for genes/proteins by looking at the neighborhoods of sequences in a similarity network. The SyS module uses NC-values to calculate a score for how similar the neighborhoods of two genes are. The SyC module utilizes the SyS-scores in a similarity network to calculate a more refined

measure for synteny. The Homology-Inference module determines if a pair of genes have significant similarity by looking at both NC-score and SyC-score. As an input the GFC program will take a tabular output file from BLAST together with synteny files that represents the genes in order on a certain chromosome/contig and the program will output a file that contain gene pairs that the program has determined to have significant similarity, together with the gene pairs NC-score and SyC-score.

Two data sets have been used in this project and although both data sets are from the same source, the intended use for this data are not to validate the GFC method results but to investigate time usage for the modules in the GFC Python program. The large data set consist of 19000 genes from the human genome and 15000 genes from the mouse genome. The small data set consist of roughly 7000 genes from each human and mouse genome. The data used in this project is from [9] (human and mouse BLAST scores) and [1]. All code and test data used for this project can be found and downloaded from this GitHub repository [8].

2.1 NC Module

NC is the first module in our Python program and is what GFC is based on. The method, *Neighborhood Correlation*, or NC, was suggested by Song et al. [11]. NC identifies homologs by looking at a sequence similarity network which shows patterns of gene duplication and domain shuffling in the neighborhoods of sequences in the network. For example, two sequences that have large amounts of similarity will have very similar neighborhoods with strong edges between them and this will increase the NC score for the two sequences. Whereas if one sequence has a large unique neighborhood, this will in turn reduce the NC score between two sequences. The unique neighborhood consists of neighbors from one sequence that does not exist in the other's neighborhood. Base code for this module can be found and downloaded on the Neighborhood Correlation website [9]. The major part of the NC source code is written in Python with Numpy, but the inner

loop of the program is written in C, which requires the user being able to compile a C-file. This is not a problem for most Unix-based systems, however macOS-systems will have to find their own commands from the given Unix-based install instructions NC comes with. Windows users can solve this by using Windows Subsystem for Linux, WSL, and following the install instructions normally. The source code for NC is unfortunately from 2008 which means that it was developed using Python 2.5 which is not compatible with any Python 3.x version. So the first thing we did was to port the NC Python code to the current version of Python, version 3.10. This was done by using a Python program called 2to3 [7] (found in the Python documentation) and then verifying that the code was changed appropriately to work with Python 3.7. An important note about the NC C-part is that the program does have a backup Python implementation in case it fails or is not found, however the Python implementation is greatly slower than the C counterpart and can take over 10 hours to complete when the C implementation will complete in about 5 minutes. So while the C part is not technically necessary for the program to run, it is strongly recommended that the NC module is being run with the C implementation. Other than the 2to3 change and some minor import fixes in the Python code, nothing else needed to be changed in the NC module. NC-score for two genes g_1 and g_2 is defined as:

$$NC(g_1, g_2) = \frac{\sum_{i \in N}(S(g_1, i) - \overline{S}(g_1))(S(g_2, i) - \overline{S}(g_2))}{\sqrt{\sum_{i \in N}(S(g_1, i) - \overline{S}(g_1))^2 \sum_{i \in N}(S(g_2, i) - \overline{S}(g_2))^2}} \tag{2.1}$$

$S(g_1, i)$ is a normalized bit score from a query sequence g_1 and i is a database sequence, N is sequences in the database and $\overline{S}(g_1)$ is the mean $S(g_1, i)$ over all sequences i. Gene pairs with a NC-score over 0.5 can be considered homologs [11], however in the GFC method a threshold of $NC > 0.3$ is used to include some of the unclear gene pairs to infer homology on [2]. Since the NC module is implemented with a Python

library called NumPy, a scientific computing package [6], NumPy is also utilized in the GFC program to speed up some calculations. This of course requires the user to have NumPy installed on their system, but should not be a problem since NumPy is a very commonly used library and the NC script also requires it.

The NC module will be installed as a script on your computer after following the install instructions included with the NC files and since the NC module is a standalone script it will be run separate from the GFC program. The input for this module is All against All BLAST-scores, which is on the same format as the output below, except instead of NC-scores the last column consist of a BLAST bit-score and the module outputs a file that follows the format in figure 2.2:

gene_id	gene_id	nc_score
gene_id	gene_id	nc_score
.	.	.

Figure 2.2: Figure showing the output format for the NC module.

Columns in figure 2.2 are separated by one tab and NC-score ranges from [0-1].

2.2 SyS Module

The Direct Synteny Score, or SyS, calculates a local synteny score by looking at the gene order conservation in a pair of genes' neighborhoods and how similar they are.

SyS-score for two genes g_1 and g_2 is defined as:

$$SyS(g_1, g_2) = max\{NC(a, b) : a \in n(g_1), b \in n(g_2)\} \tag{2.2}$$

where a, b are genes in the neighborhood, $n(g)$, of g_1 and g_2 on a chromosome or contig at most distance k from g_1 and g_2. By distance we mean the number of genes up or down the order of sequences of genes on a certain chromosome/contig. In the article written by Ali et al. [2] they mention a previous study [4] by them, where they determined $k = 5$ to be sufficient for estimating local synteny on genes from Metazoa. $NC(a, b)$ is simply the NC calculated value for genes a and b.

The GFC code for this module is written in Java, which meant that we did not re-use any code in this module for our Python program. Instead we used the definition in the GFC article to build this module in Python.

This module requires a synteny input file together with the NC-file from the previous module. The synteny input file has a three column format where the first column says what chromosome/contig a gene exist on. The second column consist of the exact placement of the gene on a chromosome/contig, although this metric is not currently being used in the program. The Third column consist of the gene identifier. This order of genes is very important when we want to find the neighborhood of genes. Finally this module outputs a file containing the format shown in figure 2.3:

gene_id	gene_id	sys_score
gene_id	gene_id	sys_score
.	.	.

Figure 2.3: Figure showing the output format for the SyS module.

Columns in figure 2.3 are separated by one tab and SyS-value ranges from [0-1].

2.3 SyC Module

Synteny Correlation Score, or SyC, is a score that incorporates more information than just the gene neighborhood of genes. SyC is based on SyS scores and NC-scores over a certain threshold and $SyC(g1, g2)$ is defined as:

$$SyC(g_1, g_2) = \frac{\sum_{i \in N}(SyS(g_1, i) - \overline{SyS}(g_1))(SyS(g_2, i) - \overline{SyS}(g_2))}{\sqrt{\sum_{i \in N}(SyS(g_1, i) - \overline{SyS}(g_1))^2 \sum_{i \in N}(SyS(g_2, i) - \overline{SyS}(g_2))^2}} \quad (2.3)$$

$\overline{SyS}$(g) represent the mean of $SyS(g, i)$, over all genes i such that $SyS(g, i)$ exist. N is a set of genes defined as:

$N = ncHits(g_1) \cap ncHits(g_2)$, where

$ncHits(g) = \{NC(g, j) \geq \beta, j \in Q\}$. j is any gene from the input data, which is denoted by Q. β is a threshold which can be set in the program, but is by default $\beta = 0.3$.
NC and SyC are calculated similarly, however there are two major differences. First is the similarity score used in the calculation, NC uses a sequence similarity score whereas SyC uses a synteny score. The second difference is the set used for the calculation. NC uses the the whole database (of query genes), whereas SyC uses the set consisting of the intersection of NC-hits for both genes that are being calculated. With this gene set we perform the calculation on we measure the similarity of the gene's neighbourhoods instead of the direct sequence similarity of genes.

SyC is also written in Java which means that no code was re-used in the Python program. More about this reasoning in the discussion chapter.

The SyC module takes as input 1) NC-scores and 2) SyS-scores from previous modules and will output a file containing the following format in figure 2.4:

gene_id	gene_id	syc_score
gene_id	gene_id	syc_score
.	.	.

Figure 2.4: Figure showing the output format for the SyC module.

Columns in figure 2.4 are separated by one tab and the SyC scores ranges from [0-1]

2.4 Homology-inference Module

The purpose of this module is to determine if a pair of genes are homologs by using a heuristic decision boundary. A decision boundary is a way for us to classify our data by applying our NC and SyC scores to a pre-defined formula and depending on the result we can classify a pair of genes as homologous. Ali et al. [2] defines the heuristic decision boundary h, for genes g_1 and g_2 as:

$$h(g_1, g_2) = NC(g_1, g_2)^2 + 0.25 \cdot SyC(g_1, g_2)^2 - 0.25 \tag{2.4}$$

The heuristic decision boundary might not perform the best for every dataset. This module takes as input NC scores and SyC scores generated from previous modules and if a pair of genes are determined to be homolog, ie $h(g_1, g_2) > 0$, then it will output following format in figure 2.5:

gene_id	gene_id	nc_score	syc_score
gene_id	gene_id	nc_score	syc_score
.	.	.	.

Figure 2.5: Figure showing the output format for the Homology-Inference module.

3 Results

How to best simplify GenFamClust?

During the investigation of code and documentation the main usability problem was found to be the large amount of manual commands required to run all modules. It was decided that the best way to simplify GFC was to make the Python program handle as many of the commands necessary to run all the different modules. To furthermore simplify the program, separate result files from the different modules is being created and put in a specific folder in the program. This helps the program to not run modules when a result file from it already exist. This also allows a user to choose which module to start or continue with. The reason to put all files in a specific folder is twofold, firstly for the program to automatically find and use the output for previous modules, therefore lessen the amount of manual commands required from the user. Secondly it is easier for the user to locate and move result files in case the user wants them for other reasons than just GFC. Due to time constraints in this project we did not have time to fully integrate the NC module into the main program, therefore the NC module needs to be run separately from the rest of GFC. After the NC module has been successfully run and yielded a result file, the rest of the program is executed with a single command.

Another important part of making GFC more easy to use is creating documentation, both documenting all code but also providing simple instructions on how to install and run the program. The install instructions are especially important when it comes to how easy a program will be to run, since it becomes much harder to run if there are unclear or missing instruc-

tions on how to operate the program. Which is why an installation/running guide has been included together with the code to make it as easy as possible to start using the program. Together with good documented code and the simple nature of Python the program is relatively easy to understand and since the program does not use any data structures other than already existing ones in Python such as lists and dictionaries. By only using standard Python methods of storing and iterating over genes, a user can very quickly identify what the code does, and therefore quickly modify some part of the code to, for example, read from a slightly different file with some extra columns or add some extra information to the final output file.

What is the best way to structure GenFamClust and how much code can be reused?

The best way to structure GFC Python was found to be keeping the overall module structure of its precedessor, but now use a main file to run all modules from. This reduces the amount of manual commands required from the user to run the whole program. However the NC module still is required to be run by the user since it is a script separate from the main program.

As for how much of the code could be reused, the simple answer is some, but not as much as hoped. The reason for this is the majority of code in the original GFC program is written in Java and it was chosen for this project to implement the program in Python. While you definitely can convert Java code into Python it will not solve the structure issue the Java code had, which led to the program requiring many inputs from the user. However the NC module was reused almost completely, except for the fixes to make code compatible with Python 3.x. The Python code for NC was ported to Python 3.x because most likely a user will have a Python 3.x version on their computer. Furthermore Python 2.x is no longer being supported which is a big reason to not use it anymore.

How to best integrate *Neighborhood Correlation* into GenFamClust?

The best way to integrate NC into GFC is to use it as its own Python module. First by downloading the source code from [9] and then put it in a module in the GFC program. This does however mean that it needs to be installed using the included install instructions. To simplify the process of installing the module, relevant install instructions for the NC module have been included in the GFC install instructions, so the user only needs to follow one set of instructions for the whole program.

The resulting Python program for GFC is similar in structure to how it was before, however it now contains a more simple structure of modules and folders to manage the program. As previously mentioned in this report, GFC now contains four modules: NC, SyS, SyC, Homology-Inference module. To run the new GFC Python program, it only requires the user to do a few things, first download the GFC folder from the Github repository [8], then follow the included install instructions for the NC scipt. Then simply follow the running instructions for GFC. The new GFC program only requires 2 commands to run the whole program, first run the NC script by using the command: 'NC_standalone -f some_file_name.dat -o nc.txt'. Then call the GFC main function with the command: 'python3 GFC.py synteny_file1 synteny_file2' to run the rest of the program. For the user this should be an improvement over its predecessor.

The new GFC Python program has some expectations when it comes to running it. First of all it requires NumPy to be installed on the system. Furthermore it is highly recommended to have a C-compiler on the system to speed up the NC module. It also expects the user to supply synteny files for two genomes following the format shown in chapter 2.2 (SyS Module) to the main program. Furthermore to run the NC script the user will have

to supply a file containing BLAST scores for the genes/proteins the user want to infer homology on. If the user can supply both of the synteny files and the file containing BLAST scores, assuming all files follow the required formats shown in this project, then the user should be able to run the GFC program without problems.

NumPy was used to calculate equation 2.3 in the SyC module, which helps the program run a little bit faster compared to using normal Python lists for the calculation. The SyC module ran a few minutes faster with NumPy when the module took approximately 60 minutes to finish. This was not the performance gain we had hoped for with NumPy, but it was nonetheless a small improvement on running times which we gladly welcome for our Python program.

	NC	SyS	SyC	Homology-Inference
small dataset	< 1	13	7	< 1
large dataset	5	68	77	< 1

Table 3.1: Table showing run times (in minutes) for GFC modules with different sized data sets.

Table 3.1 show run time in minutes for the modules in GFC for different sized data sets. As shown in table 3.1, the NC and Homology-Inference module does not contribute much to the overall run time for the GFC program. Even for large data sets the SyS and SyC modules are what takes up the majority of run time in the program. A large portion, 15 minutes, of the SyC run time for the large data set consist of only loading the necessary data into memory. Most likely if the program had access to more RAM memory this time would be reduced.

Unfortunately due to time constraints we were not able produce any direct comparisons with the original GFC code. From the test data we were using, we tried producing the necessary synteny files for the original GFC code, but unfortunately due to poor documentation of the format for the synteny file, we were unable to recreate the required file.

4 Discussion

Python as a language was chosen for this project because of its goal to simplify the usage of GFC and Python is a very simple language to code in which helps a lot when writing code, especially when three out of four modules were required to be rewritten. Python also has a wide range of libraries it can utilize, for example Numpy, SciPy and Matplotlib to help with algorithms and visualizing results. In this project we decided to write most of our own code, this makes the program a good base with the GFC method, and can easily be improved by either 1) adding (or changing/upgrading) modules or 2) incorporating more libraries to increase its effectiveness with the range of available functions but also enable GFC to do more in-depth analysis on genomes with more advanced functionality in the future. However Python is not good at everything, for its increased simplicity it sacrifices speed and memory consumption. In this program we brought down execution time by using some help functions for the big loops, where we pre-compute some part of the inner loops. An example of this is in SyC module where we need to find the mean of $SyS(g1, i)$ and $SyS(g2, i)$ for all sequences i. We help the main loop by pre-calculating the SyS mean for all genes. This helps with execution speed but increases RAM consumption because we now need to save mean SyS values for all genes.

Initially we tried to implement GFC fully with NumPy, however this approach quickly got cancelled due to excessive runtimes compared to using native Python structures when iterating over the data. Instead we used NumPy with some calculations with vectors to lower runtime in the SyC module.

As briefly mentioned in chapter 2.4 (Homology-Inference module), the decision boundary was determined from a specific multispecies dataset. This decision boundary probably does not work well with every dataset out there, and depending how similar a dataset is from the multispecies dataset (used for determining the decision boundary) it might warrant a completely other decision boundary. To find such a decision boundary the user will have to investigate and test a better one themselves.

This project have only been tested for Unix based systems with 16 GB of RAM, since this GFC Python program was developed using a IDE (Integrated Development Environment) in Windows, but installed and tested using the WSL (Windows Subsystem for Linux). Therefore we have only been able to create install and running instructions for Unix systems.

5 Conclusion

GFC has been shown to perform well with homology inference in previous studies, however the original code made it somewhat tedious to use. Most of those usability problems have been solved in this project by writing new code in Python to help automate the process of using the modules the program consist of. Furthermore this project has provided step-by-step instructions for installation and execution of the program. The new Python GFC program took a big step in the right direction when compared to its predecessor in terms of its usability and at the same time opened up the program for upgrades in the future, either swapping out modules or incorporating more advanced methods that could potentially yield better results in determining homology.

Bibliography

[1] Ensembl, n.d. https://www.ensembl.org/index.html, Last accessed on 2021-01-10.

[2] Raja H Ali, Sayyed A Muhammad, and Lars Arvestad. GenFamClust: An Accurate, Synteny-aware and Reliable Homology Inference Algorithm. *BMC Evolutionary Biology*, 16(1):1–19, 2016.

[3] Raja Hashim Ali. GenFamClust, 2019. https://bitbucket.org/rhali/genfamclust/src/master/, Last accessed on 2021-12-05.

[4] Raja Hashim Ali, Sayyed Auwn Muhammad, Mehmood Alam Khan, and Lars Arvestad. Quantitative Synteny Scoring Improves Homology Inference and Partitioning of Gene Families. *BMC Bioinformatics*, 14(15):1–9, 2013.

[5] Malay Kumar Basu, Eugenia Poliakov, and Igor B Rogozin. Domain Mobility in Proteins: Functional and Evolutionary Implications. *Briefings In Bioinformatics*, 10(3):205–216, 2009.

[6] The NumPy Community. NumPy, 2006. https://numpy.org/doc/stable/index.html, Last accessed on 2021-01-10.

[7] Python Documentation. 2to3, 2009. https://docs.python.org/3/library/2to3.html, Last accessed on 2021-12-05.

[8] Jesper Holm. GenFamClust, 2021. https://github.com/Jesperholm98/GenFamClust, Last accessed on 2021-12-05.

[9] Nan Song, Jacob M. Joseph, George B. Davis, Dannie Durand. Neighborhood Correlation, 2008. http://www.neighborhoodcorrelation.org/, Last accessed on 2021-12-04.

[10] NCBI. BLAST, 1989. https://blast.ncbi.nlm.nih.gov/Blast.cgi, Last accessed on 2021-01-10.

[11] Nan Song, Jacob M Joseph, George B Davis, and Dannie Durand. Sequence similarity network reveals common ancestry of multidomain proteins. *PLoS Computational Biology*, 4(5):e1000063, 2008.

Datalogi
www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm