OXFORD

Briefings in Bioinformatics, 2024, **25(5)**, bbae367
https://doi.org/10.1093/bib/bbae367
Problem Solving Protocol

Haxe as a Swiss knife for bioinformatic applications: the SeqPHASE case story

Yann Spöri[1] and Jean-François Flot[1,2,*]

[1]Evolutionary Biology & Ecology, Université libre de Bruxelles (ULB), 1050 Brussels, Belgium
[2]Interuniversity Institute of Bioinformatics in Brussels — (IB)[2], 1050 Brussels, Belgium

[*]Corresponding author: Evolutionary Biology & Ecology, Université libre de Bruxelles (ULB), 1050 Brussels, Belgium. E-mail: jflot@ulb.be

Abstract

Haxe is a general purpose, object-oriented programming language supporting syntactic macros. The Haxe compiler is well known for its ability to translate the source code of Haxe programs into the source code of a variety of other programming languages including Java, C++, JavaScript, and Python. Although Haxe is more and more used for a variety of purposes, including games, it has not yet attracted much attention from bioinformaticians. This is surprising, as Haxe allows generating different versions of the same program (e.g. a graphical user interface version in JavaScript running in a web browser for beginners and a command-line version in C++ or Python for increased performance) while maintaining a single code, a feature that should be of interest for many bioinformatic applications. To demonstrate the usefulness of Haxe in bioinformatics, we present here the case story of the program SeqPHASE, written originally in Perl (with a CGI version running on a server) and published in 2010. As Perl+CGI is not desirable anymore for security purposes, we decided to rewrite the SeqPHASE program in Haxe and to host it at Github Pages (https://eeg-ebe.github.io/SeqPHASE), thereby alleviating the need to configure and maintain a dedicated server. Using SeqPHASE as an example, we discuss the advantages and disadvantages of Haxe's source code conversion functionality when it comes to implementing bioinformatic software.

Keywords: programming languages; source-to-source compilation; graphical user interface; bioinformatics; web application; phasing

Introduction

Few biologists are proficient in using command-line tools [1]. As a result, bioinformatic software needs to be usable without the need to open a terminal. Nevertheless, some computer-savvy users prefer interacting with a tool via a command-line interface [2] (CLI) or need a command-line version to integrate the tool into a pipeline such as Galaxy [3–5]. Thus, in practice, most bioinformatic programs require two interfaces—a graphical user interface (GUI) and a CLI.

Even though it is pretty straightforward to program a CLI, adding a GUI to a program can be trickier. GUIs can either be provided in form of a standalone application or via an external program such as a web browser (e.g. Chrome, Firefox, Edge, or Safari). Although specialized toolkits such as Swing, SWT and JavaFX for Java, or Flutter for Dart/C++, allow the creation of platform-independent standalone GUIs, this solution requires installing and maintaining the corresponding piece of software. In contrast, a web browser is preinstalled on most operating systems and is therefore more practical for biologists to use.

Historically, embedding code into a website was usually done using Java Applets or Flash applications with the Netscape Plugin Application Programming Interface [6]. However, newer web browsers do not allow this integration anymore due to security concerns [7]. Instead, modern browsers only allow the interpretation of a particular set of programming languages, namely WebAssembly [8, 9] and ECMAScript (with its better known dialect JavaScript) [10]. Due to this limitation, most programming languages cannot directly execute code inside a web browser. Thus, when programmers want to avoid maintaining two distinct versions of the same software (e.g. one written in JavaScript and the other one in Python), there are three possible ways to write a program that can be run both using a CLI and via a GUI running in a web browser:

- writing the whole program in JavaScript, then using e.g. Node.js (https://nodejs.org) to execute the JavaScript program in a terminal environment. However, high-level programming constructs such as classes are rather awkward to use in JavaScript, except with the help of a scripting language such as CoffeeScript [11];
- writing the program with a CLI that runs on a web server. The GUI can then communicate with the program running on the web server and visualize its results (e.g. via BioJS [12]). Many scripting language such as PHP, Perl, Python, or Ruby support communication via the common gateway interface (CGI) [13]. Nevertheless, setting up and maintaining such a dedicated public server can be time and resource-costly. It also requires users' data to be sent over to the server via internet, which can be a problem in case of large and/or sensitive datasets;
- using a programming language that allows the conversion of the source code into the source code of various other programming languages (a process variously called 'trans-compiling', 'transpiling', or 'cross-compiling' depending on

Received: October 17, 2023. **Revised:** May 19, 2024. **Accepted:** July 17, 2024

authors, and that is perhaps better described as 'source-to-source translation'). A JavaScript version of the code can then run inside a web browser while another version of the program (e.g. in Python) can be used in a terminal environment.

One example of programming language enabling the latter approach is Haxe, a general purpose, highly versatile object-oriented programming language [14]. Although other source-to-source compilers exist, such as Dafny [15], Haxe is the most widely used among them. The source code of Haxe programs can be converted into the source code of a variety of other languages including Java, C++, JavaScript and Python [16]. Using Haxe, it is thus relatively easy to create multiple versions of the same tool—e.g. a JavaScript version that runs inside a web browser and a Python version that can run as a terminal application. Furthermore, Haxe supports sophisticated programming paradigms such as syntactic macros [17], making it an excellent choice for writing bioinformatic applications.

To illustrate the usefulness of Haxe as a Swiss knife for bioinformatic applications, we tell here the story of how we used Haxe to revive an old but very useful piece of code, SeqPHASE [18], originally written in Perl + CGI code, by reimplementing it in Haxe and then converting the Haxe code into JavaScript (for the graphical version) and Python (for the command-line one).

Material and methods

SeqPHASE was written in 2010 to address a pressing practical issue in population genetics: how to input FASTA files into PHASE [19] (a program originally written to infer the most probable pairs of haplotypes in a population of diploid organisms for which genotypes have been determined, but designed for microsatellite length polymorphism data and not for DNA sequences), and how to turn the output of PHASE back into FASTA [18].

Until that point, a tool allowing this conversion existed as part of the Windows program DNAsp [20, 21], but with several severe shortcomings [18], and conversion was impossible on other operating systems such as Linux or macOS. Because it filled an important need in the biological community, SeqPHASE was immediately adopted and cited a relatively large number of times (more than 500 times since published in 2010).

However, its implementation in Perl + CGI posed important security issues, and, at some point, university computer infrastructure administrators became very reluctant to host this piece of code on public servers as it could offer an entrance point to hackers. Out of this necessity, it was therefore decided to reimplement SeqPHASE completely, this time in Haxe, as we needed to provide both a user-friendly web browser tool and a command-line version for use in pipelines and on computer clusters.

For the sake of simplicity and cost saving, a choice was made to host the Haxe source code on GitHub (https://github.com/eeg-ebe/SeqPHASE). Compiling the Haxe part of the code into JavaScript resulted in a fully functional web browser tool accessible via GitHub Pages (https://eeg-ebe.github.io/SeqPHASE), whereas compiling the same Haxe code into Python produced a cross-platform Python script (with a CLI) made available for download on the same website (https://eeg-ebe.github.io/SeqPHASE/download.html).

Results

The source code of the reimplemented SeqPHASE program is available at https://github.com/eeg-ebe/SeqPHASE and licensed under the Apache 2.0 license. A web page allowing the user to run and/or download the program is available on GitHub Pages (https://eeg-ebe.github.io/SeqPHASE).

The reimplementation consists of the following files:

- a series of static HTML pages including a menu page, a FAQ page, and so on;
- two dynamic web pages where users can run respectively the first (FASTA to PHASE) and second step (PHASE to FASTA) of the SeqPHASE program. The web pages directly call the corresponding JavaScript codes;
- a download page that allows users to obtain a zipped archive of the GUI version of the program (once unzipped, users can run SeqPHASE offline inside a web browser by double-clicking the index.html file found inside that archive) as well as Python command-line versions of the two steps of SeqPHASE.

During the reimplementation process we discovered small bugs in the original code and corrected them, namely:

- for Step 1 (FASTA to PHASE), every sequence in the inputted FASTA files should have the same length, but it was possible to bypass this check by ordering the sequences by decreasing length;
- for Step 2 (PHASE to FASTA), the sequences of individuals were sorted apart if the name of one sequence was a full prefix of a longer name of another sequence (e.g. the name of sample1a and sample1b would be sorted apart if there was another sequence with the name sample1abc1a).

Although it is unlikely that these bugs had an impact on the accuracy of the results, it sometimes resulted in the program running on inputs that contained errors, instead of reporting those errors.

Table 1. Versions of the different compilers/interpreters used for our benchmark

	Linux	macOS
Haxe	haxe 4.2.4	haxe 4.2.4
Lua	lua 5.1.5	lua 5.4.6
Neko	neko 2.3.0	neko 2.3.0
Python	python 3.10.12	python 3.9.10
Perl	perl 5.34.0	perl 5.30.3
Java	java 11.0.22	java 15.0.1
Node.js	node 12.22.9	node 20.11.0
C++	g++ 11.4.0	clang 1300.0.27.3

Time usage

In order to analyze the runtime usage of the SeqPHASE programs, we used Haxe to create different program versions with C++, Lua, Neko, Python, JavaScript (Node.js), and Java as target languages (Table 1): one program version where Haxe's dead code elimination algorithm was turned on, one version where Haxe's dead code elimination algorithm was turned off, and one version where Haxe's dead code elimination algorithm was limited to classes in the Haxe standard library. This process resulted in 18 executables (= six target programming languages $*$ three dead code elimination strategies) for the FASTA to PHASE conversion process as well as 18 other executables for the PHASE to FASTA conversion process.

Twenty FASTA data files of variable sizes were generated using SimCoal [22]. We then launched the 18 programs as well as the two original Perl programs 120 times on each of these 20 datasets and measured the corresponding time usages (compare Fig. 1 and Table 2). To evaluate startup times, we also created Hello World

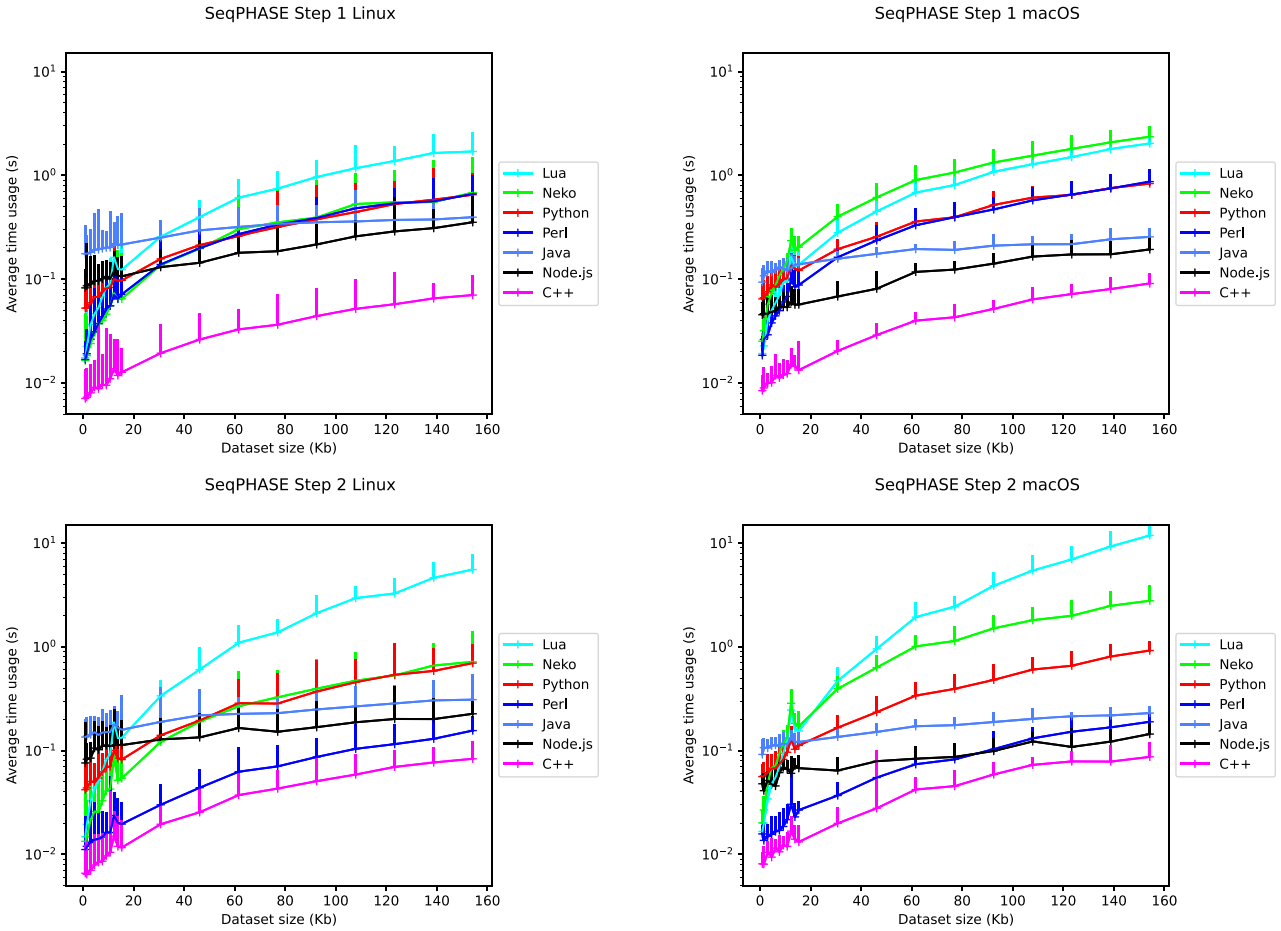

Figure 1. Measurement of the average time usage the SeqPHASE program takes to convert different dataset sizes. The top panels show the calculation time needed for the conversion of the FASTA file into the PHASE input file format, while the bottom panels show the conversion of the PHASE output file format to FASTA file format, for Linux (left) and macOS (right). The error bars indicate the measured minimum and the maximum calculation time measured. The minimum, maximum, standard deviation, average, and median time usage of these runs for the different program versions and conversions are listed in Table 2.

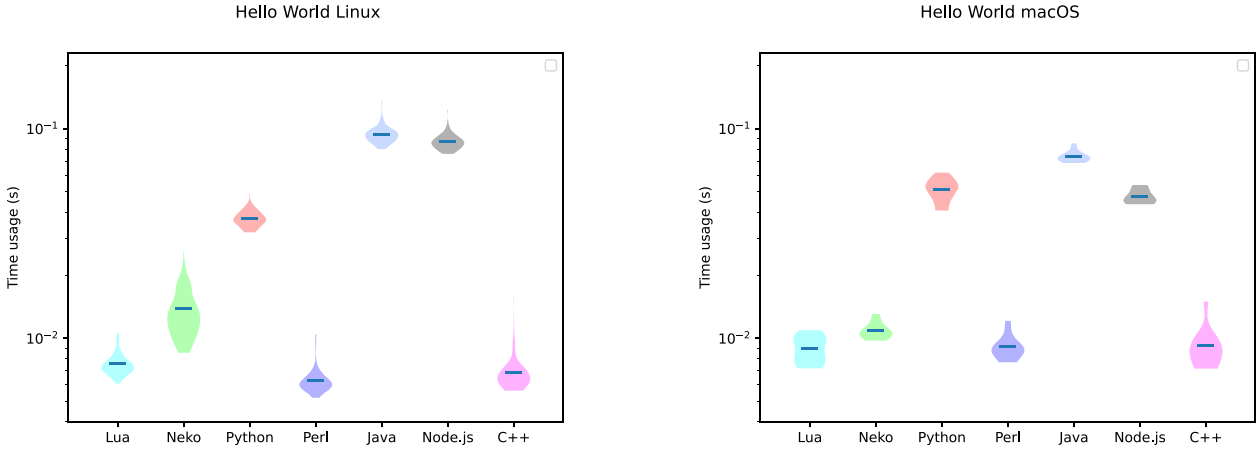

Figure 2. For this plot, each of the seven Hello World executables (one Perl executable + six executables created by source-to-source translation of the Haxe Hello World source code into the corresponding target language) were launched 120 times. The left panel shows the time usage of the Hello World programs on a Linux machine, while the right panel shows the corresponding analysis on a computer running the macOS operating system. The minimum, maximum, standard deviation, average, and median time usage of these runs for the different program versions and conversions are listed in Table 3.

executables by source-to-source translating a Haxe Hello World program to the different target programming languages, as well as writing a Perl Hello World program (Fig. 2 and Table 3).

All datasets and source codes used in this benchmark are available at https://github.com/eeg-ebe/SeqPHASE_time_mesurements.

Table 2. Time usage (in ms) for different dataset sizes for the two different steps of SeqPHASE on Linux and macOS. The table lists the minimum, maximum, standard deviation, average, and median time of the Perl, C++, Lua, Python, Neko, Node.js, and Java SeqPHASE programs

Program	Linux					macOS				
	Min	**Max**	σ	**Avg**	**Med**	**Min**	**Max**	σ	**Avg**	**Med**
	Smallest dataset (4kb)									
	FASTA -> PHASE									
Lua	22.3	62.6	10.9	34.4	28.4	22.7	53.7	9.8	33.9	30.2
Neko	18.3	86.2	6.7	31.6	31.7	31.9	46.8	4.1	40.9	41.6
Python	52.9	160.8	32.7	87.3	68.1	66.3	192.9	38.7	107.9	89.6
Perl	19.1	33.2	2.5	23.6	23.2	26.4	31.8	1.9	28.4	27.9
Java	176.1	372.9	19.9	208.3	205.5	103.6	148.9	9.7	124.0	124.2
Node.js	86.3	221.9	11.9	104.9	102.7	46.5	64.6	5.2	56.8	58.3
C++	7.1	27.3	1.8	9.5	9.1	8.3	14.2	1.3	11.0	11.1
	PHASE -> FASTA									
Lua	18.5	59.0	9.2	29.7	24.8	19.9	49.5	9.9	30.1	26.0
Neko	14.8	41.6	5.6	25.3	24.8	25.6	44.9	4.9	33.2	32.3
Python	42.3	149.3	31.8	73.8	54.2	53.8	166.8	35.8	90.9	70.4
Perl	12.0	22.9	2.0	14.3	13.8	13.6	21.0	2.3	17.3	17.4
Java	134.4	310.8	15.5	162.8	161.5	106.4	132.8	8.0	118.6	118.8
Node.js	79.1	150.5	9.9	99.1	97.6	41.1	66.1	6.1	54.9	56.6
C++	6.4	14.6	1.1	8.2	7.9	8.0	23.5	2.6	10.7	10.2
	Largest dataset (156kb)									
	FASTA -> PHASE									
Lua	1700.0	2643.1	127.3	2086.5	2070.8	2030.6	2846.0	265.1	2392.9	2397.2
Neko	634.9	1495.4	105.9	1110.4	1120.6	2257.7	3211.8	287.5	2710.3	2689.1
Python	632.3	1300.6	78.7	793.6	776.7	834.1	1188.7	118.8	1034.0	1089.4
Perl	664.0	990.5	46.1	761.7	756.7	871.1	1144.8	98.3	1014.7	1043.1
Java	371.3	646.4	34.2	449.6	445.8	241.9	315.8	22.0	274.2	272.7
Node.js	353.7	629.5	28.8	401.0	395.9	193.5	265.8	20.0	231.5	235.1
C++	69.6	109.8	6.4	81.7	81.1	84.2	124.3	9.4	100.8	102.0
	PHASE -> FASTA									
Lua	5547.6	8167.4	317.7	6459.5	6406.4	11602.5	16211.3	1502.0	13441.0	13343.5
Neko	716.1	1447.3	103.8	1035.9	1052.9	2695.3	3888.6	409.4	3266.8	3200.4
Python	696.4	1153.8	67.5	816.5	810.0	874.0	1318.2	115.7	1040.3	1064.1
Perl	155.9	217.2	12.4	177.8	175.6	189.4	241.1	16.6	216.4	215.1
Java	309.6	573.4	29.8	366.7	362.3	223.7	275.6	15.6	248.5	241.5
Node.js	226.1	442.3	21.2	263.1	259.9	144.4	192.5	12.6	164.5	163.9
C++	83.4	147.6	8.2	99.5	97.7	86.8	129.6	11.0	105.8	108.4

Table 3. Time usage (in ms) of the Hello World program on Linux and macOS. The table lists the minimum, maximum, standard deviation, average, and median time of the Perl, C++, Lua, Python, Neko, Node.js, and Java Hello World programs

Program	Linux					macOS				
	Min	**Max**	σ	**Avg**	**Med**	**Min**	**Max**	σ	**Avg**	**Med**
Lua	6.1	10.6	0.9	7.6	7.4	7.2	11.0	1.3	9.0	9.6
Neko	8.5	26.0	3.5	13.9	13.0	9.8	13.1	1.0	11.0	10.7
Python	32.1	49.0	3.2	37.5	37.2	40.7	61.8	6.6	51.6	53.6
Perl	5.2	10.5	0.9	6.3	6.14	7.7	12.1	1.3	9.2	9.0
Java	80.0	136.0	8.6	94.0	93.6	68.7	85.2	4.8	73.9	73.9
Node.js	75.9	123.2	7.1	87.0	86.5	43.7	53.9	3.7	47.7	47.2
C++	5.6	15.9	1.4	6.9	6.6	7.2	15.0	2.2	9.3	9.0

When comparing the average calculation time needed to convert a FASTA file to a PHASE input file or a PHASE output file back into a FASTA file, the C++ version greatly outperformed all other versions of the SeqPHASE program, followed by—for small datasets—the Python / Neko / Lua versions. However, for larger datasets the overhead of starting a virtual machine with a long

startup time can pay off. In that case, the Node.js and / or Java version may outperform every other versions except for the C++ version.

Although the C++ version was the fastest, we decided to not provide it for download as we found it challenging to compile a C++ program that runs on all computers. Instead, we opted for the Python version because Python is a well-known programming language already installed on most computers and because speed is not that much of an issue when it comes to small, straightforward programs that are running in less then 100 ms, such as SeqPHASE. However, in case high performance is needed, building a C++ version for a particular target computer with platform-specific code optimization would be the best option.

Even though this assumption still needs to be verified, we believe that Chrome and Node.js would need, on average, the same amount of time to execute the JavaScript version of the SeqPHASE program since both Chrome and Node.js are using the same JavaScript engine (namely, V8 [23]).

When comparing the dead code elimination strategies for the different target programming languages, we did not observe any differences in the execution times of the versions created when the dead code elimination strategy was applied to the full source code or was limited to the Haxe standard library (the default option). However the versions created without any dead code elimination were significantly slower for most programming languages, except for C++, Neko, and Java for which no difference was observed.

Discussion

Through the example of SeqPHASE's reimplementation, we illustrate how Haxe coding is a valuable yet still underused approach for bioinformaticians to make programs available online to large audiences, including both biologists with no command-line proficiency and computer scientists requiring command-line tools to run on computing clusters. The SeqPHASE website can be used as a template for users wishing to experiment with using Haxe as a valuable alternative to CGI apps, for instance.

Although we chose a fairly simple program to illustrate and benchmark our proposed approach, another more complex example of a bioinformatic tool we reimplemented in Haxe in replacement for a previous perl+CGI version is https://eeg-ebe.github.io/Champuru [24, 25]. Moreover, three further tools we directly implemented in Haxe are available online at https://eeg-ebe.github.io/HaplowebMaker and https://eeg-ebe.github.io/CoMa [26] as well as https://eeg-ebe.github.io/KoT [27].

Since websites are running inside a sandboxed web browser, tools that run inside websites are also advantageous for users who do not want to install a particular software locally due to security concerns. However, compared to Perl, Python and other programming languages traditionally used in bioinformatics, Haxe suffers from certain drawbacks: (1) the current unavailability of a 'BioHaxe' library of functions facilitating the import and processing of biological data. It is our hope to provide such a library in the future; (2) debugging a particular Haxe program may require to take a look at its translation into the target language, which necessitates some understanding of this language (in addition to knowing Haxe); (3) although Haxe supports the languages most often used in bioinformatics (Python, C++, JavaScript, Java), some other languages such as Perl and R are not supported yet.

Key Points

- The programming language Haxe allows designers of bioinformatic applications to maintain a single code for both command-line and graphical-user-interface versions of their program.
- This Haxe source code can then be compiled into various languages such as C++, JavaScript, or Python.
- As a case study to illustrate Haxe's usefulness for bioinformatics, we reimplemented in Haxe the previously published program SeqPHASE (originally written in Perl) and compared the performances of translated C++, Lua, Python, Neko, Node.js, and Java versions.

Funding

Open access fees for the present article were supported by the Fonds de la Recherche Scientifique - FNRS via Grant n°T.0078.23 to JFF.

Conflicts of interest: None declared.

References

1. Troyanskaya OG. Don't fear the command line! *Cell* 2011;**144**: 842–3. https://doi.org/10.1016/j.cell.2011.02.042.
2. Voronkov A, Martucci LA, Lindskog S. System administrators prefer command line interfaces, don't they? An exploratory study of firewall interfaces. In: *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. Berkeley, CA: USENIX Association, 2019, p. 259–271.
3. Goecks J, Nekrutenko A. James Taylor, and the Galaxy team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol* 2010;**11**:R86. https://doi.org/10.1186/gb-2010-11-8-r86.
4. Blankenberg D, Von Kuster G, Coraor N. *et al*. Galaxy: a web-based genome analysis tool for experimentalists. *Curr Protoc Mol Biol* 2010;**89**:19.10.1–21. https://doi.org/10.1002/0471142727.mb1910s89.
5. Taylor J, Schenck I, Blankenberg D. *et al*. Using galaxy to perform large-scale interactive data analyses. *Curr Protoc Bioinformatics* 2007;**19**:10.5.1–25. https://doi.org/10.1002/0471250953.bi1005s19.
6. Lammarsch T, Aigner W, Bertone A. *et al*. A comparison of programming platforms for interactive visualization in web browser based applications. In: *12th International Conference Information Visualisation*, London, UK: IEEE, 2008, p. 194–199. https://doi.org/10.1109/IV.2008.34.
7. Ayyagari R, Figueroa N. Is seeing believing? Training users on information security: evidence from Java applets. *J Inf Syst Educ* 2017;**28**:115.
8. Haas A, Rossberg A, Schuff DL. *et al* Bringing the web up to speed with WebAssembly. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York: Association for Computing Machinery, 2017, p. 185–200.
9. Perkel JM. No installation required: how WebAssembly is changing scientific computing. *Nature* 2024;**627**:455–6. https://doi.org/10.1038/d41586-024-00725-1.
10. Brock AW, Eich B. Javascript: the first 20 years. *Proc ACM Program Lang* 2020;**4**:1–189. https://doi.org/10.1145/3386327.

11. Burnham T. *CoffeeScript: accelerated JavaScript development*. The Pragmatic Bookshelf, 2015. ISBN 978-1-941222-26-3.
12. Gómez J, García LJ, Salazar GA. *et al*. BioJS: an open source JavaScript framework for biological data visualization. *Bioinformatics* 2013;**29**:1103–4. https://doi.org/10.1093/bioinformatics/btt100.
13. Gundavaram S. *CGI Programming on the World Wide Web*. St. Sebastopol, CA: O'Reilly & Associates, 1996. ISBN 978-1-56592-168-9.
14. Dasnois B. *haXe 2 Beginner's Guide*. Birmingham, UK: Packt Publishing Ltd, 2011. ISBN 978-1-849512-56-5.
15. Koenig J, Leino KRM. Getting started with Dafny: a guide. In: Nipkow T, Grumberg O, Hauptmann B (eds). *Software Safety and Security*, Amsterdam, The Netherlands: IOS Press, 2012, p. 152–81. https://doi.org/10.3233/978-1-61499-028-4-152.
16. Štrekelj D, Leventić H, Galić I. Performance overhead of Haxe programming language for cross-platform game development. *Int J Electr Comput Eng Syst* 2015;**6**:9–13.
17. Standish TA. Extensibility in programming language design. In: *Proceedings of the May 19-22, 1975, National Computer Conference and exposition*, Anaheim, California: ACM Press, 1975, p. 287. https://doi.org/10.1145/1499949.1500003.
18. Flot J-F. SeqPHASE: a web tool for interconverting PHASE input/output files and FASTA sequence alignments. *Mol Ecol Resour* 2010;**10**:162–6. https://doi.org/10.1111/j.1755-0998.2009.02732.x.
19. Stephens M, Smith NJ, Donnelly P. A new statistical method for haplotype reconstruction from population data. *Am J Hum Genet* 2001;**68**:978–89. https://doi.org/10.1086/319501.
20. Rozas J, Rozas R. DnaSP, DNA sequence polymorphism: an interactive program for estimating population genetics parameters from DNA sequence data. *Comput Appl Biosci* 1995;**11**:621–5. https://doi.org/10.1093/bioinformatics/11.6.621.
21. Rozas J, Ferrer-Mata A, Sánchez-DelBarrio JC. *et al*. DnaSP 6: DNA sequence polymorphism analysis of large data sets. *Mol Biol Evol* 2017;**34**:3299–302. https://doi.org/10.1093/molbev/msx248.
22. Excoffier L, Novembre J, Schneider S. SIMCOAL: a general coalescent program for the simulation of molecular data in interconnected populations with arbitrary demography. *J Hered* 2000;**91**:506–9. https://doi.org/10.1093/jhered/91.6.506.
23. Heller M. *What is Node.js? The JavaScript Runtime Explained*. InfoWorld, 2017.
24. Flot J-F. Champuru 1.0: a computer software for unraveling mixtures of two DNA sequences of unequal lengths. *Mol Ecol Notes* 2007;**7**:974–7. https://doi.org/10.1111/j.1471-8286.2007.01857.x.
25. Spöri Y, Flot J-F. Champuru 2: improved scoring of alignments and a user-friendly graphical interface (arXiv:2405.06032). 2024. https://doi.org/10.48550/arXiv.2405.06032.
26. Spöri Y, Flot J-F. HaplowebMaker and CoMa: two web tools to delimit species using haplowebs and conspecificity matrices. *Methods Ecol Evol* 2020;**11**:1434–8. https://doi.org/10.1111/2041-210X.13454.
27. Spöri Y, Stoch F, Dellicour S. *et al*. KoT: an automatic implementation of the K/θ method for species delimitation bioRxiv, page 2021.08.17.454531, 2021. https://doi.org/10.1101/2021.08.17.454531.