Serdica J. Computing **8** (2014), No 1, 1–14

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

MANAGEABLE WORKFLOWS FOR PROCESSING PARALLEL SEQUENCING DATA

Milko Krachunov, Ognyan Kulev, Valeriya Simeonova, Maria Nisheva, Dimitar Vassilev

Abstract. Data analysis after parallel sequencing is a process that uses combinations of software tools that is often subject to experimentation and on-the-fly substitution, with the necessary file conversion. This article presents a developing system for creating and managing workflows aiding the tasks one encounters after parallel sequences, particularly in the area of metagenomics.

The semantics, description language and software implementation aim to allow the creation of flexible, configurable workflows that are suitable for sharing and are easy to manipulate through software or by hand. The execution system design provides user-defined operations and interchangeability between an operation and a workflow. This allows significant extensibility, which can be further complemented with distributed computing and remote management interfaces.

ACM Computing Classification System (1998): D.2.11, D.1.3, D.3.1, J.3, C.2.4.

Key words: next-generation sequencing, metagenomics, workflow design, data analysis, YAML.

1. Parallel sequencing and metagenomics. Parallel sequencing is a technological process to digitise genetic data from biological organisms, producing computer data in the form of sequences of a four-letter alphabet, with each letter corresponding to a different nucleotide base. It produces considerable amounts of genetic data in the form of thousands to millions of short reads ranging from tens to hundreds of bases.

These short reads represent small fragments of the nucleotide sequences found in a biological sample. These reads can be further combined into much larger datasets and sequences during the data processing. This data is subject to various intensive computational operations, including preprocessing and numeric experiments, that often depend on the specific field of research.

De novo genome sequencing takes reads from specific biological organisms that have not been fully sequenced yet and attempts to assemble them into complete organism genomic sequence. It involves working with terabytes of data at once, in a process that involves data processing operations such as clustering the data, aligning the clusters, combining shorter reads into larger contigs from each aligned cluster and attempting to match those contigs into a single genomics sequence [10]. This process has been mostly streamlined in the various software solutions for de novo assembly that implement every required step to assemble the genomes.

In contrast, metagenomics is a newer field that deals with the analysis of short genetic fragments coming from samples of micro-organism communities found in environments such as soil, water basins and various macro-organisms. The full genomic sequencing of these organisms is technically infeasible due to the huge number of organisms and the computational complexity involved [12], so the researchers focus on the biodiversity and the evolutionary relationships between the species, studying them using less precise methods.

Metagenomics still lacks comprehensive procedures and software tools, and it often deals with numerical experiments that are performed without the availability of common well-defined pipelines. The research methods need to be adapted to the emerging sequencing technologies, which focus on full genome sequencing. Thus they require the provisional combination of multiple software programs with experiment-specific fine-tuning. Such combinations are sometimes also necessitated by the computational challenges that metagenomics datasets pose. [4, 12, 13]

The topic of this paper is the development of a solution and a software package for describing metagenomics as well as generic genomics workflows. The presented system provides all the necessary tools to glue various software packages together, tune them and provide drop-in replacements as needed. It is utilised within a larger work of designing denoising methods and implementing new quality validation techniques in metagenomics [8]. Work is also done on applying it for analysis of complex de novo sequenced genomes such as hexaploid wheat (Triticum aestivum).

2. Describing operations and data conversions in bioinformatics workflows. A key element to the presented software system is an operation description system that has some similarity to function prototyping seen in some programming languages and interfaces seen in other, but is focused on providing aid to classify compatible operations and suggest any necessary conversions between those operations that could be inferred at run time.

The need for operation and data type descriptions arises during the workflow design. Workflow construction tools would need to be aware of the available operations, together with their input and output datatypes to asses their compatibility. At the same time, workflow static validation tools would need to be able to verify this compatibility on an already completed workflow. Furthermore, a workflow execution framework would benefit from a datatype conversion system to deal with non-consequential incompatibilities.

2.1. Data type descriptions. The common data that is subject to analysis in genomics and related studies are DNA sequences and fragments, as well as aminoacid sequences and fragments. Digitally they are represented as sets of textual strings of a four-letter or a 20-letter alphabet in various formats with different forms of additional metadata. The analysis also has to work with sequences of quality scores, sequence clusters and contigs (larger fragments) in various formats, phylogenetic trees, gene and phenotype ontologies, gene annotations and many others. The main area of research in our larger metagenomics work deals mainly with DNA sequences, clusters and phylogenetic trees (evolution-based hierarchical clustering).

To describe the data types accepted and produced by the workflow operations, an extension of the Python class system is used [3]. Python provides significant extensibility features to classes, including the dynamic creation as well as defining class parent-child relationships using arbitrary expressions that can act as a ready-to-use meta language for defining hierarchies using abstract base classes (ABC) [6, 14].

In Python each standard class A can have multiple direct ancestors and descendants, and a class B is considered a subclass of A if it is A, or a subclass of any class that inherits A as a base class (ancestor).

For the purposes of creating and validating a bioinformatics workflow, and for the purposes of handling implicit data conversions, we need a collection of ABC classes to describe our datatypes, and a set of extensions thereof. Below we provide a formal description of a portion of the resulting class system, focusing on the extensions introduced as part of our work.

If we are only interested in the class hierarchy, the definition of a class A can be described as following.

$$A : A \subset B_1, A \subset B_2, \ldots, A \subset B_n \tag{1}$$

The set of class objects $\{A_i\}$ are subject to two relations—*subclass* relation, denoted here with $A_i \subset A_j$, and *membership* relation, denoted here with $x \in A_i$. Excluding their transitivity, these relations are *usually* defined explicitly—every class is defined with its superclasses, and every class member is defined with its class. Using the flexibility of the Python ABC, we will introduce extensions that define new implicit rules between special categories of classes.

Definition 1 (Class itemization)**.** *We introduce $A[B]$ describing the subclass of a container class A whose elements contain only subelements belonging to class B. Element x belongs to $A[B]$ if and only if it belongs to A, and its subelements belong to B. $A'[B']$ is a subclass of $A[B]$ if and only if A' is a subclass of A and B' is a subclass of B. Thus we define the class $A[B]$ as a class whose membership and subclass relations fulfil the following.*

$$x \in A[B] \Leftrightarrow x \in A \wedge \forall i(x_i \in B) \tag{2}$$

$$A'[B'] \subset A[B] \Leftrightarrow A' \subset A \wedge B' \subset B \tag{3}$$

The class $A[B]$ is abstract, it has no actual instances or functionality beyond the thus defined relations. $A[B_1, B_2, B_3]$ is defined in a similar manner as the container classes A containing triplets of the classes B_1, B_2, B_3.

$$x \in A[B_1, B_2, \ldots, B_n] \Leftrightarrow x \in A \wedge \forall i \forall j(x_{i,j} \in B_j) \tag{4}$$

$$\begin{aligned} A'[B'_1, B'_2, \ldots, B'_n] \subset A[B_1, B_2, \ldots, B_n] \Leftrightarrow \\ \Leftrightarrow A' \subset A \wedge B'_1 \subset B_1 \wedge \ldots \wedge B'_n \subset B_n \end{aligned} \tag{5}$$

The itemization is defined as an associative operation between our custom classes so that $A[B[C]]$ is equivalent to $A[B][C]$. This follows from the intuitive expectation that an A container of B containers of C elements is itself an $A[B]$ container of C elements.

Definition 2 (Class set-like operations)**.** *We introduce $A \cup B$, $A \cap B$, $\neg A$ describing union, intersection and complement of classes respectively. X is a subclass of $A \cup B$ if and only if X is a subclass of either A or B, a subclass of $A \cap B$ if and only if it is a subclass of both A and B, and a subclass of $\neg A$ if and only if it is not a subclass of A. The membership relationship is defined analogously to the subclass relationship.*

From a programming perspective, for the type system to be complete, we need to explicitly define some operations between set operation classes, examples of which are listed below.

- $\neg A$ is a subclass of $\neg B$ if B is not a subclass of $\neg A$.
- $A \cup B$ is a subclass of $C \cup D$ if both A and B are subclasses of $C \cup D$.
- $A \cap B$ is a subclass of $C \cap D$ if either A or B is a subclass of $C \cap D$.

Definition 3 (Constant classes)**.** *Constant classes are classes that have only one constant as an instance, e.g. $C = \{4\}$.*

Classes defined through itemization, set operation or constants are considered composite classes. In the software implementation, for all the non-composite abstract classes defined in our typing systems, we can register an arbitrary number of regular Python classes, or our own abstract classes as explicitly defined subclasses of the said abstract class. The system also supports registering Zope interfaces that are sometimes used in Python programs to denote class features instead of subclassing.

For example, we can define the abstract class *Sequence* and register the Python classes *list*, *tuple* as sequences, which signals our typing system that every *list* or *tuple* is a *Sequence* in the same manner as Python's ABC.

The most commonly used datatype our software deals with is *BioSequences* datatype which refers to a collection of nucleotide or aminoacid sequences stored in a file on the disk in an arbitrary sequence format, in a network resource or in memory. A naturally compatible datatype would be $list[BioSequence]$ which describes a list of BioSequence objects, and we would like to be able to implicitly convert between the two.

2.2. Type conversions. Let's define the *BioSequencesLike* abstract data class as the composite class $BioSequences \cup Sequence[BioSequence]$. We would like to define the conversation $BioSequencesLike \rightarrow BioSequences$ by registering a function that handles this conversion. Upon encountering an operation that requires a *BioSequences* object or a *parent* of it that has been instead provided a *BioSequencesLike* or a *child* of it, the defined convertor function for *BioSequencesLike* → *BioSequences* would be implicitly included in the workflow execution.

The convertors are registered by their target class. Let's assume a method accepting class A as input is instead provided with data of class B that is not a subclass of A. The type resolver looks for all convertors registered for the A target class or one of its *subclasses*. Then it selects those with a source that is a *superclass* of B. They are sorted by the generation distance between the convertor source class and A, and the convertor that is the most specific (closest to A) is chosen.

2.3. Service, method and function prototypes.

2.3.1. Function prototypes. Using the available classes in our typing systems, we would like to define the signatures of the functions that would execute bioinformatics operations. A potential prototype signature would look like:

$$\text{alignment-multiple} : \text{align}(\textit{sequences} : \textit{BioSequencesLike}) \rightarrow \\ \rightarrow \textit{BioSequences} \tag{6}$$

2.3.2. Method and serivce prototypes. Instance methods in Python are simply functions that are attributes of a class instance. We define service prototype as a class that contains function prototypes as attributes instead of regular methods. Each service prototype is attached to a named service, for example a service prototype with the align method defined above can be registered for the `multiple-alignment` service. This service can have multiple providers which are required to accept calls to `align` with the same signature.

In other words, a collection (or a class) of method prototypes is called a service, and any class that accepts calls with the signatures defined by these prototypes is called a provider. In this they are very similar to the concept of interface and class in languages such as Java. Unlike interfaces, these service descriptions are not used for any explicit type checking or software validation, but for providing a user interface to create links between the different services, as well as to discover service providers by their type.

When a class is registered as a provider for the `multiple-alignment` service, its methods are not checked if they match the prototypes (which with the Python's ability to dynamically create custom methods is counter-productive), and the service is not flagged as an interface the class implements. On the contrary, the provider is inserted into the service and will be used as a default provider if it has the highest priority among other providers, and the service is exposed to the user as the highest-level interface to the provider's methods, which is allowed to remain unexposed.

It should be noted that the majority of the used hierarchical operations, including typing, type conversions, service discovery and provider discovery, do not depend on the linear ordering of the classes in the hierarchy. For anything that does, the C3 linearisation [1] of the hierarchy will be used, as it is used for the Python method resolution order. Presently, this only happens for the implementation of the providers and data types, and for assigning service priorities. If any extensions to the workflow system itself happen to depend on this, they will likewise be using an order based on the same algorithm, however the class extensions defined here are not prone to such ordering, so such extensions to the typing system would need significant further work.

3. Workflow.

The core part of this paper and the respective software package is the workflow description. The workflow is described as a collection of dataset definitions. The dataset definitions can refer to an explicit data input that is stored on the disk, or to an implicit dataset that is a result of an operation performed over other datasets. Each operation can include complex functionality, comprising mapping an operation over a sequence of datasets or reducing an operation of two datasets over a sequence of datasets.

Like the operation descriptions, the workflows are described in the YAML [2] object notation language. And like operations, any loaded workflow has parameters and methods corresponding to the datasets stored inside with their own prototypes. They provide a service and can be called from other workflows.

The following types of datasets are recognized.

3.1. Target datasets.

3.3.1. Input dataset.

A *input dataset* (or *input value* for simple datatypes) is available before the execution of the workflow, either in a file, internet URL or the execution of another workflow, or it is specified by the user. It is a parameter of the workflow.

The following YAML code defines a source input named `source` and has a type of *BioSequences*. Since it is not optional, it doesn't need a default value to be specified, and key doesn't need to be specified because it matches the dataset name.

```
source: {category: input, optional: false, type: biosequences,
         key: source}
```

3.1.2. Implicit operation dataset. A *implicit operation dataset* is a dataset that does not exists when the execution of the workflow starts (except for caching), but is generated from other datasets within it (implicit or input) using a single method from a given service.

It specifies the method that needs to be called, and its arguments as either references to other datasets using the special YAML type `!ref`, or by using explicit values supported by YAML. The following examples will use $alignment-multiple : align$ from (6) to align the dataset `input` and thus produce the dataset `aligned`, and will also provide a `clustered` dependent of `aligned` that is passed through clustering.

```
aligned:
    category: operation
    method: alignment-multiple:align
    arguments: {sequences: !ref source, quality: 0}

clustered:
    category: operation
    method: clustering:cluster
    arguments: {sequences: !ref aligned,
                threshold: !optional-input threshold}
```

The `aligned` datasets has a type *BioSequences*, while the `clustered` one has a type $Sequence[BioSequences]$, because it contains multiple clusters of sequences. Note that the `!ref` and other input expansions can be applied to the method as well, allowing for the chosen method or class to be parametrised, but this can break certain forms of static checking in strict mode, and better tools for achieving the same goal are under presently consideration.

The `input` and `optional-input` extensions can be used to define input datasets and data values in-place. Usually, only the key name is specified, and the datatype is inferred from the executed operation, but the full syntax from 3.3.1 can also be used. The datasets and data-values defined in-place are not available to other operations or to the workflow users as output.

Optional parameters of the underlying operations are automatically exposed as in-place inputs under a key denoting the implicit operation dataset and its parameter. For example, the `clustered.threshold` key can be used in the absense of an explicit in-place input.

3.1.3. Implicit mapped dataset. An *implicit mapped dataset* is a dataset that is produced after multiple executions of a method of a given service. The source dataset needs to be a *Sequence*, *ItemStore*, or *ItemGenerator*, and the resulting dataset can be converted to any of the three depending on the context. The result is a series containing the results from the application of the methods, represented inside an *ItemGenerator*.

Mapped datasets can be constructed using either the `map-operation` dataset category, or using the `iterate` extension to the regular `operation` dataset category. The following shows examples how both modes can be utilised to align the clusters from the previous example.

```
realigned:
    category: map-operation
    iterate-over: !ref clustered
    method: alignment-multiple:align
    arguments: {sequences: !current}

realigned:
    category: operation
    method: alignment-multiple:align
    arguments: {sequences: !iterate:main clustered}
```

The iterations can be named, and if you include multiple names, multiple iterations will be executed, if the same name is used it simply refers to the same iterate (but it needs to come from the same dataset).

3.1.4. Implicit reduced dataset. An *implicit reduced dataset*, like the mapped dataset, is generated from the elements in another set after the application of a given operation. However, the operation is binary, and it is applied to the result so far, and the next element in it.

Here is an example that uses group alignment to merge all datasets from the aligned clusters in the previous example to construct one big aligned dataset.

```
merged:
    category: reduce-operation
    iterate-over: !ref realigned
    method: alignment-group:align
    arguments: {sequences1: !left, sequences2: !right}

merged:
    category: operation
    method: alignment-group:align
```

```
arguments: {sequences1: !reduce-left:main clustered,
            sequences2: !reduce-right:main clustered}
```

Like mapped dataset, this can also use the `reduce-operation` category as well as the implicit `reduce-left` and `reduce-right` extensions for the regular `operation`.

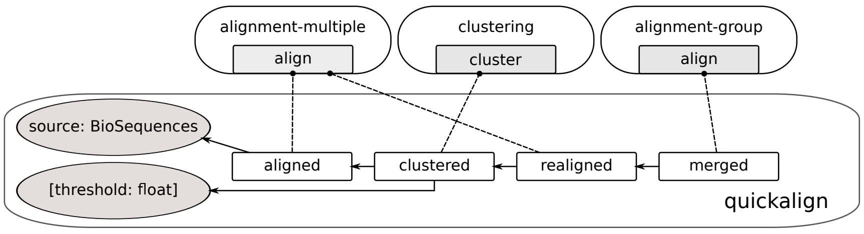

Fig. 1. Example "quickalign" workflow

If all the examples from above are merged into a workflow named `quickalign` as shown on Figure 1, the `merged` dataset can be accessed as a service method and has the following prototype.

(7) $$\text{quickalign} : \text{merged}(\textit{input} : \textit{BioSequencesLike}) \to \textit{BioSequences}$$

3.2. Workflow dependency resolution. Upon requesting a dataset, a dependency graph is constructed. The neighbours of our datasets are the datasets required to construct it with the defined operatons. The directed graph is searched for loops. As loops define cyclic dependencies, they would make the workflow unresolvable and the program will produce an error. If the workflow dataset is resolvable, all operations along the descendants of the requested dataset are executed. At every point where there is a type mismatch, one of the following things happens.

1. If a convertor is available, it is implicitly inserted into the dependency graph, unless implicit conversation has been disabled.

2. If the workflow is running in non-strict mode, a warning is produced and the operation is directly provided with the incompatible input, allowing it to produce its own error. All parts of the workflow that wouldn't generate an error are executed, and should caching of intermediate results be enabled, those intermediate results will be already ready for future executions.

3. If the workflow is running in strict mode, an error is generated before any portion of the workflow has been executed.

Note 1. Format conversion between different file formats is not handled by the convertors described here, as the data storage classes provide the necessary tools to request the data in the needed format. The convertors are used in cases where the data is not explicitly available, for example if it is provided by a database reference, or if a file providing several types of data is available but does not have an explicit data format. It is also used for any conversions required for the complex workflow operations such as map and reduce that need iterate over sequences of data.

4. Extensibility and application.

4.1. Definition of external services. The interchangeability between workflows and service providers is one way to define custom composite services that are a combination of already available service. A much higher level of extensibility will be achieved if the definition of arbitrary services is also supported.

The present software implementation of the workflow execution system provides factories for making services and providers, which are at present only for specific categories of services, which limits flexibility. An example of a YAML definition using a specific factory is shown below. It should be noted that this is not integrated into the workflow description language itself so far.

```
alignment-multiple-muscle:
    factory_name: alignment-multiple-cmdline
    commandline: muscle -in {input} -out {output} -quiet {options}
    options: {diagonals: !switch -diags,
              max_hours: !option:int -maxhours {0:d}}
```

This template factory system will be retained for its convenience, but it will be extended to allow arbitrary command-line tools to be used in the definition of arbitrary services. This will happen either through the introduction of a general purpose factory, or by providing means to define new factories, but this is subject to further research.

4.2. The application of the workflow system. The presented system allows users to define flexible workflows for processing genomics data as well as other data in bioinformatics. They are configurable through parameters, which allows fine-tuning for specific applications. They can be easily shared between users, and the interchangeability between a workflows and operation services makes it easy to combine them arbitrarily, contributing to their flexibility. The ability to change service providers through parameters also contributes to this.

The use of a widely supported object notation like YAML facilitates the development of software to handle these workflows. It is easy to implement programs that can read, write or edit them, partially or in full. It can be used to implement a graphical tool for constructing and modifying them, as well as alternative execution frameworks that can run them.

The ability to add arbitrary processing operations will significantly increase this extensibility and flexibility.

5. Software details. The software developed to implement the workflow system presented in this article is written for Python 2.7, using the Twisted framework [15] to provide for the asynchronous execution of the networking tools, as well as for future support of distributed computing, remote management of the execution system, and as dataset and potential workflow exchange.

It consists of a modular library and a couple of execution commandline scripts. The library has separate modules with the service and provider registry, including the functions to load services and providers from the YAML operation descriptions. It provides a basic workflow executor module that builds the dependency tree and processes all the datasets along it. It also includes a YAML workflow loader that registers the workflows as services and the executor as their provider. Caching and distributed computing, which would be part of the provider module, are pending development. The two scripts allow the execution of a workflow over a single dataset, or a directory of multiple datasets respectively.

It will be released as free software under the X11 license[1] once all the components presented in this article are all in place and allow the formats and APIs are reasonably finalised.

The staging software implementation can perform metagenomics data analysis operations. It supports multiple alignment through MAFFT [7], MUSCLE [5], and sequence clustering through CD-HIT [11]. Arbitrary alignment and clustering tools are also supported. It includes an error detection and correction algorithm [8, 9] that is developed together with the workflow system.

6. Conclusion. An approach for defining manageable bioinformatics workflows has been suggested. A software implementation of this approach is

[1]http://www.xfree86.org/3.3.6/COPYRIGHT2.html#3

being developed, and its staging version can perform multiple sequence alignment, clustering and error detection on metagenomics data. The format aims to be flexible and extensible, as well as to provide manageability by allowing very narrow fine-tuning of every step in the workflow. The workflow execution system has been designed with distributed computing, and well as network and web support.

The presented systems gives a tool that aids bioinformaticians, particularly those that perform analysis on metagenomics data. It provides integration between genomics and bioinformatics processing software packages, and allows them to be included in complex workflows. The workflow language is susceptible to visualisation, which allows users to illustrate their workflows in a way accessible to researchers who are not informaticians. The descriptions can be shared and can reference one another, allowing researchers to share and combine solutions in different Bioinformatics problems.

REFERENCES

[1] Barrett K., B. Cassels, P. Haahr, D. A. Moon, K. Playford, P. T. Withington. A Monotonic Superclass Linearization for Dylan. In: Proceedings of the OOPSLA'96 Conference, ACM Press, 1996, 69–82. doi: 10.1145/236337.236343

[2] Ben-Kiki O., C. Evans, I. döt Net. YAML Ain't Markup Language (YAML). Version 1.2, 3rd Edition, Oct. 2009. `http://yaml.org/spec/1.2/spec.html,December2013`

[3] Chaturvedi S. Python Types and Objects. 2009. `http://www.cafepy.com/article/python_types_and_objects/index.html`, December 2013

[4] Desai N., D. Antonopoulos, J. A. Gilbert, E. M. Glass, F. Meyer. From genomics to metagenomics. *Current Opinion in Biotechnology*, **23** (2012), 72–76.

[5] Edgar R. C. MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, **5** (2004), No. 1, doi: 10.1186/1471-2105-5-113

[6] Guido van Rossum T. Introducing Abstract Base Classes. Python Developer's Guide, 2009. `http://www.python.org/dev/peps/pep-3119`

[7] Katoh K., K. Kuma, H. Toh, T. Miyata. MAFFT version 5: improvement in accuracy of multiple sequence alignment. *Nucleid Acid Research*, **33** (2005), No. 2, 511–518.

[8] Krachunov M. Denoising of Metagenomic Data from High-Throughput Sequencing. Advanced Research in Mathematics and Computer Science, Sofia, 2013.

[9] Krachunov M., D. Vassilev. An approach to a metagenomic data processing workflow. *Journal of Computational Science*, **5** (2014), 357–362. doi: 10.1016/j.jocs.2013.08.003

[10] Li R., H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, Jian Wang, Jun Wang De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, **20** (2010), No. 2, 265–272. doi: 10.1101/gr.097261.109

[11] Li W., A. Godzik. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatic*, **22** (2006), No. 13, 1658–1659. doi: 10.1093/bioinformatics/btl158

[12] Scholz M. B., C.-C. Lo, P. S. G. Chain. Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis. *Current Opinion in Biotechnology*, **23** (2012), 9–15. doi: 10.1093/bioinformatics/btl158

[13] Valverde J., R. Mellado. Analysis of Metagenomic Data Containing High Biodiversity Levels. *PLoS ONE*, **8** (2013), No. 3, 9–15. doi: 10.1371/journal.pone.0058118

[14] Yasskin J. A Type Hierarchy for Numbers. 2010. `http://www.python.org/dev/peps/pep-3141`

[15] M. Zadka, G. Lefkowitz. The Twisted Network Framework. In: Proceedings of the 10th International Python Conference, Alexandria, February 4–7, 2002. `https://twistedmatrix.com/users/glyph/ipc10/paper.html`

M. Krachunov, O. Kulev,
V. Simeonova, M. Nisheva
Faculty of Mathematics and Informatics
Sofia University
5 James Bourchier Blvd
Sofia 1164, Bulgaria
e-mail: `milkok@fmi.uni-sofia.bg`

D. Vassilev
Bioinformatics group
AgroBioInstitute
8, Dragan Tsankov Blvd
Sofia 1164, Bulgaria

Received January 5, 2014
Final Accepted september 25, 2014