Industrial Engineering Journal

ISSN: 0970-2555

Volume : 53, Issue 2, February : 2024

BIOLOGICAL SYSTEM ADMINISTRATIONS FOR COMPELLING UTILIZE OF INFORMATION DRIVEN MODELING

Nagaraj Rathod Assistant Professor, Department of Computer Science and Engineering St. Martin's Engineering College, Secunderabad, Telangana, India E-mail: nagarajrathodcse@smec.ac.in

Abstract

The principle point of this paper is the means by which viably Data Driven Modeling (DDM) can be adequately utilized for biological system administrations contrasting and the traditional displaying, the DDM (Data Driven Modeling) measure gives the best exactness. For going through preparing, tree order calculations were utilized like choice tree, packing, irregular woods with boosting angle like XGBoosting. The biological system dataset is contrasted here and all most appropriate calculations. In Random woods the way toward finding the root hub and parting the element hubs will run arbitrarily. The highlights assume a significant part in arbitrary backwoods calculation particularly tracking down the significant component for preparing the set. Over fitting is one basic issue that may aggravate the outcomes, yet for Random Forest calculation, if there are sufficient trees in the woodland, the classifier will not over fit the model particularly for arrangement issues. Irregular backwoods with XGBoost (eXtreme Gradient Boosting) which an incredible, and lightning quick AI library where the trees are developed successively and the speed is expanded by equal preparing. This information is prepared utilizing Random Forest in XGBoosting with extra hyper boundaries and the exactness is anticipated.

Keywords: Data pre-processing, XGBoosting algorithms, Random Forest algorithm, Predictive model.

Introduction

Mathematical modeling of cellular processes for mechanism exploration has now become commonplace using various techniques [1–5], but challenges remain as to how models should be built, calibrated, analyzed and interpreted to extract much- needed mechanistic knowledge from experimental data. Historically, methods and techniques from other fields have been directly imported to systems biology with varying success. For example, early interpretations of cellular processes as circuits provided insights about basic regulatory motifs that could explain cellular behaviors [6]. Similarly, techniques from chemistry, physics, and various engineering disciplines have been used to model cellular processes [7,8], but due to the spatiotemporal complexity of cellular processes, from femtosecond/nanometer electron transfer reactions to years and meter scales in tumor growth, no established paradigm has emerged to capture the full complexity of cellular processes. Multiple tools have been developed to achieve specific modeling tasks. For example, COPASI [5], RuleMonkey [9], Simmune [10], and StochSS [11] all provide graphical user interfaces that cater to non-expert modelers wishing to encode mechanistic representations of biological processes. More abstract approaches such as BioNetGen [12], Kappa [13], and CobraPy [14] employ a domain- specific language (DSL) to describe and simulate models. However, most tools are self-contained platforms with a small set of included methods and analyses, limiting access to other standalone simulation tools such as StochKit [4], SciML tools[15], URDME [16], SmolDyn [17]. Similarly, optimization techniques ranging from vector- based optimization methods [18,19] to probabilistic-based methods [20*,21] exist in yet another isolated domain. Therefore, the current modeling and simulation ecosystem is compartmentalized and fractured, and thus, unification and intercompatibility efforts are sorely needed.

Valuable efforts toward unification have been put forth to create standards for model instantiation, simulation, analysis and dissemination [22,23**,24–26]. Of these, Systems Biology Markup Language (SBML) is perhaps the most successful to date. However, mathematical modeling for cell biology remains challenging to scale - both vertically (larger, more complex models) and horizontally (more active collaborators). While mathematical tools are the obvious way forward to describe cellular processes, the complexity challenge results in a knowledge base that is highly domain specific, with some notable exceptions [27*].

A novel, more flexible approach to encode knowledge about biological processes as computer programs is slowly emerging and gaining momentum [3,28,29]. In this approach, biological models are no longer static documents, but computer code that aggregates community knowledge and opens doors toward crowd-driven mathematical models of biological processes. Although computer languages like Lisp [30] and proprietary packages such as MATLAB have been used toward this goal, we believe Python provides the largest ecosystem, myriad learning resources, and large applicable base to unify modeling practices in the field. Adopting a programmatic modeling paradigm for systems biology automatically accrues decades of computer science practices including structured documentation, integrated development environments (IDEs), (model) version control, code-sharing platforms, code testing frameworks, and importantly, literate programming/computational notebook dissemination. Here, we review the recent developments in programming-based approaches for systems biology. The structure of the manuscript is motivated by the model specification, simulation, calibration, analysis, and visualization paradigm/pipeline, commonly practiced in systems biology. Throughout, we note how this approach could be supplemented and improved by incorporating best practices from software engineering (Figure 1).

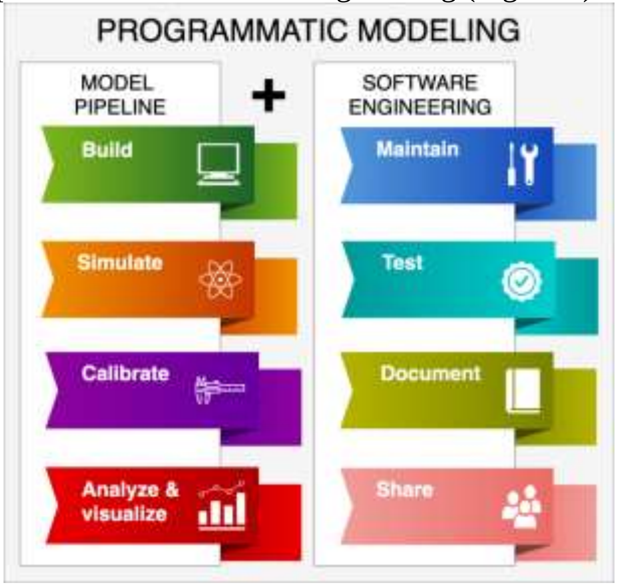

Figure 1: The traditional modeling paradigm in systems biology entails model building, simulation, calibration, and analysis (left column), which is carried out with myriad tools and practices. Software engineering practices can add much needed structure to the practice through maintenance, testing, documentation and sharing paradigms (right column), vetted by a the software community.

Table 1: List of key frameworks, tools, and services for programmatic modeling in Python. BSD, MIT, and PSF are permissive software licenses. GPL is a “copyleft” software license.
*Free for open-source projects.

Tool or service	**Usage terms**	**Notes**
Frameworks		
PySB [3]	BSD	Bespoke Python object-based model format, multiple simulation backends

Tellurium [29]	Apache	Bespoke DSL (Antimony), ODE and SSA simulationbackends
PySCeS [70]	BSD	Bespoke DSL, ODE simulation backend
ScrumPy [71]	GPL	Metabolic modeling
CobraPy [14]	GPL	Metabolic modeling
Testing		
PyTest	MIT	Testing framework; pytest.org
GitHub Actions	Free* service	Continuous Integration; github.com/features/actions
Circle CI	Free* service	Continuous Integration; circleci.com
Calibration		
PyBioNetFit [48*]	BSD	BNGL and SBML models
PyPESTO [49]	BSD	SBML and PEtab support
PyDREAM [20*]	GPL	PySB interface
Analysis & Visualization		
Matplotlib	PSF	Plotting library; matplotlib.org
Jupyter Notebooks	BSD	Computational notebooks; jupyter-notebook.readthedocs.io
PyVIPR [58*]	MIT	PySB, Tellurium interfaces
Sharing and modification		
Github	Free* service	Code hosting and collaboration suite; github.com

Model specification

Traditionally, encoding a model of biochemical reactions would require the user to write each equation by hand, encode these into a solver, and run the simulations [31]. Although this is still common practice for smaller model systems, these lists of equations often lead to a model dead-end as the biological context is completely lost in the mathematical representation, which hinders model reuse. Reaction-based modeling formats add one layer of abstraction where the user instead writes chemical reactions of the form $A + B \leftrightarrow C$ in a program-specific notation and the computer parses this information into a mathematical representation [32]. These DSLs can operate either through a GUI that generates the code in the background, or directly through a text editor. For example, Antimony [32] requires manual enumeration of every species and reaction explicitly. However, signaling pathways often comprise a large number of molecular complexes, which can assemble in multiple orders, leading to a large number of reactions and intermediate species during complex assembly and degradation. Therefore, traditional approaches become unwieldy as model systems become larger, learning to model dead-ends. Another level of abstraction is presented by rule-based modeling formalisms whereby reaction rules rather than explicit reactions (or equations) are used to encode the system [3,12,13]. A reaction rule is a template for reaction patterns to be enumerated and instantiated recursively, starting from a defined list of initial species, thereby saving the user time and reducing error- prone repetition. In rule-based approaches, the reaction center (the relevant molecular components for a given reaction) is separated from the context (attached molecular components which have minimal or no effect on the reaction). These approaches often require a pre-processing step to generate the network of nodes (chemical species) and edges (chemical reactions) from the initial pool of chemical species and a set of reaction rules. However, network-free methodologies have been proposed to bypass the network generation step [33].

Model specification can also be embedded into General Purpose Programming Languages (GPPL) to provide a more powerful approach to biological modeling. In the programmatic modeling paradigm, the model is encoded as an executable piece of code, thereby offering all the advantages of a full-fledged computer programming language (Figure 2). Modularity, in which a model can be split into smaller, reusable code objects, is perhaps the most useful aspect for cell biology modeling. For example, PySB currently includes a library of 25 macros (small modules or functions) that encode reaction patterns commonly found in biology such as catalytic activation, molecule-molecule inhibition, or complex oligomerization. From a user perspective, GPPLs have greater integration with IDEs than DSLs, thus allowing syntax highlighting and checking, and navigation between functions. The model is also inspectable at runtime, allowing searching and filtering of model components. For example, a user could check whether certain species or reactions are present before simulation commences. Currently, the most used modeling frameworks using the programmatic modeling paradigm in Python are PySB [3], written in and using Python, and Tellurium [29], which is written in Python but uses Antimony [32], a DSL with function support, for model specification.

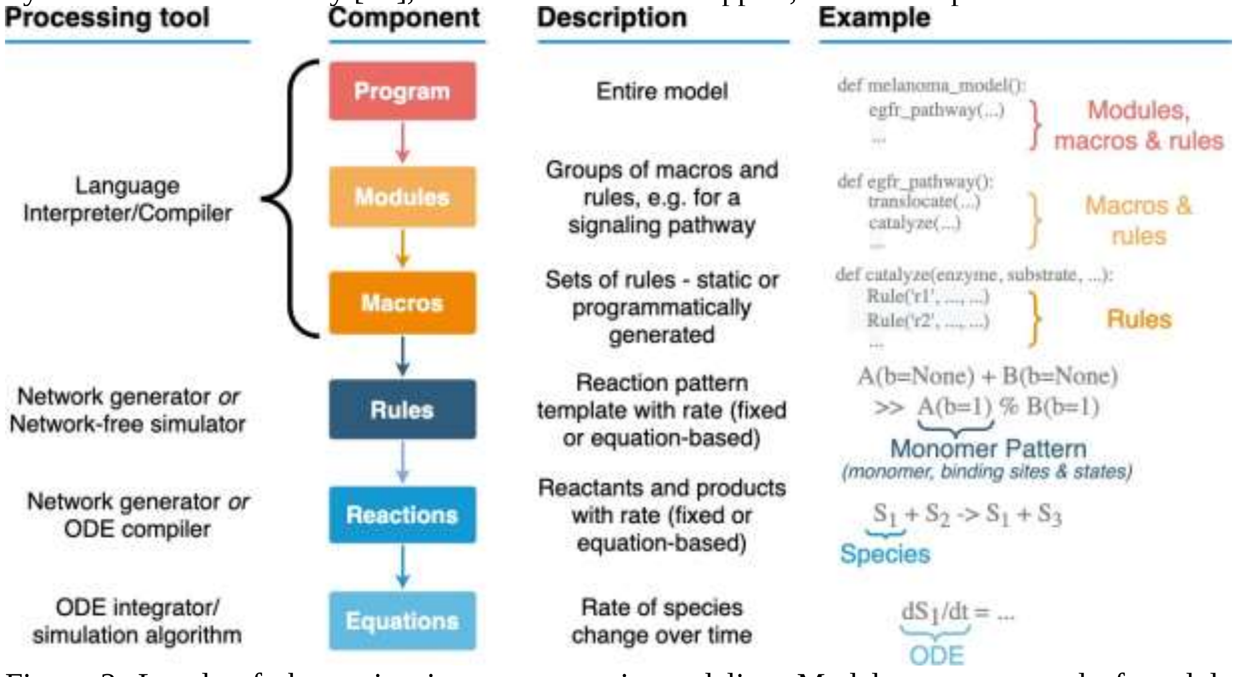
Processing tool
Component
Description
Example
Language Interpreter/Compiler
Program
Entire model
def melanoma_model():
egfr_pathway(...)
...
Modules, macros & rules
Modules
Groups of macros and rules, e.g. for a signaling pathway
def egfr_pathway():
translocate(...)
catalyze(...)
...
Macros & rules
Macros
Sets of rules - static or programmatically generated
def catalyze(enzyme, substrate, ...):
Rule('r1', ..., ...)
Rule('r2', ..., ...)
...
Rules
Network generator or Network-free simulator
Rules
Reaction pattern template with rate (fixed or equation-based)
A(b=None) + B(b=None) >> A(b=1) % B(b=1)
Monomer Pattern
(monomer, binding sites & states)
Network generator or ODE compiler
Reactions
Reactants and products with rate (fixed or equation-based)
S1 + S2 -> S1 + S3
Species
ODE integrator/ simulation algorithm
Equations
Rate of species change over time
dS1/dt = ...
ODE

Figure 2: Levels of abstraction in programmatic modeling. Models are composed of modules and macros, which are handled by the programming language interpreter/compiler; rules encode sets of reactions using structured pattern templates; reactions specify biochemical species' transformations; and finally equations are handled by an ODE integrator or simulation algorithm directly.

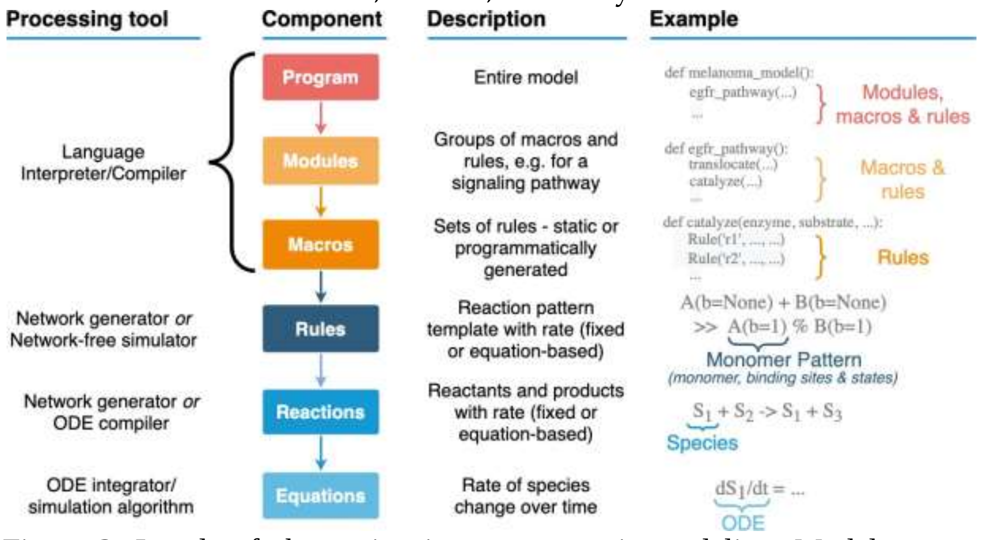
Processing tool
Component
Description
Example
Language Interpreter/Compiler
Program
Entire model
def melanoma_model():
egfr_pathway(...)
...
Modules, macros & rules
Modules
Groups of macros and rules, e.g. for a signaling pathway
def egfr_pathway():
translocate(...)
catalyze(...)
...
Macros & rules
Macros
Sets of rules - static or programmatically generated
def catalyze(enzyme, substrate, ...):
Rule('r1', ..., ...)
Rule('r2', ..., ...)
...
Rules
Network generator or Network-free simulator
Rules
Reaction pattern template with rate (fixed or equation-based)
A(b=None) + B(b=None) >> A(b=1) % B(b=1)
Monomer Pattern
(monomer, binding sites & states)
Network generator or ODE compiler
Reactions
Reactants and products with rate (fixed or equation-based)
S1 + S2 -> S1 + S3
Species
ODE integrator/ simulation algorithm
Equations
Rate of species change over time
dS1/dt = ...
ODE

Figure 2: Levels of abstraction in programmatic modeling. Models are composed of modules and macros, which are handled by the programming language interpreter/compiler; rules encode sets of reactions using structured pattern templates; reactions specify biochemical species' transformations; and finally equations are handled by an ODE integrator or simulation algorithm directly.

Model simulation

Model simulation involves numerically solving the model equations to obtain trajectories for dynamically controlled species. Concentrations or molecule counts of chemical species in the model are the most commonly simulated quantities. Integration of systems of ordinary differential equations (ODEs) for deterministic simulations is the most common model simulation approach. Many ODE integrators are available and the best choice depends on model stiffness, desired integrator tolerances, and other requirements. In Python, a family of integrators is available through SciPy [34**] including VODE and LSODA, but many other solvers have been proposed. Other commonly used solver suites include StochKit (Stochastic Simulation Algorithm) [4,35], BioNetGen (CVODE, SSA, tau-leaping algorithm, partition-leaping algorithm) [12], cupSODA (GPU ODE) [36], GPU_SSA (GPU SSA) [37], and Libroadrunner (CVODE,

SSA) [38]. Within the Python ecosystem, PySB provides a simulation class that enables users to use many of these simulation tools or to connect new tools as needed. In addition, users of other Python-based tools such as Tellurium can also take advantage of these resources.

Model calibration

Model calibration is the process of adjusting model parameters to match experimental data, also known as parameter estimation/optimization when applied to parametric models. The most common form of model calibration involves a process of running many simulations (thousands to millions or more) and checking the distance between model and experimental data error using an objective function, which gives a measure of the model's simulation "error" versus experiment; for a review see [39]. Since dynamic data for signaling models are hard to come by, the modeler often only has data for a few species, and thus model calibration often leaves a model underdetermined - multiple parameter sets fit the data equally well [40]. The concept of parameter "sloppiness" states that only a few "stiff" combinations of parameters are important in determining model outcomes, and others are "sloppy" and have little effect. Thus, an undetermined model can still be useful in predicting biological properties [41]. However, the interpretation of large, underdetermined models in the context of limited data is still up for debate. Lessons from e.g. hydrology and climate modeling have been highly influential toward addressing these issues [20*,42,43].

The landscape of model parameters is often envisioned as a multidimensional surface with "height" representing the objective function, where the (ideally global) minimum or minima (representing the best fit(s)) must be found. SciPy [34**], for example, includes gradient descent and simplex-based methods. However, the curse of dimensionality means that local optimization can give far-from-globally optimal results as the number of model parameters increases. Finding the global minimum of a multivariate nonlinear model is NP-hard [44], however several methods can make statistically good approximations. Markov Chain Monte Carlo sampling methods are among the most popular algorithms [45]. General purpose optimization toolkits for Python include SciPy.optimize [34**] and Pyomo [46]. We have found that DEAP [19] provides excellent support for PSO and genetic algorithm-based optimization.

Given the dearth of data available for biological model calibration, conditional probability (Bayesian) approaches are gaining traction. These approaches provide a probabilistic interpretation of model parameters [47], including uncertainty quantification, at the cost of increased computer time. However, new GPU-based integrators mitigate this problem. Excellent tools for Bayesian parameter inference include PyDREAM (which can readily take PySB models) [20*], PyBioNetFit [48*], PyPESTO [49], PyMC3 [50], and PySTAN [51], although popular data-science tools such as TensorFlow [52] and PyTorch [53] also provide Bayesian inference capabilities. ABC-SysBio [54] provides a hybrid solution to the computation problem but still within a Bayesian context.

Model analysis and visualization

Model analysis and visualization is likely the least developed area in systems biology as no clear standards have been proposed. In general, modelers explore the chemical species concentration trajectories in their model to infer mechanistic behaviors and properties. Exploration of biochemical flux through reactions is highly challenging with some notable attempts toward this goal in the literature [7,47], but much work is still needed. For visualization, perhaps the most useful tool in Python is matplotlib [55], which provides flexible graphing capabilities. Other Python tools include Seaborn (https://seaborn.pydata.org/), Plotly [56], and Mayavi [57]. Network visualization is perhaps the other major area of model analysis that is addressed in various ways in Python. For example, PyVIPR [58*] is a visualization tool built on Cytoscape.js [59] for rule- and reaction-based models which animates model dynamics over time, overlaid on a graph. MASSPy [60] also provides some visualization capabilities for metabolic models. We note, however, that excellent tools for graph manipulation in Python exist, such as NetworkX [61].

Model sharing and modification

Perhaps the most appealing benefit for the systems biology community from program- based paradigm is the use of literate programming for model and results dissemination. Introduced by Donald Knuth, literate programming is a paradigm whereby the code and the document coexist in an interactive format [62]. Jupyter Notebook, a popular format, has been described as "data scientists' computational notebook of choice" [63]. Jupyter Notebooks allow analyses to be run in a web browser, checked into version control, and include documentation alongside analyses, in turn improving transparency and reproducibility. We believe that Jupyter notebooks are a highly desirable step forward in systems biology as it greatly contributes to model transparency, revision, and dissemination, and should be included in paper submissions where computational simulation and analysis are involved.

Programmatic models' code can be managed using existing version control tools. Git has emerged as the de facto standard for version control, providing powerful capabilities for decentralized editing, branching, and merging, with online platforms such as GitHub adding a collaborative interface for change management, commenting, and other functions. In PySB, models are Python programs, and so can be imported like other Python modules and extended or modified. The code can be inspected, for example the model can be searched for species or reactions using pattern matching. Tellurium's antimony language has an import function, but previous model definitions are currently not programmatically searchable or modifiable.

Good documentation can be vital to ensure model reproducibility and interpretability by others. Sphinx (sphinx-doc.org) is a de facto documentation standard for Python code, which allows code comments to be compiled into multiple formats including website (HTML) and PDF. The former can be combined with continuous integration, for always up-to-date documentation (readthedocs.io).

Model checking and testing

Complex biochemical models present challenges in both ensuring they are correctly encoded, and ensuring their dynamics meet a given specification. In software engineering, it has become common practice to build an accompanying test suite while developing code, which runs the code under scrutiny to test that works as expected.

Subtle errors can be introduced as models grow larger. In our opinion, the field should establish minimum standards to ensure software is runnable, reproducible, and meets basic quality standards [64]. In the context of models-as-programs, unit and integration tests can be borrowed from software engineering to ensure code correctness. Unit tests refers to code which checks the functionality of other, minimal units of code; integration tests check that units work as expected when combined. Python has several frameworks for testing, PyTests is a popular option with a plugin for Jupyter Notebooks [65]. PySB introduces a framework for testing static properties of rule- based models after network generation; for example, checking that certain species are produced by the reaction network, or that certain reactions are present. Using continuous integration (CI), these tests can be run automatically when changes are made and checked into version control, and/or on a regular basis. Running tests regularly is recommended because, even if a model itself does not change, changes to software dependencies could lead to unexpected errors. The importance of this is emphasized by a recent review, which found a majority of Jupyter Notebooks were not automatically reproducible, often due to dependency errors [66**]. For open-source models, these tests can be run for free using services such as Github Actions, Travis, and Circle CI. Finally, we recommend containerization technologies such as Docker
[67] and Singularity [68], which bundle model and software dependencies together in a self-contained environment, further aiding reproducibility and cross-platform compatibility.

Conclusions

Python has recently turned 30 years old and is now one of the most popular programming languages in the world. There are many reasons for its success, but a key insight of its creator is that code is read much more often than it's written [69]. The same principle applies to models, which emphasizes the importance of clear documentation, transparency of approach, and the separation of model specification from simulation and downstream analysis code. These efforts are central to improving reproducibility, code maintenance, and model extensions, by original authors and third parties.

For beginners interested in modeling cell signaling, we recommend either the PySB or Tellurium frameworks, both of which have high quality documentation and active communities for support. We expect the Python modeling ecosystem will continue to grow, and efforts for framework and package interoperability to increase.

References

1. Albert R, Thakar J: Boolean modeling: a logic-based dynamic approach for understanding signaling and regulatory networks and for making useful predictions. Wiley Interdiscip Rev Syst Biol Med 2014, 6:353–369.

2. Ghaffarizadeh A, Heiland R, Friedman SH, Mumenthaler SM, Macklin P: PhysiCell: An open source physics-based cell simulator for 3-D multicellular systems. PLOS Comput Biol 2018, 14:e1005991.
3. Lopez CF, Muhlich JL, Bachman JA, Sorger PK: Programming biological models in Python using PySB. Mol Syst Biol 2013, 9.
4. Sanft KR, Wu S, Roh M, Fu J, Lim RK, Petzold LR: StochKit2: software for discrete stochastic simulation of biochemical systems with events. Bioinformatics 2011, 27:2457–2458.
5. Hoops S, Sahle S, Gauges R, Lee C, Pahle J, Simus N, Singhal M, Xu L, Mendes P, Kummer U: COPASI—a COmplex PAthway SImulator. Bioinformatics 2006, 22:3067– 3074.
6. Tyson JJ: Modeling the cell division cycle: cdc2 and cyclin interactions. Proc Natl Acad Sci 1991, 88:7328–7332.
7. Mallela A, Nariya MK, Deeds EJ: Crosstalk and ultrasensitivity in protein degradation pathways. PLOS Comput Biol 2020, 16:e1008492.
8. Lander AD, Nie Q, Sanchez-Tapia C, Simonyan A, Wan FYM: Regulatory feedback on receptor and non-receptor synthesis for robust signaling. Dev Dyn Off Publ Am Assoc Anat 2020, 249:383–409.
9. Colvin J, Monine MI, Gutenkunst RN, Hlavacek WS, Von Hoff DD, Posner RG: RuleMonkey: software for stochastic simulation of rule-based models. BMC Bioinformatics 2010, 11:404.
10. Angermann BR, Klauschen F, Garcia AD, Prustel T, Zhang F, Germain RN, Meier-Schellersheim M: Computational modeling of cellular signaling processes embedded into dynamic spatial contexts. Nat Methods 2012, 9:283–289.
11. Drawert B, Hellander A, Bales B, Banerjee D, Bellesia G, Jr BJD, Douglas G, Gu M, Gupta A, Hellander S, et al.: Stochastic Simulation Service: Bridging the Gap between the Computational Expert and the Biologist. PLOS Comput Biol 2016, 12:e1005220.
12. Harris LA, Hogg JS, Tapia J-J, Sekar JAP, Gupta S, Korsunsky I, Arora A, Barua D, Sheehan RP, Faeder JR: BioNetGen 2.2: advances in rule-based modeling. Bioinformatics 2016, 32:3366–3368.
13. Boutillier P, Maasha M, Li X, Medina-Abarca HF, Krivine J, Feret J, Cristescu I, Forbes AG, Fontana W: The Kappa platform for rule-based modeling. Bioinformatics 2018, 34:i583– i592.
14. Ebrahim A, Lerman JA, Palsson BO, Hyduke DR: COBRApy: COnstraints-Based Reconstruction and Analysis for Python. BMC Syst Biol 2013, 7:74.
15. Rackauckas C, Nie Q: DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. J Open Res Softw 2017, 5:15.
16. Drawert B, Trogdon M, Toor S, Petzold L, Hellander A: MOLNs: A CLOUD PLATFORM FOR INTERACTIVE, REPRODUCIBLE, AND SCALABLE SPATIAL STOCHASTIC COMPUTATIONAL EXPERIMENTS IN SYSTEMS BIOLOGY USING PyURDME. SIAM J Sci Comput Publ Soc Ind Appl Math 2016, 38:C179–C202.
17. Andrews SS: Smoldyn: particle-based simulation with rule-based modeling, improved molecular interaction and a library interface. Bioinforma Oxf Engl 2017, 33:710–717.
18. Kennedy J, Eberhart R: Particle swarm optimization. In Proceedings of ICNN'95 - International Conference on Neural Networks. . 1995:1942–1948 vol.4.
19. Fortin F-A, De Rainville F-M, Gardner M-AG, Parizeau M, Gagné C: DEAP: evolutionary algorithms made easy. J Mach Learn Res 2012, 13:2171–2175.
20. * Shockley EM, Vrugt JA, Lopez CF: PyDREAM: high-dimensional parameter inference for biological models in python. Bioinformatics 2018, 34:695–697.