nature methods

Perspective

https://doi.org/10.1038/s41592-023-01832-z

Julia for biologists

Received: 21 September 2021

Accepted: 27 February 2023

Published online: 6 April 2023

Check for updates

Elisabeth Roesch[1,2,3], Joe G. Greener[4], Adam L. MacLean[5], Huda Nassar[6], Christopher Rackauckas[3,7,8], Timothy E. Holy[9] & Michael P. H. Stumpf[1,2,10,11]

Major computational challenges exist in relation to the collection, curation, processing and analysis of large genomic and imaging datasets, as well as the simulation of larger and more realistic models in systems biology. Here we discuss how a relative newcomer among programming languages—Julia—is poised to meet the current and emerging demands in the computational biosciences and beyond. Speed, flexibility, a thriving package ecosystem and readability are major factors that make high-performance computing and data analysis available to an unprecedented degree. We highlight how Julia's design is already enabling new ways of analyzing biological data and systems, and we provide a list of resources that can facilitate the transition into Julian computing.

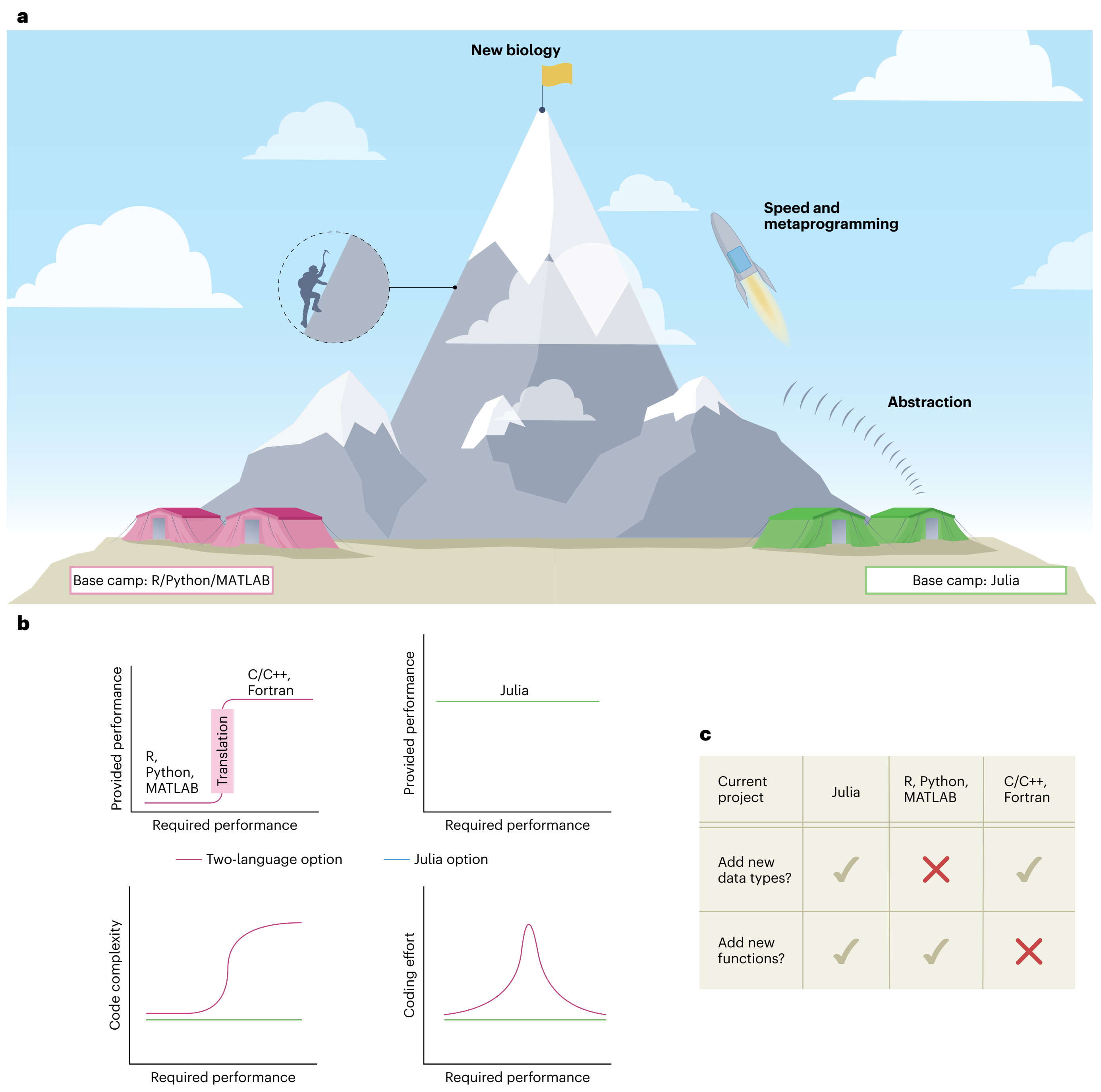
Computers are tools. Like pipettes or centrifuges, they allow us to perform tasks more quickly or efficiently, and like microscopes, they give us new, more detailed insights into biological systems and data. Computers allow us to develop, simulate and test mathematical models of biology and compare models with complex datasets. As computational power evolved, solving biological problems computationally became possible, then popular and, eventually, necessary[1]. Entire fields such as computational biology and bioinformatics emerged. Without computers, the reconstruction of structures from X-ray crystallography, NMR or cryogenic electron microscopy methods would be impossible. The same goes for the 1000 Genomes Project[2], which used computer programs to assemble and analyze the DNA sequences generated. More recently, vaccine development has benefited from advances in algorithms and computer hardware[3].

Programming languages are also tools. They make it possible to instruct computers. Some languages are good at specific tasks (think Perl for string processing tasks or R for statistical analyses), whereas others—including C/C++ and Python—have been used with success across many different domains. In biomedical research, the prevailing languages have arguably been R[4] and Python[5]. Much of the high-performance backbone supporting computationally intensive research that is hidden from most users, however, continues to rely on C/C++ or Fortran. Computationally intensive studies are often initially designed and prototyped in R, Python or MATLAB and subsequently translated into C/C++ or Fortran for increased performance. This is known as the two-language problem[6].

This two-language approach has been successful but has limitations (Fig. 1a). When moving an implementation from one language to another, faster, programming language, verbatim translation may not be the optimal route: faster languages often provide the programmer with higher autonomy to choose how memory is accessed or allocated or to employ more flexible data structures[7]. Exploiting such features may involve a complete rewrite of the algorithm to ensure faster implementation or better scaling as datasets grow in size and complexity. This requires expertise across both languages, but also rigorous testing of the code in both languages.

Julia[8] is a relatively new programming language that overcomes the two-language problem. Users do not have to choose between ease of use and high performance. Julia has been designed to be easy to program in and fast to execute (Fig. 1b). This efficiency and the growing ecosystem of state-of-the-art application packages (Table 1 and Fig. 2) and introductions[7,9] make it an attractive choice for biologists.

[1]School of Mathematics and Statistics, University of Melbourne, Melbourne, Victoria, Australia. [2]Melbourne Integrative Genomics, University of Melbourne, Melbourne, Victoria, Australia. [3]JuliaHub, Somerville, MA, USA. [4]Medical Research Council Laboratory of Molecular Biology, Cambridge, UK. [5]Department of Quantitative and Computational Biology, University of Southern California, Los Angeles, CA, USA. [6]RelationalAI, Berkeley, CA, USA. [7]Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, USA. [8]Pumas-AI, Centreville, VA, USA. [9]Departments of Neuroscience and Biomedical Engineering, Washington University in St. Louis, St. Louis, MO, USA. [10]School of BioSciences, The University of Melbourne, Melbourne, Victoria, Australia. [11]ARC Centre of Excellence for the Mathematical Analysis of Cellular Systems, Melbourne, Victoria, Australia. e-mail: mstumpf@unimelb.edu.au



Current project	Julia	R, Python, MATLAB	C/C++, Fortran
Add new data types?	✓	✗	✓
Add new functions?	✓	✓	✗

Fig. 1 | Julia is a tool enabling biologists to discover new science. a, In the biological sciences, the most obvious alternatives to the programming language Julia are R, Python and MATLAB. Here we contrast the two potential pathways to new biology with a mountaineering analogy. The top of the mountain represents new biology[49]. There are two potential base camps for the ascent: base camp 1 (left, red) is R/Python/MATLAB. Base camp 2 (right, green) is Julia. To get to the top, the mountaineer, representing a researcher, needs to overcome certain obstacles, such as a glacier and a chasm. These represent research hurdles, such as large and diverse datasets or complex models. Starting at the Julia base camp, the mountaineer has access to efficient and effective tools, such as a bridge over the glacier and a rocket to simply fly over the chasm. These represent Julia's top three language design features: abstraction, speed and metaprogramming. With these tools, the journey to the top of the mountain becomes much easier for the excursionist. Julia allows biologists to not be held back by the problems discussed in **b** and **c**. **b**, The two-language problem refers to having separate languages for algorithm development and prototyping (such as R or Python) and production runs (such as C/C++ or Fortran), respectively. Julia was designed to be good at both tasks, which can reduce programming efforts and software complexity. **c**, The expression problem refers to the effort required by users to define new (optimized) data types and functions that can be added to existing external code bases.

Biological systems and data are multifaceted by nature, and to describe them or model them mathematically requires a flexible programming language that can connect different types of highly structured data (Fig. 1c). Three hallmarks of the language make Julia particularly suitable for meeting current and emerging demands of biomedical science: speed, abstraction and metaprogramming.

In this article, we discuss each language feature and its relevance in the context of one concrete biological example per feature. An additional example per feature can be found in the Supplementary Information. Furthermore, in Supplementary Table 1, we provide a summary of why we believe Julia is a good programming language for biologists. Supporting online material is provided in a GitHub repository at

Table 1 | Julia provides a rich package ecosystem for biologists

Community	Topic	Example packages
JuliaData	Data manipulation, storage, and input and output	DataFrames.jl, JuliaDB.jl, DataFramesMeta.jl and CSV.jl
JuliaPlots	Data visualization	Plots.jl, Makie.jl, StatsPlots.jl and PlotlyJS.jl
JuliaStats	Statistics and machine learning	Distributions.jl, GLM.jl, StatsBase.jl, Distances.jl, MixedModels.jl, TimeSeries.jl, Clustering.jl, MultivariateStats.jl and HypothesisTests.jl.
BioJulia	Bioinformatics and computational biology	BioSequences.jl, BioStructures.jl, BioAlignments.jl, FASTX.jl and Microbiome.jl
JuliaImages	Image processing	Images.jl, ImageSegmentation.jl, ImageTransformations.jl and ImageView.jl
EcoJulia	Ecological research	SpatialEcology.jl, EcologicalNetworks.jl, Phylo.jl and Diversity.jl
SciML	Scientific machine learning	DifferentialEquations.jl, ModelingToolkit.jl, DiffEqFlux.jl and Catalyst.jl
FluxML	Machine learning	Flux.jl, Zygote.jl, MacroTools.jl, GeometricFlux.jl and Metalhead.jl

Related packages are organized in package communities. In this table, we present an overview of the package communities we consider to be most relevant to biologists.

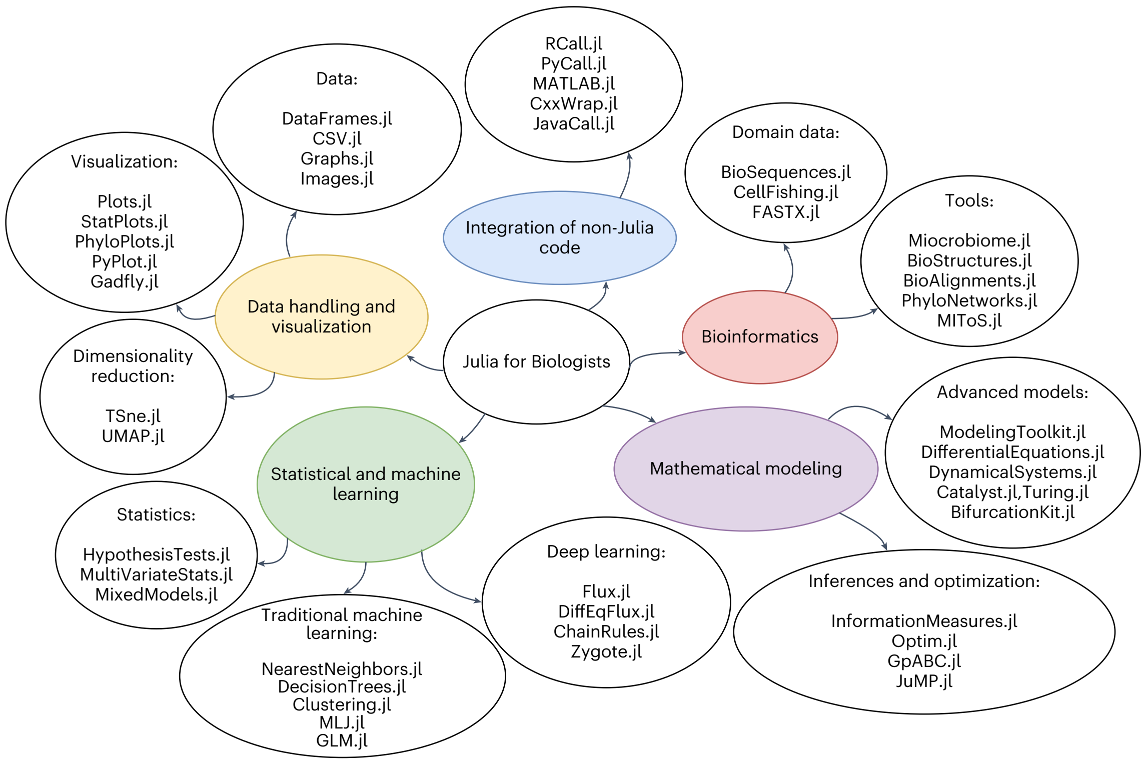


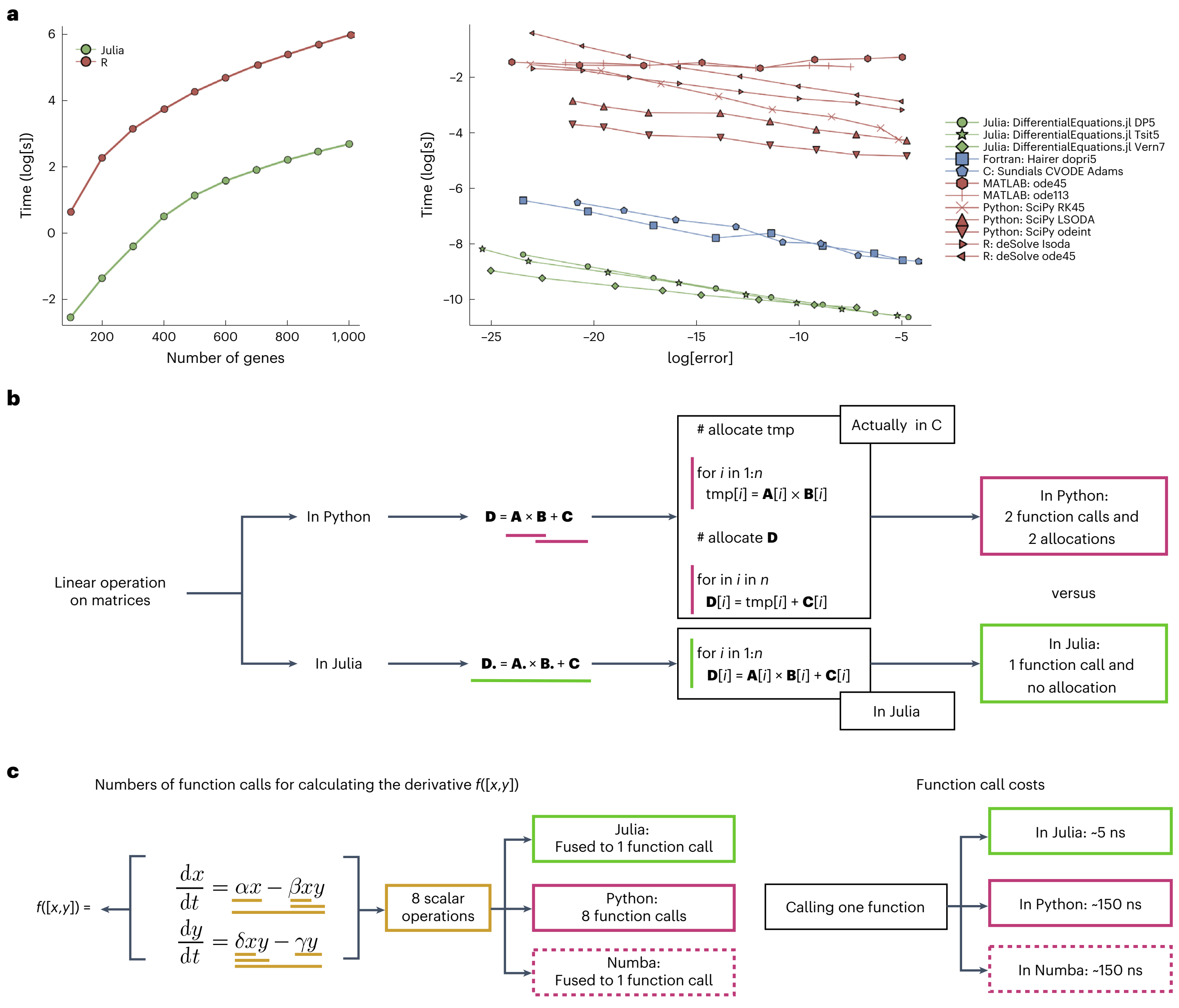
Fig. 2 | Overview of Julia's package ecosystem, presented by topic group. Julia consists of packages related to five main biological topics: bioinformatics, mathematical modeling, statistical and machine learning, data handling and visualization, and the integration of non-Julia code.

https://github.com/ElisabethRoesch/Perspective_Julia_for_Biologists/tree/main/examples/Abstraction/Example_Structural_bioinformatics_with_composable_packages. First, the online material shows code for the examples discussed here. Code examples have been chosen and designed to be accessible to a wide audience. We group them based on computational focus (high- and low-level user case) and access points (for example, Julia files and interactive notebooks). Second, a summary of helpful resources for starting with Julia and building Julia solutions is provided. The latter include, for example, platform-specific Julia installation guides, links to introductory Julia courses and a selection of pointers to relevant Julia communities.

Speed

The speed of a programming language is not just a matter of convenience that allows us to complete analyses more quickly (Fig. 3). It can enable new and better science. Speed is important for analyzing large datasets[10] that are becoming the norm across many areas of modern biomedical research. Slow computations might not hold back scientific discovery when performed a small number of times. However, when performed repeatedly on large datasets, the execution speed of a programming language can become the limiting factor. Similarly, simulation of large and complex computational models is only possible with fast implementations. For example, digital twins[11,12] in precision medicine will be useless without fast computation.

The speed of the programming language also determines how extensively we can test statistical analysis or simulation algorithms before using them on real data. Thorough testing of a new statistical algorithm can be expected to be around two to three orders of magnitude more costly in computational terms than a single production run[13]. Furthermore, the quality of approximations depends on many factors (for example, the number of tested candidates[14,15] and grid step sizes[16]), and faster code enables better analysis. Here and in the Supplementary Information we provide insights into the design features underlying Julia's speed[6]. The speed rivals that of statically compiled languages such as Fortran and C/C++. Higher-level language features—hallmarks of R, Python, MATLAB and Julia—typically lead to



Theoretically inferred and real-time calculation of *f*([*x*,*y*])

	Time of array allocation	+	Time of floating point operations	+	Time of function calls	=	Inferred time	Real time
Julia			8 × 2 ns	+	1 × 5 ns	=	21 ns	20 ns
Python	300 ns	+	8 × 2 ns	+	8 × 150 ns	=	1,516 ns	1,510 ns
Numba	300 ns	+	8 × 2 ns	+	1 × 150 ns	=	466 ns	425 ns

Fig. 3 | Julia's speed feature. a, Speed-up examples relevant to biology. Left, comparison of the time required to calculate the mutual information for all possible pairs of genes of a single-cell dataset[13]. Right, benchmark of ODE solvers implemented in Julia, Fortran, C, MATLAB, Python and R for the Lotka–Volterra model (more systems are described in ref. 50). **b**, Schematic of the speed up of vectorizable code (as in **a**). **c**, Schematic of the speed up of nonvectorizable code (as in **b**).

shorter development times. Going from an initial idea to working code can be orders of magnitude faster than, for example, C/C++. This is in no small measure helped by the flexible Jupyter and Pluto.jl notebook user interfaces (which fulfill similar functions to, for example, R's Shiny) and flexible software editing environments. Julia combines fast development with fast run-time performance and is therefore appropriate for both algorithm/method prototyping and time- and resource-intensive applications.

Example: network inference from single-cell data

In single-cell biology, we can measure expression levels of tens of thousands of genes in tens of thousands of cells[17]. Increasingly, we are able to do this with spatial resolution. However, searching for patterns in complex and large datasets is computationally expensive. Even apparently simple tasks, such as calculating the mutual information across all pairs of genes in a large dataset can quickly become impossible.

Gene regulatory network inference from single-cell data is a statistically demanding task and one for which Julia's speed helps. Chan et al.[13] used higher-order information theoretical measures to infer gene regulatory networks from transcriptomic single-cell data of a range of developmental and stem cell systems. The mutual information has to be calculated for gene pairs, but a multivariate information measure—partial information decomposition—is also considered to separate out direct and indirect interactions[18], and this requires consideration of all gene triplets.

The run time of algorithms implemented in the Julia package InformationMeasures.jl can be compared with that of the minet R package[19] (Fig. 3a, left). For small numbers of genes, differences are considerable but not prohibitive. Inferring a network with 100 genes takes around 0.3 s in Julia compared with 1.5 s in R, but already for 1,000 genes the inference times differ substantially (17 s in Julia and 390 s (>20-fold difference) in R). For datasets with 3,500 genes and 600 cells (by today's standards, small datasets), R needs over 2.5 h compared with Julia's 134 s (~64-fold difference) and, in real-world applications, 400-fold speed differences are possible (this corresponds to computing times of hours versus weeks). Here we reach the threshold of what can be tested and evaluated rigorously in many high-level languages. Overall, multivariate information measures would almost certainly be unfeasible in pure R or Python implementations.

The reason for this performance difference is Julia's ability to optimize vectorizable code[6] (Fig. 3b). Users of Python and R are familiar with vectorized functions, such as maps and element-wise operations. Julia's performance improves by combining just-in-time compilation, whereby computer code is compiled at run time (and the compiler can therefore be informed by the current state of the program and data), rather than ahead of execution, using vectorized functions via a trick known as operator fusion. When writing a chain of vector expressions, such as $\mathbf{D} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ (where **A**, **B**, **C** and **D** are n-dimensional vectors), libraries such as NumPy call optimized code, which is typically written in languages such as C/C++, and these operations are computed sequentially (Fig. 3c). To evaluate $\mathbf{A} \times \mathbf{B}$, C code is called to produce a temporary array, tmp, then tmp + **C** is evaluated (using C) to produce **D**. Allocating memory for the temporary intermediate tmp and the final result **D** is $O(n)$ (which means that the time it takes to complete the computation increases approximately linearly with n, the length of the vectors) and scales proportionally to the compute cost; thus, no matter what the size of the vectors, there is a major unavoidable overhead. Julia uses the "**.**" ("dot") operator to signify element-wise action of a function, and we write $\mathbf{D} = \mathbf{A}.\times\, \mathbf{B}.+\, \mathbf{C}$. When the Julia compiler sees this so-called broadcast expression, indicated by the "**.**" operator, it fuses all nearby dot operations into a single function and just-in-time compilation compiles this function at run-time into a loop. Thus, NumPy makes two function calls and spends time generating two arrays, whereas Julia makes a single function call and reuses existing memory. This and similar performance features are now leading package authors of statistical and data science libraries to recommend calling Julia for such operations, such as the recommendation by the principal author of the R lme4 linear mixed effects library to use JuliaCall to access MixedModels.jl in Julia (both written by the same author) for an approximately 200× acceleration[20].

The code for this example can be found at https://github.com/ElisabethRoesch/Perspective_Julia_for_Biologists/tree/main/examples/Abstraction/Example_Structural_bioinformatics_with_composable_packages.

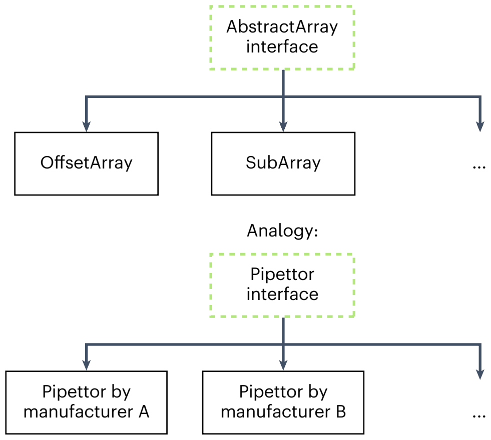


Fig. 4 | Interfaces in Julia. It is possible for experimental scientists to switch between different pipettors without recreating entire experimental protocols because a common understanding (or interface) exists that specifies tasks that pipettors should be able to perform in a similar manner. In Julia, we can define interfaces, such as the AbstractArray class, in which we specify rules that any array-like computational object has to follow. Interfaces allow us to apply methods developed for abstract types to custom types. By building our algorithms around interfaces, we can make the use, reuse and refinement of code easier.

Abstraction

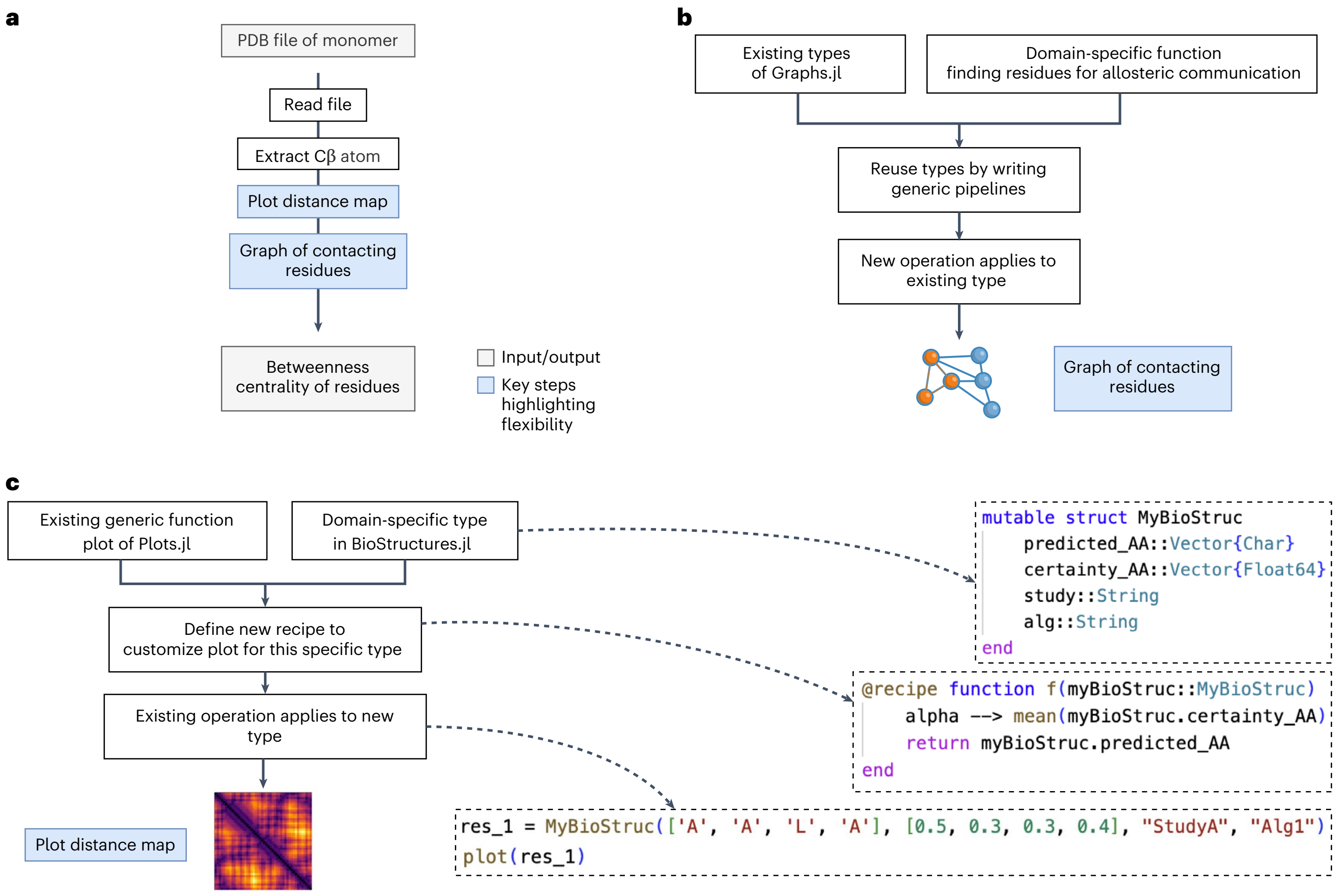
Julia allows an exceptionally high level of abstraction[21]. We can illustrate the advantages of abstraction by drawing an analogy to a standard laboratory tool: the pipettor. Pipettors produced by different manufacturers have slightly different designs. Nevertheless, they all perform the same task in a similar way. It thus takes minimum effort to get used to a new pipettor without having to retrain on every aspect of an experimental protocol. Abstraction achieves the same for software. Similar to the described abstract interface pipettor, in Julia we have interfaces such as the AbstractArray interface (Fig. 4 and discussed in detail in the Supplementary Information). All of its implementations are array-like structures that provide the same core functionalities that an array-like structure is expected to have. This allows us to easily and flexibly switch between different implementations of the same interface[22].

Abstraction is especially advantageous in the biological sciences where data are frequently heterogeneous and complex[23,24]. This can pose challenges for software developers[22] and data analysis pipelines, as changes to data may require substantial rewriting of code for processing and analysis. We may either end up with separate implementations of algorithms for different types of data or we may remove details and nuance from the data to enable analysis by existing algorithms. With abstraction, we do not have to make such choices. Julia's abstraction capabilities provide room for both specialization and generalization through features such as abstract interfaces and generic functions that can exploit the advantages of unique data formats with varying internal characteristics without an overall performance penalty. Here we illustrate the effect of Julia's abstraction via an example of a structural bioinformatics pipeline. Additionally, we provide a second, more technical abstraction example focusing on image analysis in Supplementary Fig. 1.

Example: structural bioinformatics with composable packages

Julia's flexibility means that packages from different authors can generally be combined with ease into workflows—a feature known as composability (Fig. 5). Users benefit from Julia's flexibility just as much as package developers. For example, we consider a standard structural bioinformatics workflow (Fig. 5a) in which we want to download

https://doi.org/10.1038/s41592-023-01832-z


a
PDB file of monomer
Read file
Extract Cβ atom
Plot distance map
Graph of contacting residues
Betweenness centrality of residues
Input/output
Key steps highlighting flexibility
b
Existing types of Graphs.jl
Domain-specific function finding residues for allosteric communication
Reuse types by writing generic pipelines
New operation applies to existing type
Graph of contacting residues
c
Existing generic function plot of Plots.jl
Domain-specific type in BioStructures.jl
Define new recipe to customize plot for this specific type
Existing operation applies to new type
Plot distance map

```julia
mutable struct MyBioStruc
    predicted_AA::Vector{Char}
    certainty_AA::Vector{Float64}
    study::String
    alg::String
end
```

```julia
@recipe function f(myBioStruc::MyBioStruc)
    alpha --> mean(myBioStruc.certainty_AA)
    return myBioStruc.predicted_AA
end
```

```julia
res_1 = MyBioStruc(['A', 'A', 'L', 'A'], [0.5, 0.3, 0.3, 0.4], "StudyA", "Alg1")
plot(res_1)
```

Fig. 5 | The abstraction feature in Julia. a, Abstract Julia code enables a flexible structural bioinformatics pipeline. The flow chart shows a pipeline that combines multiple Julia packages seamlessly together. This gives developers and users flexibility so that the effort and time required to generate new models and complex workflows is substantially reduced and collaboration is made easier. PDB, Protein Data Bank. **b**, An example pipeline showing the solving of the first part of the expression problem (an illustration of which is provided in Fig. 1) via the easy code base extension to new functions (step highlighted in blue). **c**, Left, an example pipeline showing the solving of another expression problem: extension to new types. The step highlighted in blue represents the point at which a new plot recipe is defined for a domain-specific type (that is, we demonstrate the extension of an existing code base to new types). Right, Julia code for defining a new type and and a new plot recipe. This example is for the structure MyBioStruc, which captures the results of prediction algorithms of amino acid sequences based on data. It is defined with the fields predicted_AA (a vector of characters that represent the predicted AAs), certainty_AA (a vector of numbers quantifying the certainty for each predicted AA), study (a string naming the respective study that the prediction is based on) and alg (a string naming the respective prediction algorithm). With the macro @recipe, we can specify how the function plot(…) should work for our newly specified example type. Here we define that this should create a line plot of the predicted amino acids with the mean of the certainty of the prediction shown by the opacity of the line, specified by the Plots.jl package as α. More details on the selected example code are provided in the Supplementary Information.

and read the structure of the protein crambin from the Protein Data Bank. This can be done using the BioStructures.jl package[25] from the BioJulia organization, which provides the essential bioinformatics infrastructure. Protein structures can be viewed using Bio3DView.jl, which uses the 3Dmol.js JavaScript library[26] as Julia can easily connect to packages from other languages. We can show the distance map of the Cβ atoms using Plots.jl. While Plots.jl is not aware of this custom type, a Plots.jl recipe makes this straightforward. BioSequences.jl provides custom data types of sequences and allows us to represent the protein sequence efficiently. With this, BioAlignments.jl can be used to align our sequences of interest. This suite of packages can be used to carry out single-cell, full-length total RNA sequencing analysis[27] quickly and with ease. A few lines of code in BioStructures.jl allow us to define the residue contact graph using Graphs.jl, giving access to optimized graph operations implemented in Graphs.jl for further analysis, such as calculating the betweenness centrality of the nodes. If coding and analysis are performed in Pluto.jl, then updating one section updates the whole workflow, which assists exploratory analysis (Fig. 5b).

Packages can be combined to meet the specific needs of each study; for example, to generate protein ensembles and predict allosteric sites[28] or to carry out information theoretical comparisons using the MIToS.jl package[29]. In this example, we have used at least five different packages together seamlessly. Plots.jl, BioAlignments.jl and Graphs.jl do not depend on or know about BioStructures.jl, but can still be used productively alongside it (Fig. 5c). Abstraction means that the improvements in any of these packages will benefit users of BioStructures.jl, despite the packages not being developed with protein structures in mind.

Package composability is common across the Julia ecosystem and is enabled by abstract interfaces supported by multiple dispatch (that is, the ability to define multiple versions of the same function with different argument types). Programmers can define standard functions such as addition and multiplication for their own types. Abstraction means that functions in unrelated packages often just work despite knowing nothing about the custom types. This is rarely seen in languages such as Python, R and C/C++, where the behavior of an object is tightly confined and combining classes and functions from different projects requires much more (of what is known as) boilerplate code.

For example, the Biopython project[30] has become a powerful package covering much of bioinformatics. However, extensions to Biopython objects are generally added to (an increasingly monolithic) Biopython, rather than to independent packages. This can lead to objects and algorithms that have the difficult task of fitting all use cases, including their dependencies, simultaneously[31]. In contrast, Julia's composability facilitates writing generic code that can be used beyond its intended application domain. Tables.jl, for example, provides a common interface for tabular data, allowing generic code for common tasks on tables. Currently, some 131 distinct packages draw on this common core for purposes far beyond the initially conceived application scope. This is an example that showcases how abstraction ensures the interoperability and longevity of code.

The code for this example can be found at https://github.com/ElisabethRoesch/Perspective_Julia_for_Biologists/tree/main/examples/Abstraction/Example_Structural_bioinformatics_with_composable_packages.

Metaprogramming

As our knowledge of the complexity of biological systems increases, so does our need to construct and analyze mathematical models of these systems (Fig. 6). Currently, most modeling studies in biology rely on programming languages that treat source code as static. Once written, it can be processed into loaded and executing code, but it is never changed while running. We can compare this linear control process with the central dogma of biology: source code (DNA) is transformed into loaded code (RNA) and executing code (protein). We now know that this process (DNA⟶RNA⟶protein) is not linear and unidirectional. RNA and proteins can alter how and when DNA is expressed. Programming languages that support metaprogramming break the linear flow of the computer program in a analogous manner (Fig. 6a). With metaprogramming, source code can be written that is processed into loaded and executing code and that can be modified during run time. This shifts our perception from static software to code as a dynamic instance when the program can modify aspects of itself during run time.

Metaprogramming originated in the LISP programming language in the early days of artificial intelligence research. It enables a form of reflection and learning by the software, but the ability of a program to modify computer code needs to be channeled very carefully. In Julia, this is done via a feature called hygenic macros[32]. These are flexible code templates, specified in the program, that can be manipulated at execution time. They are called hygenic because they prohibit accidentally using variable names (and thus memory locations) that are defined and used elsewhere. These macros can be used to generate repetitive code efficiently and effectively.

However, there are other uses that can enable new research, and this includes the development of mathematical models of biological systems. Unlike in physics, first principles (the conservation of energy, momentum and so on) offer little guidance as to how we should construct models of biological processes and systems. For these notoriously complicated biological systems, trial and error, coupled with biological domain expertise and state-of-the-art statistical model selection, is required[33]. Great manual effort is spent on the formulation of mathematical models, the exploration of their behavior and their adaptation in light of comparisons with data. Metaprogramming (or the abilities of introspection and reflection during run time[32]) and the ability to automate parts of the modeling process open up enormous scope for new approaches to modeling biological systems (Fig. 6b), including whole cells (see Supplementary Information).

Example: biochemical reaction networks

Mathematical models of biochemical reaction networks allow us to analyze biological processes and make sense of the bewilderingly complex systems underlying cellular function[34,35]. However, the specification of mathematical models is challenging and requires us to specify all of our assumptions explicitly. We then have to solve these models based on these assumptions. Analyzing a given reaction network can involve the solution, for example, of ordinary differential equations, delay differential equations, stochastic differential equations (SDEs) or discrete-time stochastic processes. To create instances of each of these models would—in languages such as C/C++ or Python—typically require the writing of different snippets of code for each modeling framework. In Julia, via metaprogamming, different models can be generated automatically from a single block of code. This simplifies workflows and makes them more efficient, but also removes the possibility of errors due to model inconsistencies.

For example, we can consider the ERK phosphorylation process shown in Fig. 6b[36]. Here ERK is doubly phosoporylated (by its cognisant kinase, MEK), upon which it can shuttle into the nucleus and initiate changes in gene expression. Its role and importance have made ERK a target of extensive further analysis, and modeling has helped to shed light on its function and role in cell fate decision-making systems[37]. This small system, albeit one of great importance and subtlety, forms building blocks for larger, more realistic biochemical reaction and signal transduction[38] models.

In Julia, using the package Catalyst.jl[39], this model can be written directly in terms of its reactions, with the corresponding rates. Source code is human readable and differs minimally from the conventional chemical reaction systems shown in Fig. 6c.

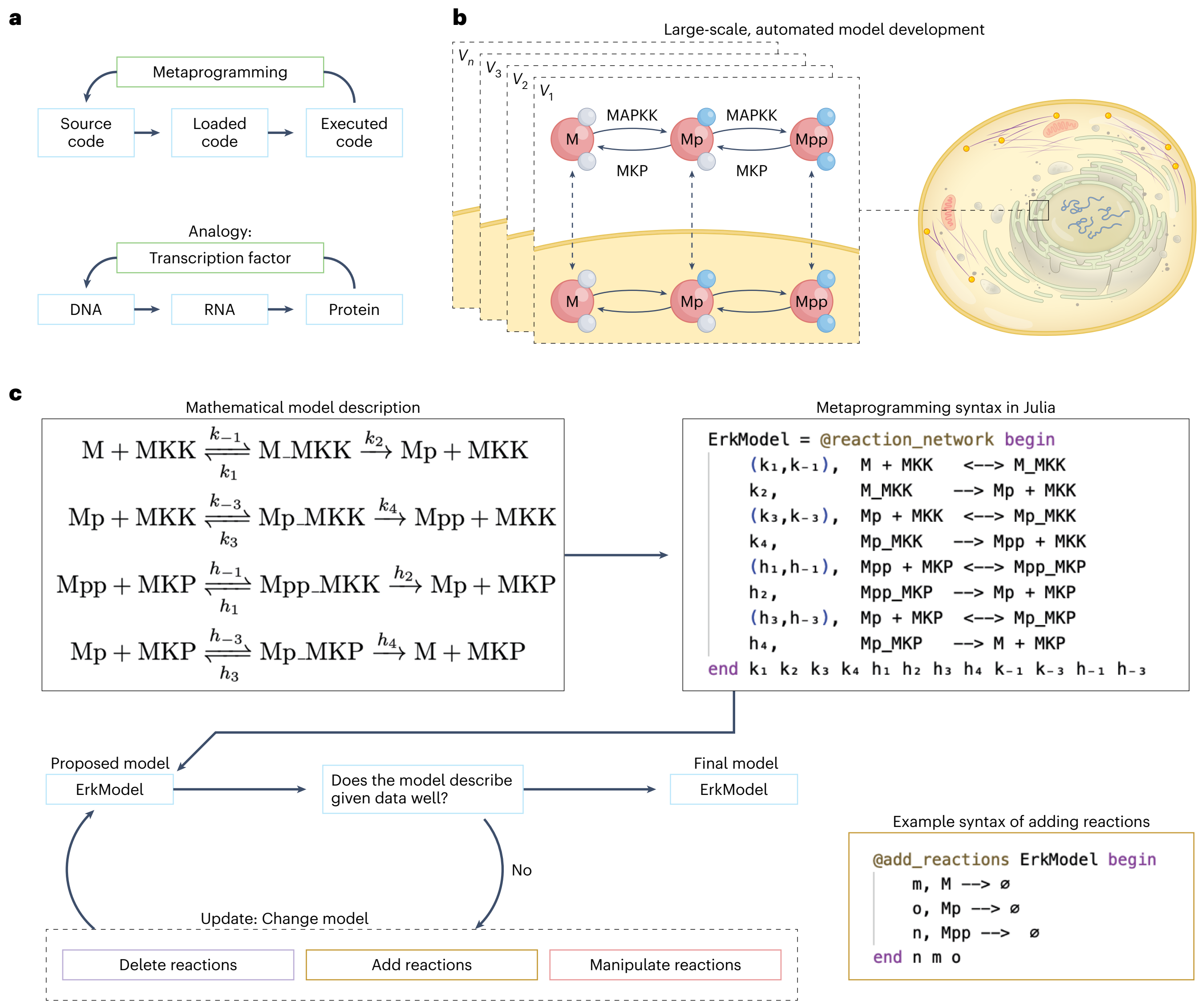
The science is encapsulated in this little snippet. Solving of the reaction systems then proceeds by calling the appropriate simulation tool from DifferentialEquations.jl. For a deterministic model, the reaction network is directly converted into a system of ordinary differential equations (via ODESystem). The same reaction network can be directly converted into a model that is specified by SDEs (via SDEProblem) or a discrete-time stochastic process model (via DiscreteProblem). Each of these cases leads to the creation of a distinct model that can be simulated or analyzed; yet, all of the models share the underlying structure of the same reaction network. To simulate one of the resulting models, the user needs to specify only the necessary assumptions required for a simulation (that is, the parameter values and initial conditions), as well as any further assumptions required that are specific to the model type (for example, the choice of noise model for a system of SDEs). Adapting the model to include nuclear shuttling[40] of ERK, as in Fig. 6c, or extrinsic noise upstream of ERK[36] is easily achieved using metaprogramming.

The fitting of models to data, or estimation of their parameters, is also supported by the Julia package ecosystem. Parameter estimation by evaluating the likelihood, the posterior distribution or a cost function is straightforward using the Optim.jl[41] or JuMP.jl[42] packages. Also, because of Julia's speed, it has become much easier to deploy Bayesian inference methods. Here, too, metaprogramming helps tools such as the probabilistic programming tool Turing.jl[43]. Approximate Bayesian computation approaches[44] also benefit from Julia's speed, abstraction and metaprogramming and are implemented in GpABC.jl[14].

The code for this example can be found at https://github.com/ElisabethRoesch/Perspective_Julia_for_Biologists/tree/main/examples/Metaprogramming/Example_Biochemical_reaction_networks.

Outlook

Computer languages, like human languages, are diverse and changing to meet new demands. When selecting a programming language, we have many choices, but often they reduce to essentially two options: using a widely used language that everybody else is using or using the best language for the problem. Traditional languages have an enviable track record of success in biological research. A frightening proportion of the Internet and modern information infrastructure probably depends on legacy software that would not pass modern quality control. However, it does the job, for the moment. Similarly, scientific



$$\mathrm{M} + \mathrm{MKK} \underset{k_1}{\overset{k_{-1}}{\rightleftharpoons}} \mathrm{M\_MKK} \xrightarrow{k_2} \mathrm{Mp} + \mathrm{MKK}$$

$$\mathrm{Mp} + \mathrm{MKK} \underset{k_3}{\overset{k_{-3}}{\rightleftharpoons}} \mathrm{Mp\_MKK} \xrightarrow{k_4} \mathrm{Mpp} + \mathrm{MKK}$$

$$\mathrm{Mpp} + \mathrm{MKP} \underset{h_1}{\overset{h_{-1}}{\rightleftharpoons}} \mathrm{Mpp\_MKK} \xrightarrow{h_2} \mathrm{Mp} + \mathrm{MKP}$$

$$\mathrm{Mp} + \mathrm{MKP} \underset{h_3}{\overset{h_{-3}}{\rightleftharpoons}} \mathrm{Mp\_MKP} \xrightarrow{h_4} \mathrm{M} + \mathrm{MKP}$$

```julia
ErkModel = @reaction_network begin
    (k₁,k₋₁),   M + MKK    <--> M_MKK
    k₂,         M_MKK      --> Mp + MKK
    (k₃,k₋₃),   Mp + MKK   <--> Mp_MKK
    k₄,         Mp_MKK     --> Mpp + MKK
    (h₁,h₋₁),   Mpp + MKP  <--> Mpp_MKP
    h₂,         Mpp_MKP    --> Mp + MKP
    (h₃,h₋₃),   Mp + MKP   <--> Mp_MKP
    h₄,         Mp_MKP     --> M + MKP
end k₁ k₂ k₃ k₄ h₁ h₂ h₃ h₄ k₋₁ k₋₃ h₋₁ h₋₃
```

```julia
@add_reactions ErkModel begin
    m, M --> ∅
    o, Mp --> ∅
    n, Mpp -->  ∅
end n m o
```

Fig. 6 | Julia's metaprogramming feature. a, Illustration of metaprogramming and an analogy to the central dogma of molecular biology. Similar to how a transcription factor, initially encoded in DNA, can control gene expression and modify RNA levels of an organism, with metaprogramming we can create code with a feedback effect. **b**, An example application of metaprogramming in biology. Metaprogramming is especially helpful for large-scale, automated model development. We can write code that adapts the model definition automatically (for example, in light of new data or based on how they interact with other submodels). For example, when constructing models of cellular systems V_1, V_2, ..., V_n, we can combine structurally similar models for the different MAP kinases present in human cells and build compartmental models by explicitly modeling the kinase dynamics in the nucleus and cytosol[40]. **c**, Example workflow of model construction. The adaption process of models could, for example, start with a theoretical inferred mathematical description, captured via the @reaction_network syntax of the Julia package Catalyst.jl. Subsequently, given experimental data, we evaluate an objective function of the current model, capturing the descriptiveness of the model in light of the data. Depending on the outcome of this evaluation, the model will be updated (for example, by adding new reactions to the model via the macro @add_reactions). More details on the selected example code are provided in the Supplementary Information.

progress is possible with legacy software. Python and R are far from legacy and have plenty of life in them, and there are tools that allow us to overcome their intrinsic slowness[45].

Here we have tried to explain why we consider Julia a language for the next chapter in the quantitative and computational life sciences. Julia was designed to meet the current and future demands of scientific and data-intensive computing[46]. It is an unequivocally modern language and it does not have the ballast of a long track record going all the way to the pre-big data days. The deliberate choices made by the developers furthermore make it fast and give developers and users of the language a level of flexibility that is difficult to achieve in other common languages such as R and Python, but also C/C++ and Fortran. On top of all of that, is a state-of-the-art package manager. All packages and Julia itself are maintained via Git, which makes installing and updating the Julia language, packages and their dependencies straightforward[6].

Julia has a smaller user base than R and Python, but it is growing. In some domains these languages have truly impressive package ecosystems. R and the associated Bioconductor project, in particular, have been instrumental in bringing sophisticated bioinformatics, data analysis and visualization methods to biologists. For many, they have also served as a gateway into programming. In other application areas (notably, the simulation of dynamical systems), Julia has leapfrogged the competition[47]. Many of the speed advantages of Julia come from just-in-time compilation, which underlies and enables good run-time performance. This, however, takes time and causes what is known as latency. Latency can be a problem for applications with hard real-time constraints, such as being the embedded code on a medical device that requires strict accurate updates at 100-ms intervals.

Julia was designed to meet the current and future demands of scientific and data-intensive computing. The Julia alternative that arguably has the most traction is Rust. Rust is an emerging language that has syntactic similarity to C++ but is better at managing memory safely. It detects discrepancies of type assignments at compile time and not just at run time, as is the case for C/C++. For this reason, it is being used in, for example, the Linux kernel. In the biological domain, it could become a choice for medical devices (as we can control latency) or bioinformatics servers that would previously have been developed in Java or C/C++.

These advantages of a new language need to be balanced against the convenience of programmers who are able to tap into the collective knowledge of vast user communities. All languages have started small and had to develop user bases. The Julia community is growing, including in the biomedical sciences, and it appears to be acutely aware of the needs of newcomers to Julia (and under-represented minorities in the computational sciences more generally[48]; see, for example, https://julialang.org/diversity/ for details), which makes the switch to Julia easier[9].

We have described the three main language design features that make Julia interesting for the scientific computing: speed, abstraction and metaprogramming. We have provided some intuition that fills these concepts with life, and we have illustrated how they can be exploited in different biological domains, and how speed, abstraction and metaprogramming together enable new ways of performing biological research. Even though we have introduced these features separately, they are deeply intertwined. For example, a lot of the speed-up opportunities of Julia derive from the language's abstraction powers; abstraction in turn makes metaprogramming easier.

References

1. Tomlin, C. J. & Axelrod, J. D. Biology by numbers: mathematical modelling in developmental biology. *Nat. Rev. Genet.* **8**, 331–340 (2007).
2. Auton, A. et al. A global reference for human genetic variation. *Nature* **526**, 68–74 (2015).
3. Robson, B. Computers and viral diseases. preliminary bioinformatics studies on the design of a synthetic vaccine and a preventative peptidomimetic antagonist against the SARS-CoV-2 (2019-nCoV, COVID-19) coronavirus. *Comput. Biol. Med.* **119**, 103670 (2020).
4. Seefeld, K. & Linder, E. *Statistics Using R with Biological Examples* (K. Seefeld, 2007).
5. Ekmekci, B., McAnany, C. E. & Mura, C. An introduction to programming for bioscientists: a Python-based primer. *PLoS Comput. Biol.* **12**, e1004867 (2016).
6. Sengupta, A. & Edelman, A. *Julia High Performance* (Packt Publishing, 2019).
7. Nazarathy, Y. & Klok, H. *Statistics with Julia: Fundamentals for Data Science, Machine Learning and Artificial Intelligence* (Springer, 2021).
8. Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. Julia: a fresh approach to numerical computing. *SIAM Rev.* **59**, 65–98 (2017).
9. Lauwens, B. & Downey, A. *Think Julia: How to Think like a Computer Scientist* (O'Reilly Media, 2021).
10. Marx, V. The big challenges of big data. *Nature* **498**, 255–260 (2013).
11. Björnsson, B. et al. Digital twins to personalize medicine. *Genome Med.* **12**, 4 (2019).
12. Laubenbacher, R., Sluka, J. P. & Glazier, J. A. Using digital twins in viral infection. *Science* **371**, 1105–1106 (2021).
13. Chan, T. E., Stumpf, M. P. & Babtie, A. C. Gene regulatory network inference from single-cell data using multivariate information measures. *Cell Syst.* **5**, 251–267.e3 (2017).
14. Tankhilevich, E. et al. GpABC: a Julia package for approximate Bayesian computation with Gaussian process emulation. *Bioinformatics* **36**, 3286–3287 (2020).
15. Innes, M. Flux: elegant machine learning with Julia. *J. Open Source Softw.* **3**, 602 (2018).
16. Rackauckas, C. & Nie, Q. DifferentialEquations.jl—a performant and feature-rich ecosystem for solving differential equations in Julia. *J. Open Res. Softw.* **5**, 15 (2017).
17. Chen, J. et al. Spatial transcriptomic analysis of cryosectioned tissue samples with Geo-seq. *Nat. Protoc.* **12**, 566–580 (2017).
18. Mahon, S. S. M. et al. Information theory and signal transduction systems: from molecular information processing to network inference. *Semin. Cell Dev. Biol.* **35**, 98–108 (2014).
19. Meyer, P. E., Lafitte, F. & Bontempi, G. minet: a R/Bioconductor package for inferring large transcriptional networks using mutual information. *BMC Bioinformatics* **9**, 461 (2008).
20. Bates, D. Julia MixedModels from R. https://rpubs.com/dmbates/377897 (2018).
21. Lange, K. *Algorithms from the Book* (SIAM, 2020).
22. Oliveira, S. & Stewart, D. E. *Writing Scientific Software: a Guide to Good Style* (Cambridge Univ. Press, 2006).
23. Alyass, A., Turcotte, M. & Meyre, D. From big data analysis to personalized medicine for all: challenges and opportunities. *BMC Med. Genom.* **8**, 33 (2015).
24. Gomez-Cabrero, D. et al. Data integration in the era of omics: current and future challenges. *BMC Syst. Biol.* **8**, I1 (2014).
25. Greener, J. G., Selvaraj, J. & Ward, B. J. BioStructures.jl: read, write and manipulate macromolecular structures in julia. *Bioinformatics* **36**, 4206–4207 (2020).
26. Rego, N. & Koes, D. 3Dmol.js: molecular visualization with WebGL. *Bioinformatics* **31**, 1322–1324 (2014).
27. Hayashi, T. et al. Single-cell full-length total RNA sequencing uncovers dynamics of recursive splicing and enhancer RNAs. *Nat. Commun.* **9**, 619 (2018).
28. Greener, J. G., Filippis, I. & Sternberg, M. J. Predicting protein dynamics and allostery using multi-protein atomic distance constraints. *Structure* **25**, 546–558 (2017).
29. Zea, D. J., Anfossi, D., Nielsen, M. & Marino-Buslje, C. MIToS.jl: mutual information tools for protein sequence analysis in the Julia language. *Bioinformatics* **33**, 564–565 (2017).
30. Cock, P. J. A. et al. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* **25**, 1422–1423 (2009).
31. Kunzmann, P. & Hamacher, K. Biotite: a unifying open source computational biology framework in Python. *BMC Bioinformatics* **19**, 346 (2018).
32. Perera, R. Programming languages for interactive computing. *Electron. Notes Theor. Comput. Sci.* **203**, 35–52 (2008).
33. Kirk, P. D. W., Babtie, A. C. & Stumpf, M. P. H. Systems biology (un) certainties. *Science* **350**, 386–388 (2015).
34. Kirk, P., Thorne, T. & Stumpf, M. P. Model selection in systems and synthetic biology. *Curr. Opin. Biotechnol.* **24**, 767–774 (2013).
35. Warne, D. J., Baker, R. E. & Simpson, M. J. Simulation and inference algorithms for stochastic biochemical reaction networks: from basic concepts to state-of-the-art. *J. R. Soc. Interface* **16**, 20180943 (2019).
36. Filippi, S. et al. Robustness of MEK-ERK dynamics and origins of cell-to-cell variability in MAPK signaling. *Cell Rep.* **15**, 2524–2535 (2016).

37. Michailovici, I. et al. Nuclear to cytoplasmic shuttling of ERK promotes differentiation of muscle stem/progenitor cells. *Development* **141**, 2611–2620 (2014).
38. MacLean, A. L., Rosen, Z., Byrne, H. M. & Harrington, H. A. Parameter-free methods distinguish Wnt pathway models and guide design of experiments. *Proc. Natl Acad. Sci. USA* **112**, 2652–2657 (2015).
39. Loman, T. E. et al. Catalyst: fast biochemical modeling with Julia. Preprint at *bioRxiv* https://doi.org/10.1101/2022.07.30.502135 (2022).
40. Harrington, H. A., Feliu, E., Wiuf, C. & Stumpf, M. P. Cellular compartments cause multistability and allow cells to process more information. *Biophys. J.* **104**, 1824–1831 (2013).
41. Mogensen, P. K. & Riseth, A. N. Optim: a mathematical optimization package for Julia. *J. Open Source Softw.* **3**, 615 (2018).
42. Dunning, I., Huchette, J. & Lubin, M. JuMP: a modeling language for mathematical optimization. *SIAM Rev.* **59**, 295–320 (2017).
43. Ge, H., Xu, K. & Ghahramani, Z. Turing: a language for flexible probabilistic inference. In *Proc. 21st International Conference on Artificial Intelligence and Statistics* 1682–1690 (Proc. Machine Learning Res., 2018).
44. Liepe, J. et al. A framework for parameter estimation and model selection from experimental data in systems biology using approximate bayesian computation. *Nat. Protoc.* **9**, 439–456 (2014).
45. Harris, C. R. et al. Array programming with NumPy. *Nature* **585**, 357–362 (2020).
46. Stanitzki, M. & Strube, J. Performance of Julia for high energy physics analyses. *Comput. Softw. Big Sci.* **5**, 10 (2021).
47. Rackauckas, C. et al. Accelerated predictive healthcare analytics with Pumas, a high performance pharmaceutical modeling and simulation platform. Preprint at *bioRxiv* https://doi.org/10.1101/2020.11.28.402297 (2020).
48. Whitney, T. & Taylor, V. Increasing women and underrepresented minorities in computing: the landscape and what you can do. *Computer* **51**, 24–31 (2018).
49. Sharpe, J. Computer modeling in developmental biology: growing today, essential tomorrow. *Development* **144**, 4214–4225 (2017).
50. Rackauckas, C. Benchmark of ODE solvers in Julia. https://github.com/SciML/MATLABDiffEq.jl (2019).

Acknowledgements

We thank all attendees of the Birds of a Feather session Julia for Biologists at JuliaCon2021; D. F. Gleich for allowing us to run an experiment on his servers; and R. Patro for discussions about Rust. E.R. acknowledges financial support through a University of Melbourne PhD scholarship. A.L.M. acknowledges support from the National Science Foundation (DMS 2045327). T.E.H. acknowledges NIH 1UF1NS108176. The information, data and work presented herein was funded in part by the Advanced Research Projects Agency—Energy under award numbers DE-AR0001222 and DE-AR0001211, as well as National Science Foundation award number IIP-1938400. The views and opinions of the authors expressed herein do not necessarily state or reflect those of the US Government or any agency thereof. M.P.H.S. acknowledges funding from the University of Melbourne Driving Research Momentum initiative and Volkswagen Foundation Life? program grant (grant number 93063), as well as support through an Australian Research Council Laureate Fellowship.

Author contributions

E.R. and M.P.H.S. conceived of the concept of the project and were in charge of the overall direction and planning. All authors contributed to writing the manuscript and have read and approved the final version.

Competing interests

E.R. is a Sales Engineer at JuliaHub. C.R. is the Vice President of Modeling and Simulation at JuliaHub and Director of Scientific Research at Pumas-AI. T.E.H. is a steward of the Julia project. H.N. is a Senior Computer Scientist at RelationalAI. J.G.G., A.L.M. and M.P.H.S. declare no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41592-023-01832-z.

Correspondence should be addressed to Michael P. H. Stumpf.

Peer review information *Nature Methods* thanks Nico Stuurman and the other, anonymous, reviewers for their contribution to the peer review of this work. Primary Handling Editor: Rita Strack, in collaboration with the *Nature Methods* team.