Reproducible, Robust, and Reliable Biochemical Reaction Network Models for Systems Biology

Kiri Choi

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Herbert M. Sauro, Chair

James Carothers

Georg Seelig

Program Authorized to Offer Degree:
Bioengineering

University of Washington

Abstract

Reproducible, Robust, and Reliable
Biochemical Reaction Network Models for Systems Biology

Kiri Choi

Chair of the Supervisory Committee:
Professor Herbert M. Sauro
Department of Bioengineering

Reproducibility, robustness, and reliability are features desired for a biochemical reaction network model. Scientific research is reproducible when the findings can be independently verified and reproducibility is crucial for the integrity of science. Unfortunately, however, scientific studies, including computational studies, are often not reproducible. It is hard to achieve robustness and reliability due to technical difficulties with experiments producing high quantity, high-quality data and inherent unidentifiability arising from multiparametric nature of biological processes. Robustness and reliability can be achieved by obtaining more data and by implementing better computational algorithms. To improve the reproducibility of biochemical reaction network models, software tools are necessary. We need novel algorithms to increase the robustness and reliability of models. Most of all, a scalable and extensible computing environment is necessary for incorporating tools and algorithms. Therefore, we build a Python-based modeling and simulation environment called Tellurium to ensure reproducibility of studies while supporting a wide array of tools to help design robust and reliable models. Tellurium is specifically designed for high-throughput studies which are necessary to deploy novel modeling algorithms. Next, software tools to improve the reproducibility of computational studies have been built and integrated into Tellurium. In particular, Python support for standards related to describing simulation experiments

has been improved. Lastly, two algorithms to help construct robust and reliable mechanistic models have been designed. The algorithms actively explore the concepts of ensemble modeling and specifically utilize the data from perturbation studies. It is demonstrated that a model ensemble can provide reasonable predictions on the system of interest. The idea of the model ensemble directing future experiments toward maximal reduction of the potential topology of models in the ensemble is also explored. Once deployed, ensemble-based experiment selection is expected to close the cycle between modeling and experimental endeavors, bridging the disparity between data-driven modeling and modeling-driven data collection.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

BI-BI: a reaction type of $A + B \rightarrow C + D$.

BI-UNI: a reaction type of $A + B \rightarrow C$.

COMBINE ARCHIVE: a standard to encapsulate all files for a simulation study into a single archive.

COMPUTING ENVIRONMENT: a collection of software tools, platforms, languages, etc. to facilitate the development and the execution of computer codes.

KISAO: The Kinetic Simulation Algorithm Ontology. An ontology of simulation algorithms for systems biology models.

PACKAGE: an archive of software tools configured to be installed, upgraded, and removed by a package manager.

SBML: The Systems Biology Markup Language. A standard to describe models of biochemical processes.

SBOL: The Synthetic Biology Open Language. A standard to describe designs for synthetic organisms.

SED-ML: The Simulation Experiment Description Markup Language. A standard to describe simulation experiments.

UNI-BI: a reaction type of $A \rightarrow B + C$.

UNI-UNI: a reaction type of $A \rightarrow B$.

ACKNOWLEDGMENTS

First, I would like to express my deepest gratitude towards my supervisor Professor Herbert M. Sauro for all his support and guidance throughout this work. He is the reason why I am here. Through his guidance I have been able to get mature academically and to understand the true meaning of research. I also thank Professor James Carothers, Professor Georg Seelig, Professor Wendy Thomas, and Professor Paul Wiggins for valuable inputs and constructive comments. Their inputs have been crucial for expanding my perspectives and bringing my research to a higher level. I would further like to thank Dr. Bryan Bartley, Dr. Joseph Hellerstein, Dr. Matthias König, Dr. J. Kyle Medley, and Dr. Steven Wiley for their help and suggestions. In particular, their companionship, both academic and daily, has made my life more enjoyable as a graduate student pursuing the Ph.D. degree. Finally, I thank all of my friends and family for their endless encouragement and support throughout my time completing the work and this manuscript.

DEDICATION

To my parents, sister, and grandma.

Chapter 1

INTRODUCTION

1.1 Modeling in Systems Biology

Systems biology is a field of science that investigates the subject at a systems level. This approach differs from the traditional reductionist approach taken by other fields of biology where one usually takes into account a single interaction or a single protein at a time. Instead of treating the system as a sum of its parts, systems biology takes the viewpoint of holism and attempts to understand the system as a whole. In particular, cells are considered to be a complex system, where many diverse elements come together to generate complex behaviors [69]. To understand the complex, emergent phenomenological behavior of a cell, we need to study the system on a larger scope. Thankfully, advancements in proteomics, metabolomics, and genomics have made possible high-throughput experiments generating a large amount of data necessary for studying biological systems at a bigger scale.

Systems of interest in the field of systems biology include various biochemical reaction networks such as metabolic pathways (Figure 1.1), signaling pathways, and gene regulatory networks. These systems are similar in the sense that they are related to complex interactions between numerous biological components crucial for understanding emergent biological behaviors, including but not limited to: metabolism, intra and extracellular communications, cell division, differentiation, apoptosis, etc.

Biochemical reaction networks are incredibly complex. The scope of systems biology can be as small as a part of a signaling pathway to a whole cell or even to a whole organ. Figure 1.2 illustrates a large-scale epidermal growth factor receptor-insulin receptor (EGFR-IR) model [20]. Whole cell models are even larger and more complex. To comprehend models like this, one needs to perform various analyses on the model.

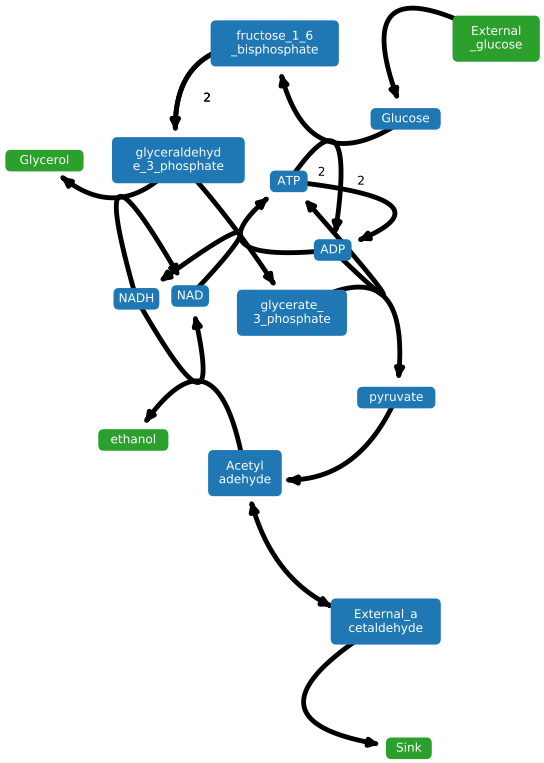


Figure 1.1: A glycolysis model [61], converting glucose to pyruvate while releasing energy through production of ATP and NADH.

The models used in systems biology are often represented by a set of ordinary differential equations (ODEs). The set of ordinary differential equations determines changes of species amounts over time. The specifics of kinetics are determined by rate laws; simulations can be deterministic or stochastic. Time-course simulations will show how the system dynamically changes over time on a given time scale. The steady-state analysis determines both the stable and the unstable equilibria where the net rate of change is zero. Certain systems exhibit bistability, which is explored through bifurcation analysis. Finally, metabolic control analysis (MCA) studies the local and global sensitivity of a system, analyzing how a perturbation in the system propagates through the network.

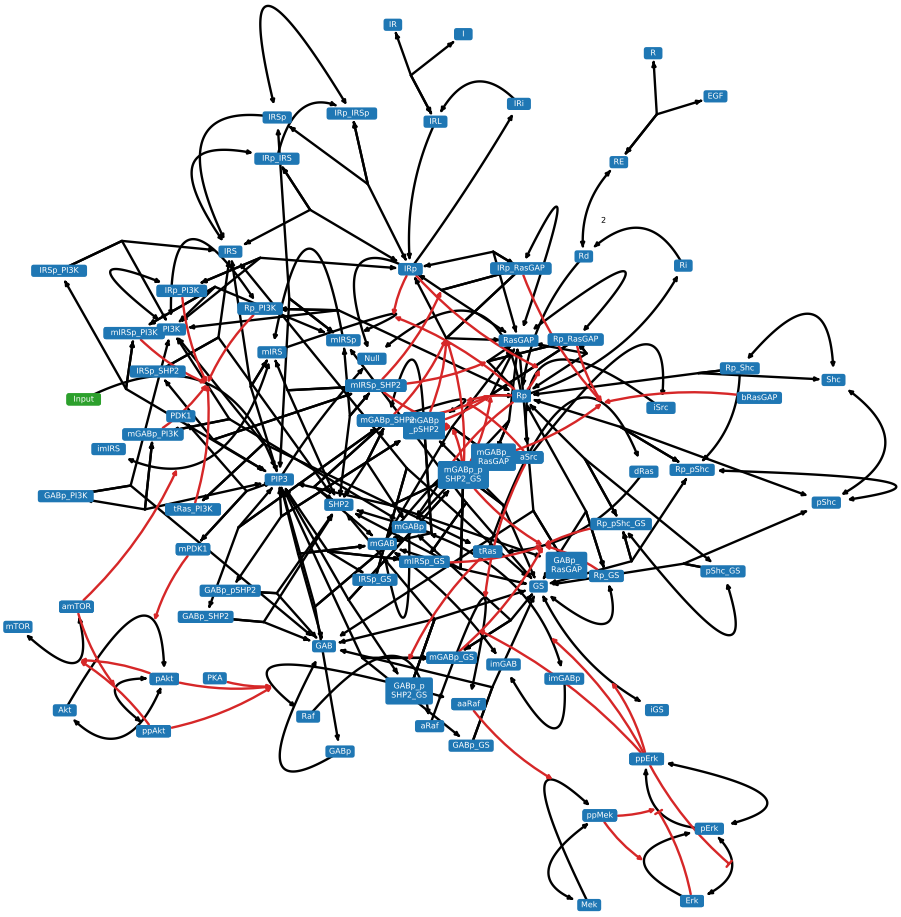


Figure 1.2: A large EGFR-IR model [20].

1.2 Reproducibility of Computational Models

Reproducibility is a cornerstone of scientific research. As one of the core principles of the scientific method, scientific findings are expected to be reproducible; they are distrusted otherwise. Unfortunately, over the past years, several studies have demonstrated that a significant number of scientific studies are not reproducible [6, 11, 102]. This has led researchers and funding agencies to call for more reproducible research [6]. Many experimental biomed-

ical studies are inherently challenging reproduction because it is often difficult to control every experimental variable. For instance, it is challenging to reproduce mammalian cell culture experiments that rely on undefined media derived from animal sources.

Here we focus on the reproducibility of computational systems and synthetic biology. At the first glance, computational studies are expected to be reproducible because computation can be precisely controlled, unlike wet laboratory experiments. However, computational studies often turn out not to be reproducible. Such irreproducibility often comes from: incomplete, untested, outdated, or missing instructions for running the software; missing, outdated, or unannotated source code files; missing or incorrect data. Without well-annotated source codes, readers cannot help trying to reproduce computational experiments from scratch, relying on the descriptions provided in publications, which are often incomplete. Furthermore, computational results often cannot be independently verified via distinct computational or experimental methods, because the assumptions made in computational experiments are often not clearly communicated. The poor reproducibility of computational systems and synthetic biology studies is a serious problem. As computational systems and synthetic biology studies become more ambitious and require more collaboration, e.g., in building comprehensive models of entire cells, organs, and tissues, reproducibility will become more critical for establishing trust in each other's work.

In this section, we review recent efforts to increase the reproducibility of computational systems and synthetic biology, and highlight the most common standards and software tools for reproducible research in these fields. Unfortunately, the terminology used to describe whether a scientific experiment is reproducible or not is not universally agreed upon [7]. Therefore we have to define our terms first: We consider a study to be reproducible when the findings of the study can be independently verified. The degree of independence may vary from partial or complete independence. By verified, we usually mean that the result of an experiment is reasonably close to the reported result, given some moderate variance in the experimental procedure.

For a computational study to be reproducible, the study should be exchangeable and

transparent [84]. Exchangeability stands for the ability to carry out a study in more than one computational environment and produce similar results [84]. When a study is exchangeable, one can utilize outputs from a published computational study to another. Transparency is a feature that allows one to inspect and understand the details of a study including models and simulation experiments [84]. When a study is transparent, one can understand various assumptions and decisions made in the study.

So far the computational systems and synthetic biology community has attempted to ensure exchangeability by designing standards. Unifying the way information is stored indeed ensures exchangeability because standards are interoperable, meaning that it is platform-independent. The original objective of these standards was to exchange models, simulation experiments, and designs between software tools without losing information. As a result, the standards help make computational studies reproducible. If the specifications are extensive enough, the standards can encode the information necessary for complete reproduction of the output. The standards that are accepted by the community reduce the barrier for the authors to exchange results with others, eliminating issues associated with the lack of information. In consequence, several standards for different types of information emerged. Among them, four of the most common standards will be discussed here, including the Systems Biology Markup Language (SBML) [58], the Simulation Experiment Description Markup Language (SED-ML) [130], the COMBINE archive [14], and the Synthetic Biology Open Language (SBOL) [47, 46], which are all widely used standards in the field of systems and synthetic biology.

First, SBML [58] describes biological processes. It is an XML-based *de facto* standard for representing models of biochemical processes such as cell signaling pathways and metabolism. The core SBML standard describes kinetic models. In addition, there are currently twelve SBML packages which extend SBML to describe additional types and aspects of models. These packages range from supporting random distributions and constraint-based models to hierarchical compositions and layouts for a visual representation of pathways. As a *de facto* standard, SBML is widely used by the community. For example, the BioModels model repos-

itory (`https://www.ebi.ac.uk/biomodels`) uses SBML to represent models. CellML [76] is also a popular standard for describing computational systems biology models. In contrast to SBML, which emphasizes the semantic biological meaning of models, CellML focuses on the mathematics of models. This focus on mathematics makes CellML applicable to a broader range of scales – tissues, organs, and whole organisms – than SBML. However, the focus on mathematics also makes CellML-encoded models more difficult to comprehend, reuse, and compose.

Next, SED-ML describes simulation experiments [130]. This includes the algorithms used in the experiments, the parameters for each simulation run, which simulation predictions should be recorded, and how these predictions should be analyzed and plotted. SED-ML is necessary because model descriptions alone are insufficient to reproduce simulation results. The information on how to simulate the model is required. Together, SBML and SED-ML can capture enough information to reproduce completely a simulation result such as a figure in an article.

The COMBINE archive is a standard for archiving all the information necessary for reproducing a computational experiment. Such information might include raw data, model descriptions (e.g., SBML files), simulation experiment descriptions (e.g., SED-ML files), and more [14]. In particular, the COMBINE archive aims to standardize the encapsulation of all files needed for reproducing a computational study in a single, easily exchangeable file.

Finally, SBOL describes genetic designs for synthetic organisms [47, 46, 33, 9], including information about the genetic sequence, components, and their interactions. The standard encodes genetic designs from the sequence level to the component level along with their interactions, which is the largest difference compared with the GenBank format. SBOL also supports hierarchical designs to enable researchers to compose collaboratively genetic elements into entire synthetic organisms. Table 1.1 lists the standards discussed in this manuscript.

Standard	Purpose(s)
COMBINE archive	Standard to encapsulate all files for a simulation study into a single archive
SBML	Standard to describe models of biochemical processes (`http://sbml.org/SBML_Software_Guide`)
SBOL	Standard to describe designs for synthetic organisms (`http://sbolstandard.org/applications`)
SED-ML	Standard to describe simulation experiments (`https://sed-ml.github.io/showcase.html`)

Table 1.1: Common standards for computational systems and synthetic biology.

1.3 Overview of Tools Available for Systems Biology

While each of all these standards is a great step towards reproducible computational studies, without proper implementation a standard is effectively useless. Using the standards described above requires user-friendly, standards-compliant software tools for building and simulating models. Many software applications for systems biology support SBML to varying degrees including, but not limited to, CellDesigner [44], COPASI [57], iBioSim [89], Jarnac [114], JWS Online [98], openCOR [48], PathwayDesigner [109] (formerly called JDesigner), PySCeS [97], SBW [17], and TinkerCell [24]. This widespread support has been spurred by the availability of software libraries for reading, writing, and modifying SBML-encoded models for C, C++, Java, MATLAB, Python, and other languages. Several of these software tools also support SED-ML and the COMBINE archive to varying degrees, including CellDesigner, COPASI, iBioSim, JWS Online, openCOR, and SED-ML Web Tools [16].

Software	Descriptions
CellDesigner	Diagram-based editor that also supports simulation and analysis (http://www.celldesigner.org)
COPASI	GUI-based simulation and analysis tool for biochemical networks (http://copasi.org)
iBioSim	JAVA-based simulation and analysis tool targeted for genetic circuits (http://www.async.ece.utah.edu/ibiosim)
Jarnac	Simulator for reaction networks (http://systems-biology.org/software/simulation/jarnac.html)
JWS Online	Web-based tool for modeling and simulation (http://jjj.biochem.sun.ac.za)
openCOR	GUI-based simulation and analysis platform for CellML (http://www.opencor.ws)
PathwayDesigner	GUI-based graphical modeling environment for biochemical networks (http://pathwaydesigner.org)
PySCeS	Python package for simulation and analysis (http://pysces.sourceforge.net)
SBOLDesigner	GUI-based CAD software tool for designing genetic circuits that supports SBOL (http://www.async.ece.utah.edu/SBOLDesigner)
SBW	Software framework integrating various applications for systems biology (http://sbw.sourceforge.net)
TinkerCell	GUI-based CAD software for genetic circuits (http://www.tinkercell.com)

Table 1.2: Software discussed in this section.

Although SBOL is a comparatively new standard, it is also supported by several software tools including iBioSim, SBOLDesigner [132], and TinkerCell. Online repositories such as SynBioHub [82] also support SBOL. This is facilitated by SBOL libraries for C++, Java, JavaScript, and Python. Table 1.2 lists some of the many tools which support these standards.

Software tools discussed here are designed to support a wide range of tasks including designing models (e.g., PathwayDesigner), simulating models (e.g., COPASI, JWS online), and visualizing models (e.g., CellDesigner). Software tools also support the standards in varying degrees. Some might support SBML only while others might support both SBML and SED-ML. Regardless of the extent of support, availability of software tools is crucial for the longevity of standards. One of the merits of designing a standard is that as long as the software support is available, the standard survives. Many of these software tools might become obsolete after few years, but as long as there are new tools available to keep the standard afloat, information encoded in these standardized formats will still be exchangeable and study reproducible. Open source software tools are advantageous because even if the original authors of a software drop the support, the source code will always be available to the public to maintain if desired. This is why projects involving software tool support for standards should prefer open source license and use public repositories such as GitHub (`https://github.com`).

1.4 Robust, Reliable Models and Model Ensembles

A model is considered to be robust when it can withstand various perturbations. In other words, a model is robust when it can not only explain the observations used to construct the model but also predict new behaviors under various conditions. Furthermore, a model is considered to be reliable if it can predict the expected behavior without failure. A model that is both robust and reliable is considered as a good model because it can make novel predictions, which are trustworthy.

Unfortunately, building a robust and reliable model of biological processes is difficult

for a few different reasons. First, model validation is challenging. Experimental techniques have been greatly improved but a certain aspect of experimental data is expensive or outright impossible to collect. Second, models of biological processes are often 'sloppy'. Sloppy models are generally insensitive to changes except to a small number of 'stiff' parameters [52, 34]. The sloppiness of a model seems to be related to multiparametric nature of biological models. For biochemical reaction networks models used in systems biology, even some of the simplest models exhibit issues of parameter unidentifiability arising from this characteristic. When no prior knowledge is available of which parameters are 'stiff', experiments for validation can be incredibly inefficient.

Models used in systems biology simulate the dynamics of biochemical networks such as signaling pathways, metabolic pathways, and gene regulatory networks. These models are typically encoded with a set of ODEs and visualized as network diagrams. Models can be solved and simulated to make predictions under various conditions, furthering our understanding of the network and making them useful for the processes such as drug discovery [22, 69, 70]. Building a robust, reliable model of biochemical networks is of particular interest largely because of its applications to medicine. Systems biology today is heading towards multi-scale modeling, constructing larger and more complex models [91]. Examples include the whole-cell model of *Mycoplasma genitalium* [66] and the central metabolism model of *E. coli* [86]. However, as the size and complexity of a model grow, validation becomes more and more difficult. A large portion of these models is composed of multiple submodels, where each submodel should be validated against the data. Building a robust, reliable biochemical network model requires the modeling effort to define accurately 1) model topology, 2) reaction kinetics, and 3) parameter values.

There are two ways to build a better model: First, one should supply more, better data. Collecting additional measurements (e.g, increase the size of the dataset, collect other types of data) and increasing the quality of the data (e.g. reducing the noise) help construct a better model. Second, better algorithms should be implemented. For example, the algorithm used for parameter estimation may be changed to one that performs better.

There have been significant advances in both experimental and computational techniques that might contribute to improving the quality of the model. On the experimental side, we currently have high-throughput data acquisition techniques for various types of experimental data. Notable examples include those techniques involving CRISPR-Cas9 [103, 27, 50, 26], proteomics [120, 119], metabolomics [117], genomics [36, 127], and so on, all of which now have multiple ways to acquire large-scale experimental data.

One striking innovation comes from the advancement of CRISPR through the introduction of the CRISPRa/i (activation/inhibition) technique. CRISPRa/i screening allows highly selective activation and inhibition of specific target genes [103, 27, 50, 26], making it adequate for perturbation studies. Proteomics is another area where there has been significant progress in terms of experimental techniques. These advances allow targeted proteomics to be ultra-sensitive and quantitative, allowing the measurement of low levels of protein abundance [120, 119].

Computational biology, in general, has experienced significant progress as well. The computational hardware is progressively improving. Availability of commercial clusters allows large-scale computational studies. There have also been numerous attempts to integrate various advanced and effective computational approaches to solve biological problems, in the form of ensemble modeling [55, 19], information theory [55], machine learning [19, 131, 43], inference techniques [96, 35, 81], and others [99, 74].

In particular, ensemble modeling provides a way to improve the robustness of biochemical reaction network models. The idea of ensemble modeling is to generate a group of models instead of a single model when there are multiple competing models. For biochemical reaction networks such as metabolic pathways, it is hard to get a complete description of the system due to lack of information of kinetics, etc. [126] The model ensemble can be analyzed further to infer characteristics of the system of interest. A model ensemble can make robust predictions when the right methodology to construct the ensemble has been chosen [106]. The multivariate nature of a general biochemical reaction network model makes it difficult to pinpoint the topology, kinetics, and parameter values. In such a case, the reasonable

path to take is to consider the ensemble of models as a whole, holding back from making assumptions until new, differentiating data are supplied. Apart from making predictions out of the model ensemble, the ensemble can be analyzed to infer a set of experimental data that can maximally reduce the size of the ensemble. In this fashion, we can complete the cycle between modeling and experimental efforts, increasing the efficiency of experiments.

Chapter 2

ENVIRONMENT FOR MODELING, SIMULATION, AND VISUALIZATION

Reproducible, robust, and reliable computational studies require software tools to do so. The manuscript discussed several software tools available for systems biology but there are reasons why many of these software tools are inadequate to be used as a general computing environment for complex and high-throughput studies. First, many software tools are standalone applications. Standalone applications are hardly extensible, meaning you cannot easily add new functionality to the software. The lack of flexibility restricts what you can do with the software tool, potentially limiting the complexity of the problem you want to solve. Second, many software tools have no scripting capability, severely limiting the scale of the study. No scripting capability means very little automation and limited low-level data processing, both of which are crucial for complex and high-throughput studies.

It has become apparent that there is a need for a computing environment for systems biology that circumvents these issues. At the same time, we want the environment to be easy to use, widely available and widely supported. Of all the choices available, one programming language stands out: Python.

Python has proven to be a very popular language for scientific computing and data science. The ease of learning and use encourages newcomers. The extensible nature of the environment and general computing language means that the language appeals to core users. The open-source nature of the language naturally attracts scientific computing projects for building and maintaining tools. This generates positive feedback, where people try out and build new software packages for the language, therefore attracting more people. As a result, Python now hosts extensive libraries of scientific computing packages. All these features

have made Python an ideal platform to host computing environment for systems biology.

The systems biology community also have shown support for Python through the development of a variety of software tools. These include PySCeS [97] with a focus on simulation via differential equations, structural analysis, and metabolic control analysis; SloppyCell [90], with a focus on model fitting and calculating the resulting uncertainties; pySB [77], with a focus on rule-based reaction models; COBRApy [40], with a focus on constraint-based modeling; GillesPy [1], with a focus on stochastic simulations; and ssbio [85], with a focus on structural analysis. Our lab also has developed packages for Python, including libRoadRunner [122], which is a high-performance ordinary differential equation simulator, and Antimony [121], which is a human-readable/writable interpretation of SBML.

However, software support for systems biology in Python is lacking in three ways. First, there is no coherent modeling and simulation environment available for Python. Python tools discussed in this manuscript are distributed as individual packages and focus on a specific set of functionalities. This makes setting up a Python environment for systems biology quite cumbersome, requiring users to follow multiple and often fragile steps for proper configuration. This can be problematic for both novices and experts in the field. Not only that, package-based distributions limited the extent to which one can customize the environment. A concrete environment was necessary to build other tools and algorithms for reproducible, robust, and reliable models upon.

Second, there are no good model visualization and network analysis tool for Python. Systems biology often uses network diagrams to visualize reaction networks. While a set of ordinary differential equations provide a complete description of the model, network diagrams are better for the qualitative understanding of the model. Moreover, the visualization of model analysis directly on the network diagram is a powerful way to connect the qualitative and quantitative aspects of the model. Currently, there are no good options to visualize network diagrams for systems biology in Python. Python packages like NetworkX [53] and PyGraphviz provide support for graph visualization, but these packages are tailored to computer science rather than systems biology. These packages might be able to visualize

undirected or directed graphs, but will not support reversible reactions, regulators, reaction nodes, etc. which are crucial for reaction networks. Not only that, none of these packages support models encoded in SBML.

Third, support for reproducibility leaves a lot to be desired. Many of the existing tools support the standards for systems and synthetic biology to a limited degree. For example, PySCeS supports SBML and a portion of SED-ML. SloppyCell also supports SBML, as does COBRApy. PySB offers some support for reading and writing SBML models. However, none of the Python tools described here supported the COMBINE archive or even fully supported SBML and SED-ML, which is the minimum requirement necessary for ensuring the reproducibility of a study.

In this chapter, I present solutions to the first two of the issues discussed above. First, I present Python-based modeling, simulation, and analysis environment called Tellurium. Second, I present a network visualization and analysis package called netplotlib. For each of this software, I discuss the implementation and provide some examples as a demonstration. Tools to ensure reproducibility will be extensively discussed in the next chapter. All of the software tools discussed in this chapter are included in Tellurium environment.

2.1 Tellurium

The philosophy behind Tellurium [31, 84] is to provide high-performance modeling, simulation, and analysis computing environment that is both scalable and extensible. We bring together a wide variety of libraries and tools for researchers in systems and synthetic biology. Tellurium is distributed using one-click installers to make the installation process extremely simple. Tellurium provides a convenient one-stop solution for many of the needs of the community, which is especially helpful for novices who do not wish to manually configure the various tools we distribute with Tellurium. For systems biology modeling, Tellurium supports various modeling standards including SBML, CellML, SED-ML, the COMBINE archive, and SBOL. In addition, we distribute libRoadRunner [122] for simulation, AUTO2000 [39] for bifurcation analysis, and Antimony [121], phraSED-ML [32] for simplified model creation and

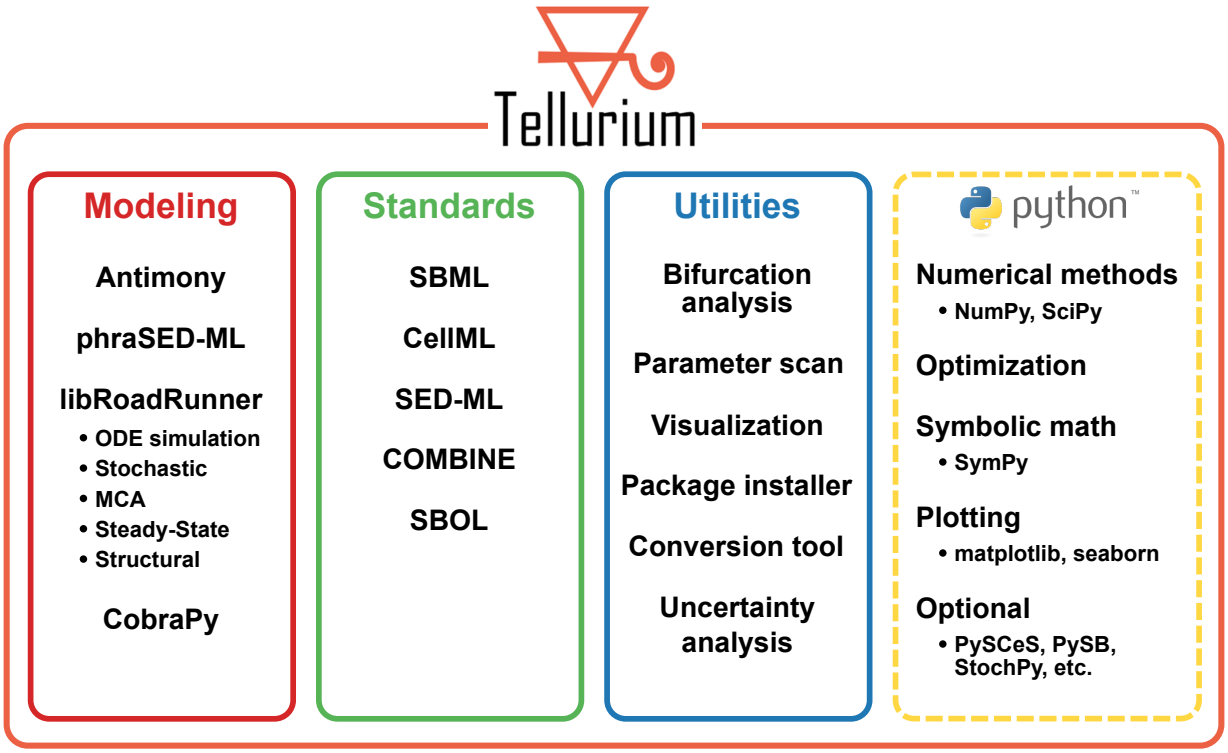


Figure 2.1: Overview of Tellurium. Tellurium is composed of three distinct functional pillars including standards support, modeling support, and utilities. Several third-party Python packages come with Tellurium and additional packages can be installed if needed.

modification. Along with the tools distributed with Tellurium, we provide a simple method for users to install additional Python packages, making Tellurium highly extensible.

2.1.1 Implementation

Tellurium is implemented in a mixture of C, C++, and Python. Python is used as the front-end language for the sake of ease of use. The software can be roughly partitioned into three functional pillars: 1) modeling; 2) standards support; and 3) general utilities (Figure 2.1).

Tellurium includes the modeling and numerical support for model design and analysis. Tellurium comes with packages such as Antimony [121] and phraSED-ML [32] which translate

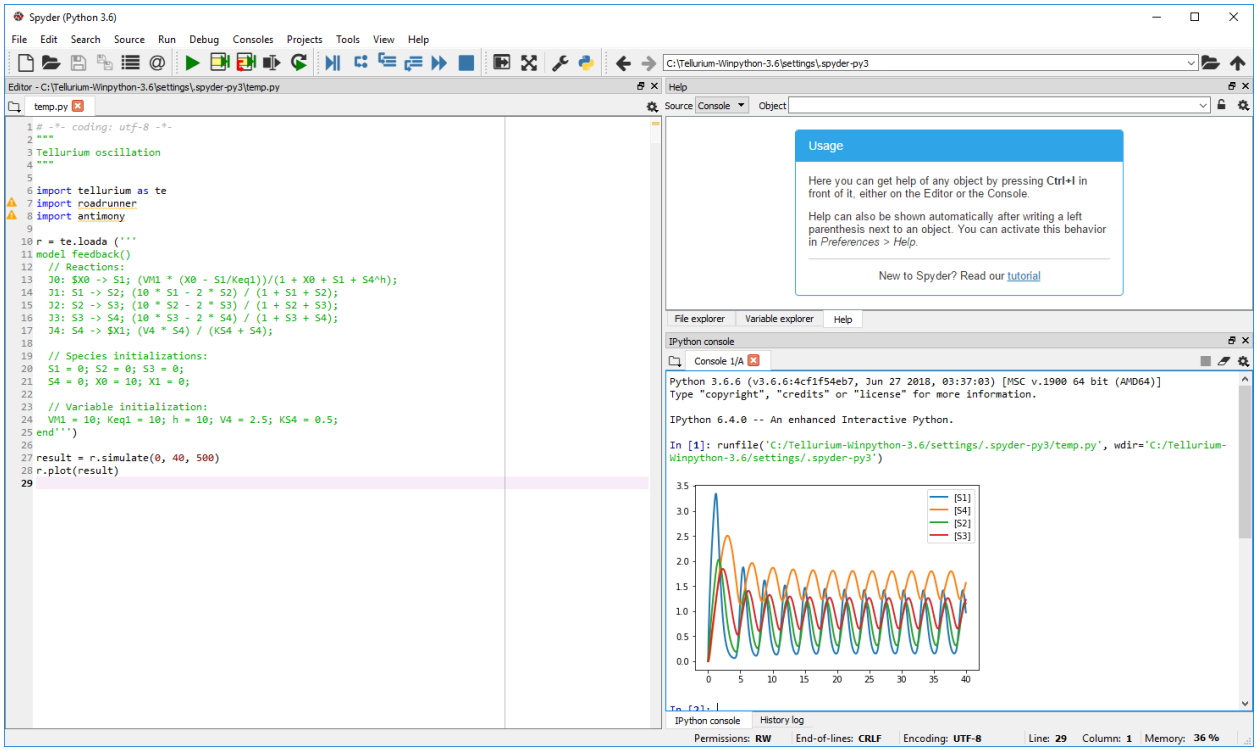


Figure 2.2: Screenshot of Tellurium. Tellurium is based on Spyder IDE which provides a MATLAB-like development environment.

model and simulation setup in SBML and SED-ML format to human-readable counterparts. The numerical support includes libRoadRunner [122] which provides a variety of analyses including ordinary differential equation simulation, Gillespie-based stochastic simulation, metabolic control analysis, and structural analysis of networks via libStructural [10].

Support for standards in systems and synthetic biology is included in Tellurium via the respective libraries such as libSBML [21], libSEDML [12], libCOMBINE [15], libSBOL, and basic support for CellML [54] via Antimony. Many of these libraries come from third-party developers and some have been augmented for Tellurium to make them easier to use. For example, SimpleSBML simplifies model building instead of requiring users to use low-level methods in libSBML. Tellurium provides extensive layers to libSBML and libCOMBINE to simplify the process of generating COMBINE archives. We use COMBINE archives to

Packages	Descriptions
Antimony	Human-readable/writable representation of SBML (`https://github.com/sys-bio/antimony`)
phraSED-ML	Human-readable/writable representation of SED-ML (`https://github.com/sys-bio/phrasedml`)
simpleSBML	Library for simplifying model manipulation in libSBML (`https://github.com/sys-bio/simplesbml`)
libRoadRunner	Fast simulator supporting ODE and stochastic simulations, metabolic control analysis, and others (`http://libroadrunner.org/`)
COBRApy	Library for constraint-based modeling (`https://opencobra.github.io/cobrapy/`)
NumPy	Library for array-based computation (`http://www.numpy.org/`)
SciPy	Library for scientific computing (`https://www.scipy.org/scipylib/index.html`)
SymPy	Library for symbolic mathematics (`http://www.sympy.org/en/index.html`)
matplotlib	General 2D plotting library (`https://matplotlib.org/`)
seaborn	Statistical visualization library (`https://seaborn.pydata.org/`)

Table 2.1: Table of selected packages distributed with Tellurium.

facilitate simulation reproducibility.

Tellurium also comes with numerous utilities designed for systems biology. For example, Tellurium comes with a module for bifurcation analysis, which is crucial for understanding

models with multiple steady states. This type of analysis can be difficult for a novice to perform, so a wrapper to AUTO2000 is provided which interfaces itself to libRoadRunner. By implementing it as a plugin for libRoadRunner, AUTO2000 can directly access the simulation engine and perform computations without the overhead of a cross-language API. This also means that the bifurcation tool can be used outside of Python and hosted by other tools. Note that unlike other AUTO2000 implementations, our implementation does not require an external compiler because this task is handled by libRoadRunner.

To demonstrate the flexibility in a Python ecosystem, we also bundle COBRApy [40], which is one of the primary constraint-based modeling packages. In addition, common Python packages that are essential in scientific computing are bundled with Tellurium. These include, but are not limited to, SciPy and NumPy (for a large variety of numerical methods), SymPy (for symbolic manipulation), and plotting libraries such as matplotlib and seaborn. Tellurium comes with numerous Python packages for systems and synthetic biology and scientific computation, but more packages can be installed using `installPackage` function. Any Python package available on PyPI can be installed, making Tellurium highly extensible.

Tellurium comes with its own Python distribution. This is beneficial for the user who wishes to run multiple different Python environments within the same machine. It also helps us customize the environment further. Table 2.1 lists short descriptions of the packages discussed in this manuscript.

Tellurium is distributed in two interfaces: The first is Tellurium Spyder (Figure 2.2), which is based on Spyder IDE and provides a MATLAB-like environment for researchers who are already familiar with editor/console type programming. Spyder IDE is a Python-based development environment that comes with powerful tools like profiler and static code analysis. Spyder IDE is ideal for modelers and developers who prefer generating and debugging raw Python scripts. For those who prefer notebook-like interfaces, we provide a Jupyter notebook-based version called Tellurium Notebook. Jupyter notebook differs from Spyder IDE as it creates documents containing live code, plots, narrative texts, and equations. Moreover, Jupyter notebooks are interactive, making it ideal for sharing and displaying the

work with others. It is also possible to install Tellurium and its dependencies in an existing Python environment through pip. Examples of alternative hosts that have employed Tellurium include PyCharm, Sublime Text, and Atom.

2.1.2 Applications

In this section, several use cases of Tellurium are illustrated. In particular, various tools included in Tellurium are demonstrated as well as its ability to integrate with other Python packages. Example of model building and simulation are presented. Various analysis tasks such as metabolic control analysis, bifurcation analysis, parameter estimation, and parameter confidence interval estimation are also demonstrated.

Model Building and Simulation

One of the basic functionalities of Tellurium is to build models and run simulations. Models in Tellurium are defined using Antimony [121], a human-readable model definition language for biochemical reactions. Antimony supports a large part of SBML specification including events and assignment rules and can be translated to and from SBML. Figure 2.3 illustrates a model of a simple linear chain involving five floating species and corresponding simulation result. The corresponding model and the simulation result can be written in Tellurium using the following script.

Listing 2.1: A script modeling the linear chain illustrated in Figure 2.3

```
import tellurium as te


AntimonyStr = """
J1: $Xo -> S1; k1*Xo - k2*S1;
J2: S1 -> S2; k3*S1 - k4*S2;
J3: S2 -> S3; k5*S2 - k6*S3;
J4: S3 -> S4; k7*S3 - k8*S4;
```

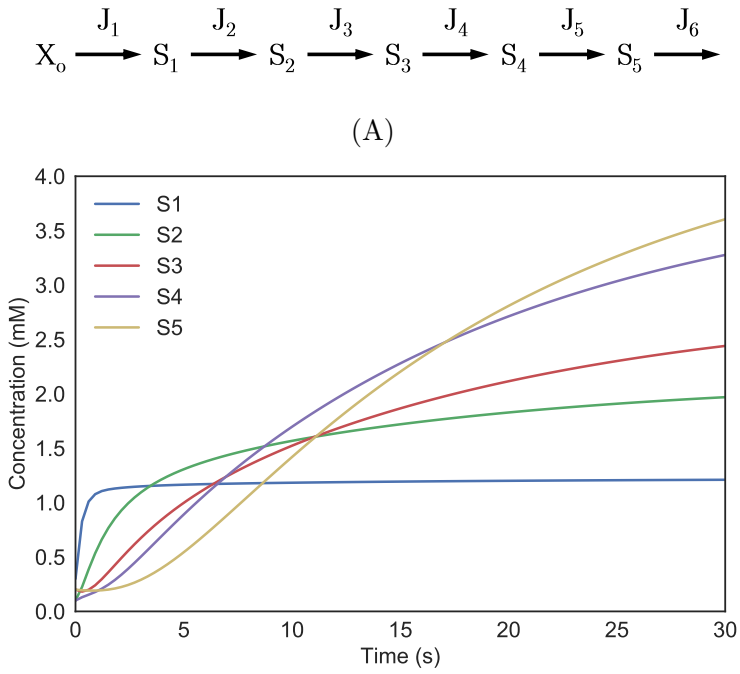


Figure 2.3: A simple linear chain model with five floating species is written in Antimony language and simulated to produce time-course traces. (A) The model represented in network diagram. Species X_o is a boundary species (fixed) and each reaction is modeled using reversible mass-action kinetics. J_i is used to label each reaction. (B) The Result of the simulation.

```
J5: S4 -> S5; k9*S4 - k10*S5;
J6: S5 -> ; k11*S5;
Xo = 1.0; S1 = 0.3; S2 = 0.1;
S3 = 0.2; S4 = 0.1; S5 = 0.2;
k1 = 3.92; k2 = 2.83; k3 = 0.8;
k4 = 0.24; k5 = 0.68; k6 = 0.35;
k7 = 0.82; k8 = 0.47; k9 = 0.37;
```

```
k10 = 0.22; k11 = 0.1;
"""

r = te.loadAntimonyModel(AntimonyStr)
result = r.simulate(0, 30, 100)
r.plot()
```

Antimony language allows users to define reactions using arrows and the type of reaction is determined using rate law next to it. Here, species X_o is a boundary species, where the species concentration is fixed. Once defined and species and parameters initialized, the string block can be loaded in to create a libRoadRunner object using `loadAntimonyModel()` function. Than the libRoadRunner instance is simulated using `simulate()` and plotted using `plot()` functions.

Metabolic Control Analysis

An important part of the modeling and model analysis process is sensitivity analysis, which provides information about the effect of system parameters and states on the results. A standard approach for sensitivity analysis is metabolic control analysis (MCA). Tellurium calculates the various elasticities and control coefficients defined in MCA [64, 112] using libRoadRunner [122]. In addition, there is support for frequency dependent MCA in the form of Bode plots [60]. A number of utilities are provided to make it easier to visualize results from MCA studies. In particular, we provide utilities to help visualize flux control, concentration control, and elasticity profiles using heat maps. Figure 2.4 shows heatmaps of the distribution of flux and concentration control coefficients in a linear pathway of six reactions (Figure 2.3).

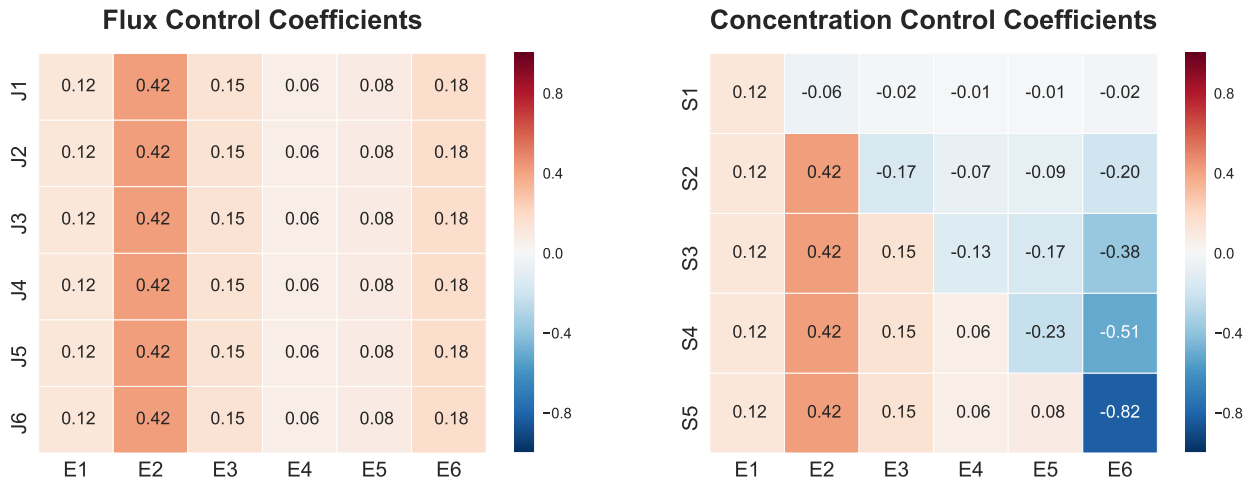


Figure 2.4: Two heatmaps showing the flux and concentration control coefficients for a linear reaction chain of six reactions and five floating species illustrated in Figure 2.3. E_i is the enzyme level for reaction i, and J_i is the flux through reaction i. S_i is the substrate label. Red indicates positive values and blue indicates negative values. For example, reaction step six, E_6, has a strong negative influence on species S_5.

Monte Carlo Simulation of Linear Reaction Chain

Monte Carlo simulation is a method based on a repeated random sampling of variables with given probability distributions. Here, we show that Tellurium can be used to visualize the distribution of flux control coefficients by performing a Monte Carlo simulation with rate constants as parameters. Consider a model of four reversible reactions

$$X_0 \rightleftharpoons S_1 \rightleftharpoons S_2 \rightleftharpoons S_3 \rightleftharpoons X_1,$$

where X_0 and X_1 are boundary species and the rate law is given by

$$k_n \cdot \text{Reactant} - \frac{k_n}{Keq_n} \cdot \text{Product} \tag{2.1}$$

for each reaction n [110]. For our example, we keep the value of Keq_n constant but vary k_n randomly. The sampling is done 1000 times for each parameter k_n where a random

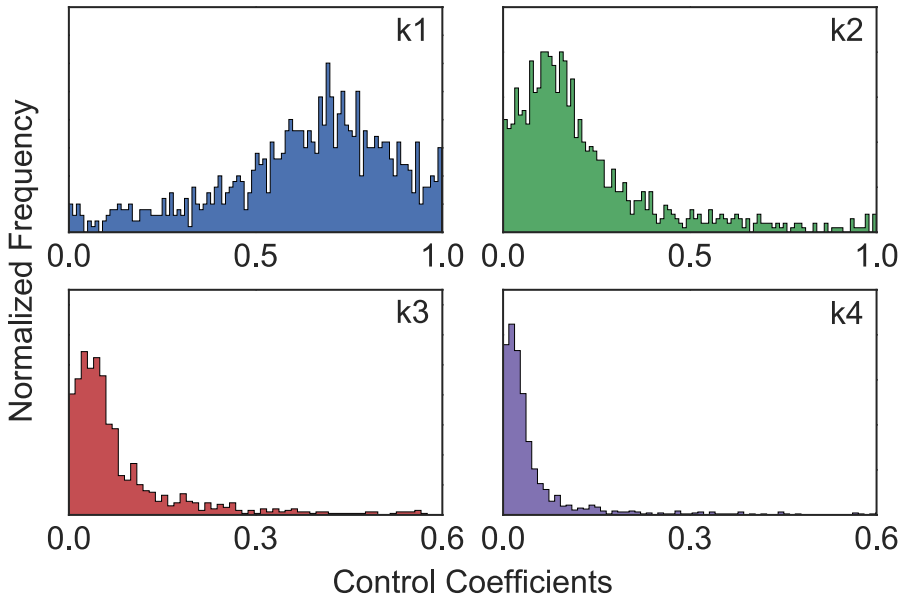


Figure 2.5: Normalized distribution of scaled flux control coefficients for four different rate constants with respect to the first reaction of a regular chain.

parameter value is drawn from a uniform distribution with an interval ranging from zero to ten. For each run, new parameter values are assigned to the roadrunner model and scaled flux control coefficients are calculated for each rate constant with respect to the first reaction of the regular chain. The sampled distributions of flux control coefficients are plotted in Figure 2.5. The distribution reveal that the first rate constant, k_1, has the largest effect on the first reaction with decreasing effects for the downstream reactions.

Bifurcation Analysis

Bifurcation analysis enables qualitative changes in model behavior to be studied as a function of a model parameter. Such qualitative changes include bistability and oscillatory behavior [5, 42]. Tellurium's bifurcation facility is designed to automatically compute a bifurcation in specified parameter search space and plot a bifurcation diagram without user intervention.

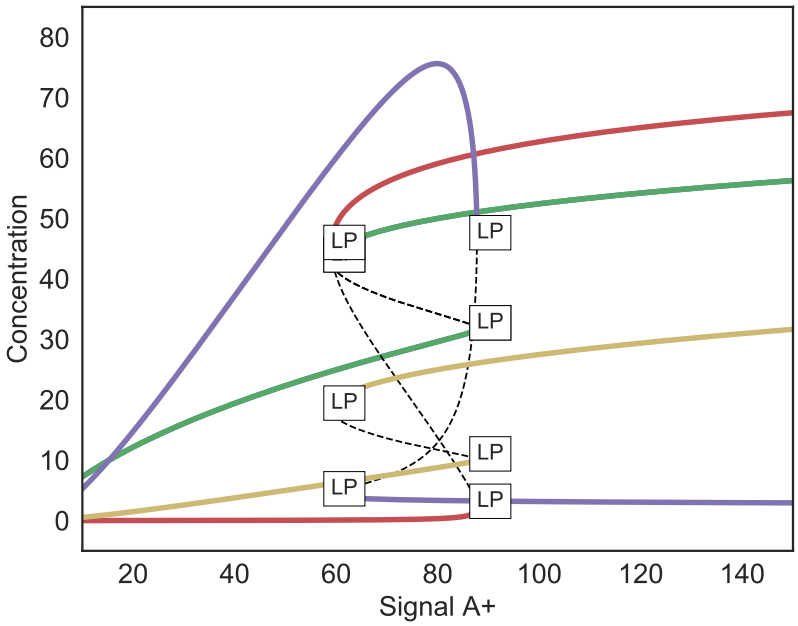


Figure 2.6: Bifurcation analysis applied to a model of an embryonic stem cell switch. The label LP represents a fold or turning point bifurcation. Blue, green, red, and yellow indicate transcription factors OCT4, SOX2, NANOG, and OCT4–SOX2 heterodimer respectively. Blue (OCT4) trace is covered by the green trace (SOX4).

The user specifies a model parameter as the basis for the analysis. The bifurcation tool will then automatically scan a user-specified range of parameter values. If at some point the system changes to an alternate stationary state, the bifurcation is recorded and scanning continues. Figure 2.6 illustrates a number of bifurcation changes in a model of the embryonic stem cell switch [28] with Tellurium. For models where the stoichiometry matrix does not have full rank, libRoadRunner creates the appropriately reduced model [18] thus permitting bifurcation analysis of protein signaling networks to be carried out [108, 107].

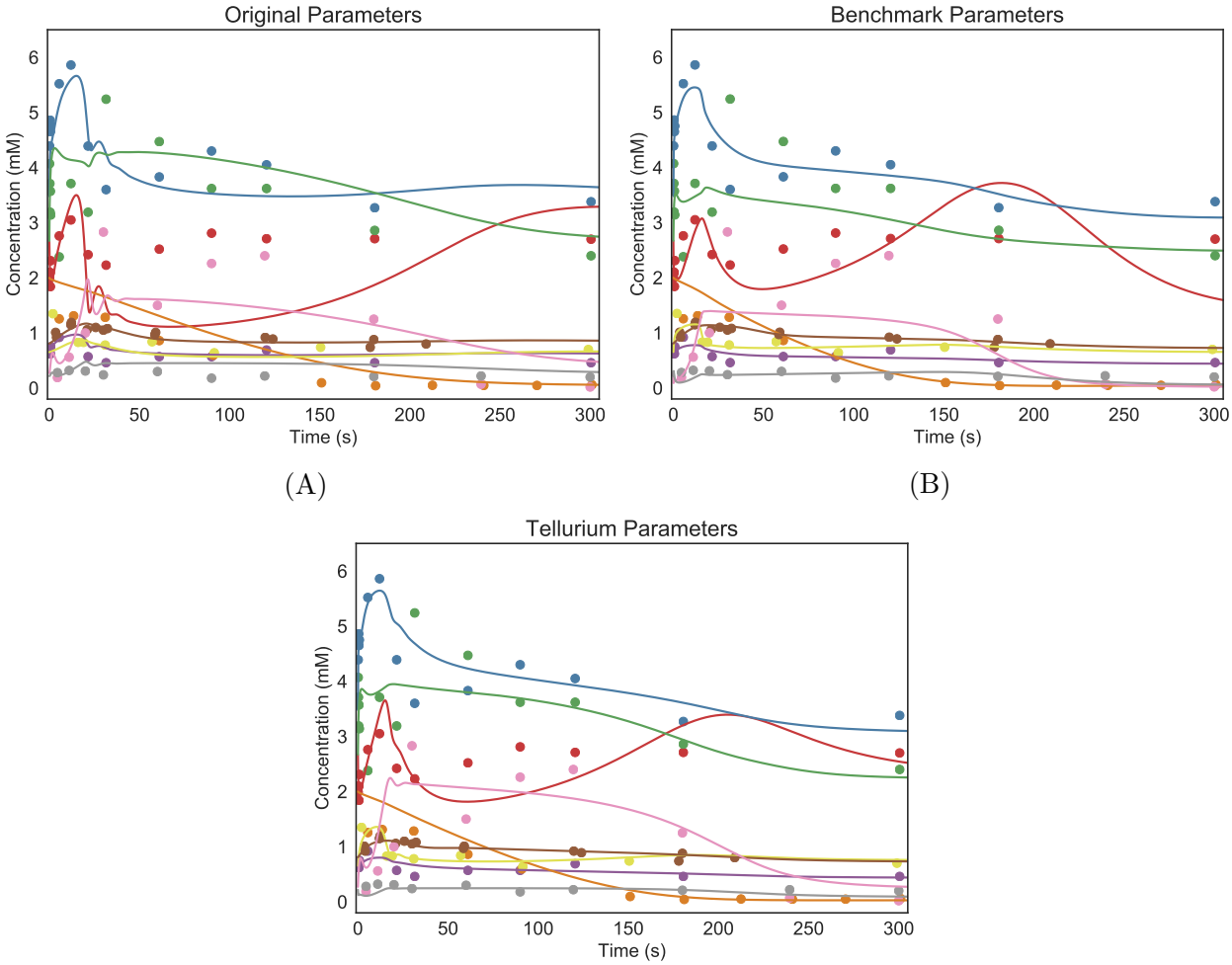


Figure 2.7: Comparison of central carbon metabolism model of *E. coli* fitted against experimental data of 9 metabolites. (A) Fitted curves using the original parameters; (B) fitted curves using parameters from the benchmark suite; (C) fitted curves using parameters from Tellurium. Lines represent simulated data using fitted parameters and dots represent the experimental data. Red, blue, green, purple, orange, yellow, brown, pink, and gray traces and dots corresponds to pep, g6p, pyr, f6p, glcex, g1p, pg, fdp, and gap, respectively.

Parameter Estimation

Parameter estimation is a common step in developing a model where the model is fitted to experimental data. Since Tellurium is based on Python, users can use the various optimization packages available in Python. Moreover, Tellurium provides an environment where parameter estimation routines can be easily customized to deal with almost any fitting problem. To demonstrate Tellurium's abilities in parameter estimation, we used a model of the central carbon metabolism of *E. coli* originally published by Chassagnole *et al.* [25] and later reformulated to be used as a part of benchmark suite for parameter estimation by Villaverde *et al.* [129]. The model is composed of 18 species and 48 reactions with 116 parameters to fit. Experimental data was supplied by the original authors, which consists of 110 time-course data points spread over 9 different metabolites. The reason why we choose this particular model is that 1) we have reference results to compare against, 2) the model is based on measured experimental data, and 3) the model presents a challenging parameter estimation problem where the reported 'optimized' results still does not fit well. Therefore, in this application, our goal will be to get a fit comparable to that obtained by Villaverde *et al.* [129].

The model presents a relatively large number of parameters to fit and many standard local optimization methods fail. Instead, a global optimizer is used to find a proper set of parameters. Here, we use the differential evolution optimizer supplied by the SciPy package. Figure 2.7 shows the result of parameter estimation on Tellurium (Figure 2.7C), which is similar to the fit reported in the benchmark [129] (Figure 2.7B) and better than that of the original paper [25] (Figure 2.7A). To compare the fit, we use cumulative normalized root-mean-squared error ($\sum$ NRMSE), as was done by Villaverde *et al.* [129]. The root-mean-square error measures the average of differences between observed and predicted values, and is given by

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}, \tag{2.2}$$

where N is the total number of the sample, y_i is the observed value, and $\hat{y}_i$ is the predicted

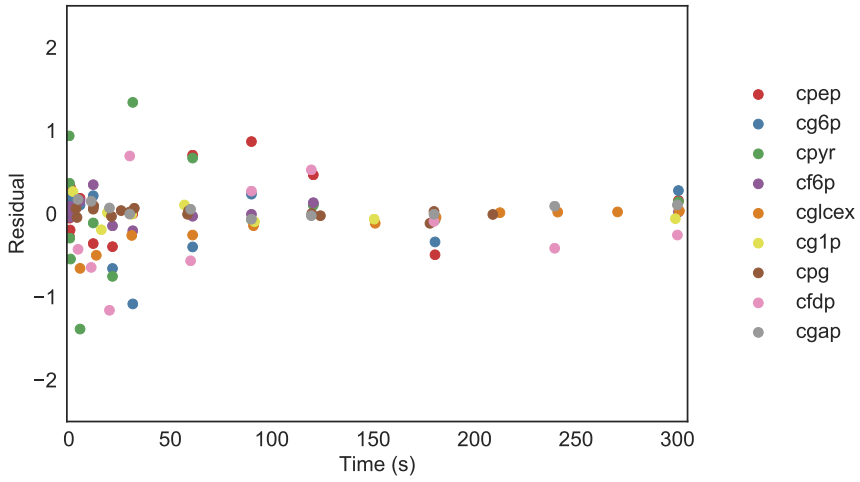


Figure 2.8: Residuals of central carbon metabolism model of *E. coli* fitted against experimental data of 9 metabolites.

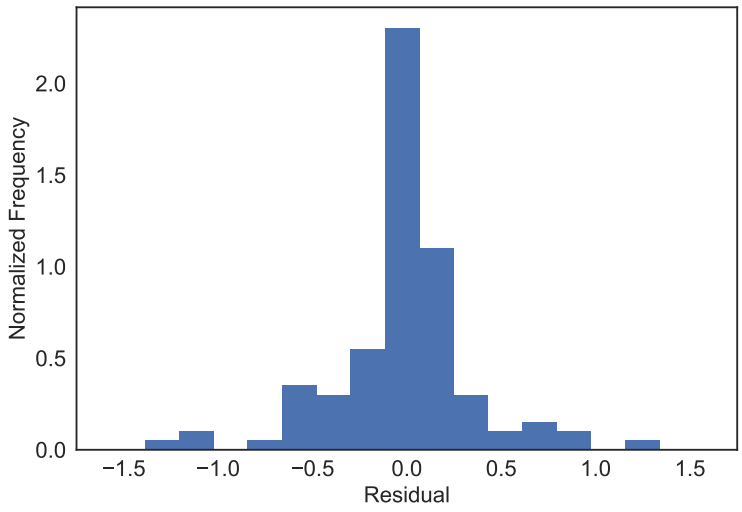


Figure 2.9: Normalized histogram of residuals of all data points combined.

value. This can be normalized properly by dividing by the difference between the maximum and minimum values of observables as follows:

$$\text{NRMSE} = \frac{\text{RMSE}}{\max(y_i) - \min(y_i)}. \tag{2.3}$$

Our parameter optimization run results in $\sum$NRMSE $\approx$ 2.29 which is similar to the value reported by Villaverde *et al.* ($\sum$NRMSE $\approx$ 2.49) [129] and better than the original parameterization given by Chassagnole *et al.* ($\sum$NRMSE $\approx$ 3.61) [25]. Figure 2.7 compares the fitted curves using the parameters published in the original paper, parameters reported by the benchmark suite, and parameters obtained from Tellurium. A plot of the residuals is provided in Figure 2.8. Histogram of the residuals is provided in Figure 2.9.

A single run of parameter estimation using differential evolution took about 4.5 hours on a single core of Intel i7 4770 machine running at 3.4 GHz with 8GB RAM. Approximate standard errors on the fitted parameters can be obtained from the Hessian. For more accurate estimates it would be possible to use Monte Carlo or Profile Likelihood methods [115]. For the scope of this paper, we omit this step, but in the future, we will be supporting massively parallelized workloads through commercial cloud services that a user might be subscribed to.

Parameter Confidence Intervals Estimation using Monte Carlo Bootstrapping

In parameter estimation, a confidence interval refers to the bounds on the estimated parameter value for a given likelihood. Bootstrapping estimates the effect of noise in the data on the fitted parameter values.

In this application, we show how to estimate the confidence intervals of fitted parameters through the use of the Monte Carlo bootstrap method [113, 111]. Monte Carlo bootstrapping estimates confidence intervals by repeatedly resampling the experimental data used for the fitter. The standard procedure starts with an initial estimation of parameters on original experimental data to obtain the residuals and the expected output calculated from the model using the estimated parameters. For each run of bootstrapping, a residual is randomly picked with replacement and added to the expected curve to create a unique synthetic data set, on which another parameter estimation is performed. This procedure is repeated multiple times to obtain a sample of fitted parameter values. This sample is then used to compute the confidence bounds. For a sample of parameter values, the 95% confidence interval is given

Parameters	**Confidence Intervals**
k_s	± 0.073
k_{cat}	$\pm 9.7 \times 10^{-5}$
k_p	± 0.064
k_i	± 0.0083
k_{de}	$\pm 4.7 \times 10^{-4}$

Table 2.2: List of confidence intervals for each of parameters in HIV protease model fitted using Nelder-Mead algorithm

by

$$CI = \frac{1.96 * \sigma_k}{\sqrt{k}}, \tag{2.4}$$

where k is the sampled values and σ is the standard deviation. We used the HIV protease model by Kuzmic [72]). This model contains the following set of reactions:

$$M + M \underset{k_{dm}}{\overset{k_{md}}{\rightleftharpoons}} E$$

$$S + E \underset{k_s}{\overset{k_{on}}{\rightleftharpoons}} ES$$

$$ES \xrightarrow{k_{cat}} P + E$$

$$P + E \underset{k_p}{\overset{k_{on}}{\rightleftharpoons}} EP$$

$$I + E \underset{k_i}{\overset{k_{on}}{\rightleftharpoons}} EI$$

$$EI \xrightarrow{k_{de}} EJ$$

Here we have taken $k_{on} = 100$, $k_{dm} = 0.001$, $k_{md} = 0.1$ and allowed the other five rate constants (k_s, k_{cat}, k_p, k_i, and k_{de}) to vary. We used libRoadRunner to perform time-course simulations and compared the results with the experimental data set provided by Kuzmic [72]. For parameter estimation, we utilized the Nelder-Mead algorithm [94] via the

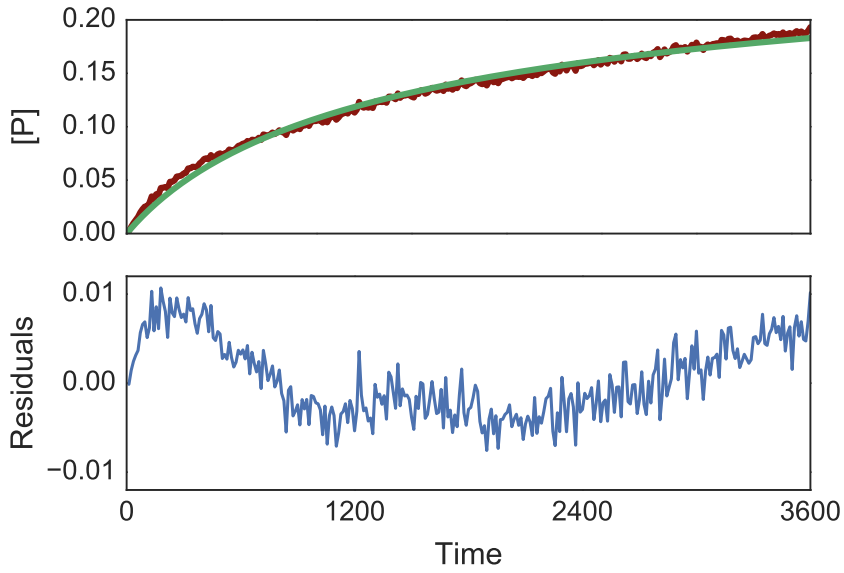


Figure 2.10: Output of parameter estimation on HIV protease data. The upper panel illustrates time-course concentration of product P. The red line represents the raw data used for fitting and the green line represents time-course data simulated through libRoadRunner using the fitted parameters. The lower panel shows the residuals between the raw and fitted line. Note the noticeable trend in the residual, which indicates an issue with the fitted model.

`lmfit` package [95] which provides simple wrapper functions for various optimization methods packaged with SciPy. The initial parameters are obtained from the original paper [72]. Boundaries are set so that the parameter values do not become negative. Bootstrapping was repeated 500 times, which took about 5 minutes (302.4 $\pm$ 6.8 sec) on an Intel i7 4770 machine. Confidence intervals calculated from the Monte Carlo bootstrapping algorithm are listed in Table 2.2. The upper panel in Figure 2.10 shows a typical outcome of parameter estimation on the concentration of product P, where red dots represent experimental data and the green line represents the fitted curve. The residual of the fitting is illustrated in the panel below. It is also possible to check the correlation between fitted parameters, which is illustrated in Figure 2.11.

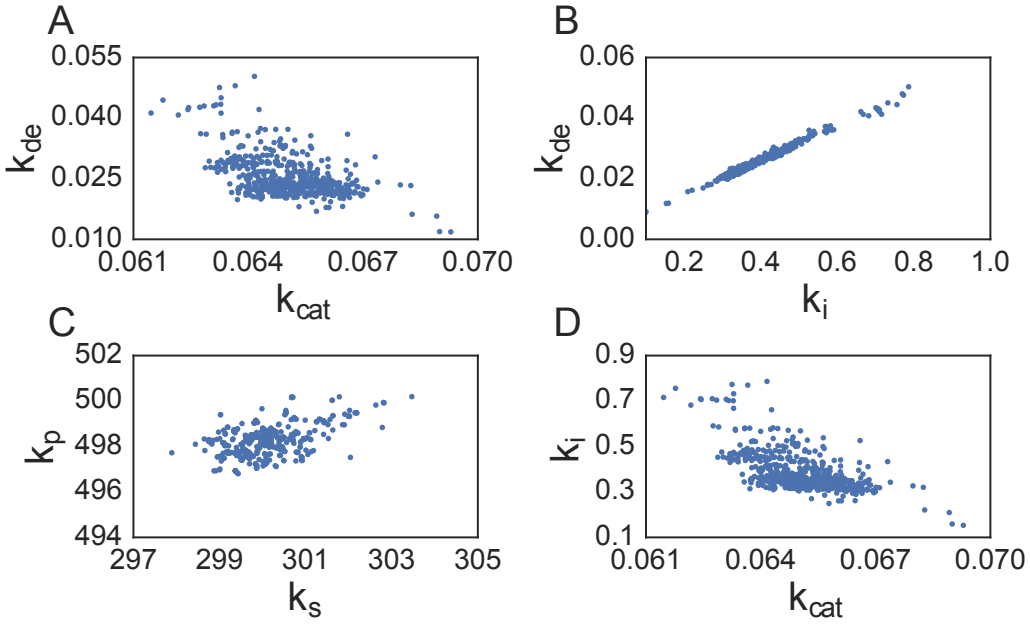


Figure 2.11: Plots showing correlations between rate constants (A) k_{cat} and k_{de}, (B) k_i and k_{de}, (C) k_s and k_p, and (D) k_{cat} and k_i, which are obtained from Monte Carlo bootstrapping.

2.2 Netplotlib

Data visualization is critical for qualitative understanding in a study. Similarly, the visualization of a reaction network helps with a qualitative understanding of the system. Oftentimes, reaction networks are defined using a set of differential equations but visualized with network diagrams. There are several standalone applications for systems biology that does support plotting network diagrams but not a lot of option is are available for Python. Also, combining network diagrams with the output of model analysis provides valuable insight into the system through the connection of the qualitative and quantitative aspects of the model. By doing so, it is possible to compare the quantitative aspects of different models since comparing the qualitative aspects of the model is relatively simple. Netplotlib is a Python package specifically designed to do these types of analyses on a reaction network model through network diagrams.

Netplotlib is a purely Python-based package for visualizing and analyzing reaction network models. Netplotlib supports models encoded in SBML or Antimony. The package features an auto-layout algorithm based on Kamada-Kawai path-length cost-function [65]. Therefore, layout information does not need to be supplied through an SBML file (via SBML layout package). Netplotlib supports visualizing different types of reactions including UNI-UNI, BI-UNI, UNI-BI, and BI-BI. The package also supports visualizing regulations. Netplotlib can detect regulators even if the regulator is not explicitly defined as a regulator in the SBML file by analyzing the rate laws. Netplotlib also comes with several plotting functions designed to visualize fluxes and species rates of change. Furthermore, netplotlib is ideal to plotting multiple network diagrams at once. It supports grid plots and combined network diagrams where unique reactions are recorded and weighted according to the frequency of it appearing throughout the ensemble of models.

2.2.1 Implementation

Netplotlib depends on NetworkX [53] for layout algorithm and matplotlib [59] for plotting. The package has two classes, `Network` and `NetworkEnsemble` which take a model or a list of models, respectively. Once initialized, the classes provide various functions to plot network diagrams. The classes have multiple methods to analyze and customize the network diagram.

Netplotlib is easy to use. The plotting process starts with loading a model to a `Network` class.

```
import netplotlib as npl


AntimonyStr = '''
$Xi -> S1; k0*Xi
S1 -> S2; k1*S1
S2 -> S3; k2*S2
S1 -> S3; k3*S1
```

```
S3 -> $Xo; k4*S3


Xi = 3; Xo = 2
k0 = 0.46; k1 = 0.73; k2 = 0.64;
k3 = 0.51; k4 = 0.22
'''
net = npl.Network(AntimonyStr)
```

`Network` class accepts both SBML and Antimony strings. `Network` provides various customization for the diagram. Everything from label font size to node colors can be customized by setting the properties of a `Network` class.

```
net.fontsize = 15
net.nodeColor = 'tab:blue'
```

Analyses are done by setting corresponding properties. Once the customization is done, simply run `draw()` to generate the network diagram.

```
net.draw()
```

Netplotlib can not only plot reactions but also regulations. Netplotlib does this by intelligently analyzing rate laws to extract regulations. This is done by using SymPy, which is a symbolic math package. Netplotlib determines the existence of regulations and whether the regulations are activatory or inhibitory. The same method is used to determine whether the reaction should be treated as a reversible reaction. Reversible reactions are visualized with edges with arrows on both ends.

`NetworkEnsemble` class is similar to `Network` but accepts a list of models. The idea is to provide a tool to analyze multiple models at once. Currently, there are two plotting functions for `NetworkEnsemble` class. These functions are demonstrated in the next section, where some of the applications of netplotlib are discussed.

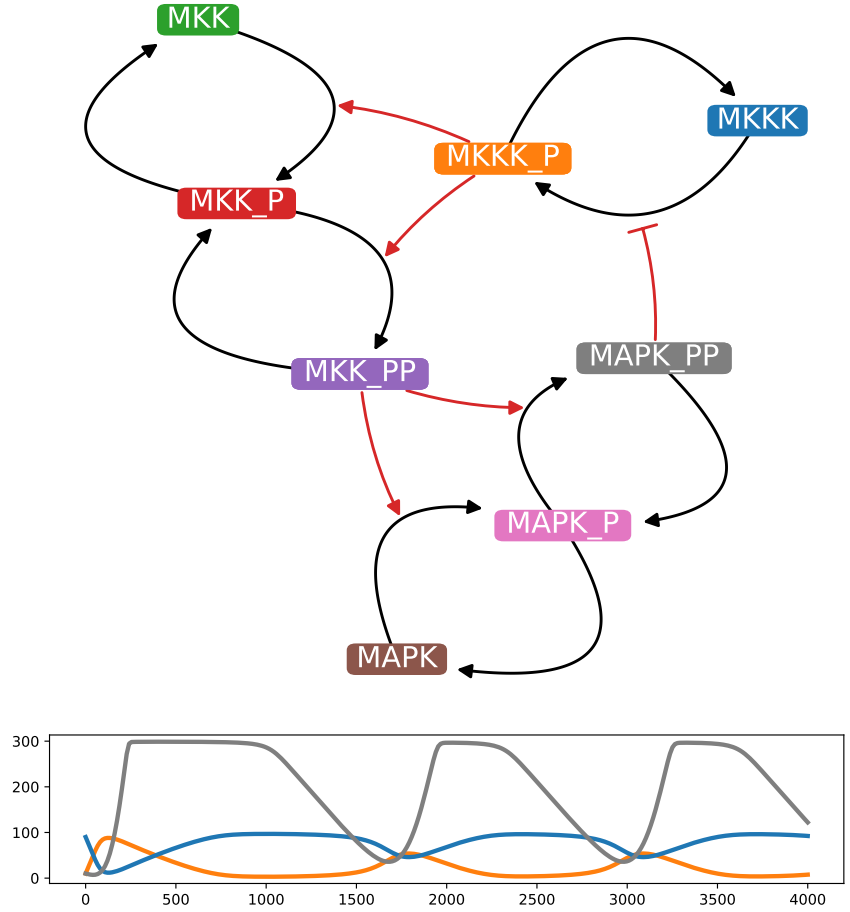


Figure 2.12: A network diagram of MAPK cascade model [68] with inline time-course plot. Species 'MKKK', 'MKKK_P', and 'MAPK_PP' has been specifically selected for the time-course plot. The color of nodes matches the color of time-course traces.

2.2.2 Applications

In this section, several use cases of netplotlib are illustrated which demonstrates some of the features of netplotlib for qualitative understanding of quantitative data.

Inline Time-course Visualization

Netplotlib provides an easy way to plot network diagrams and the results of time-course simulations at the same time. A network diagram is an excellent way to comprehend the dynamics observed in the time-course plot. The package uses consistent colors for species so that the color of nodes matches that of time-course traces. A subset of species can be selected for better clarity of time-course plots. The user can also specify the duration of simulations. Figure 2.12 illustrates A network diagram of MAPK cascade model [68] with inline time-course plot. Species 'MKKK', 'MKKK_P', and 'MAPK_PP' has been selected for time-course plot. The network diagram provides insights on the cycle between 'MKKK' and 'MKKK_P' and inhibition from 'MAPK_PP' which makes the time-course simulation data easier to understand. The total amount of species 'MKKK' and 'MKKK_P' is fixed and higher levels of 'MAPK_PP' leads to lower levels of 'MKKK_P' due to inhibition. Below is an example code that demonstrates this feature which results in Figure 2.12.

```
net = npl.Network(mapkcascade)
net.fontsize = 20
net.drawInlinetime-course = True
net.drawReactionNode = False
net1.inlinetime-courseSelections = ['MKKK_P', 'MKKK', 'MAPK_PP']
net.simTime = 4000
net.draw()
```

Boundary Species Visualization

Many models are written in a way that several reactions share a few boundary species. For examples, many models often use an arbitrary source and sink for production and degradation of species. In SBML, the source and the sink become boundary species. If there are multiple productions or degradation events, SBML will encode this information as multiple reactions to and from the same source and sink. When visualizing these models, network diagrams are

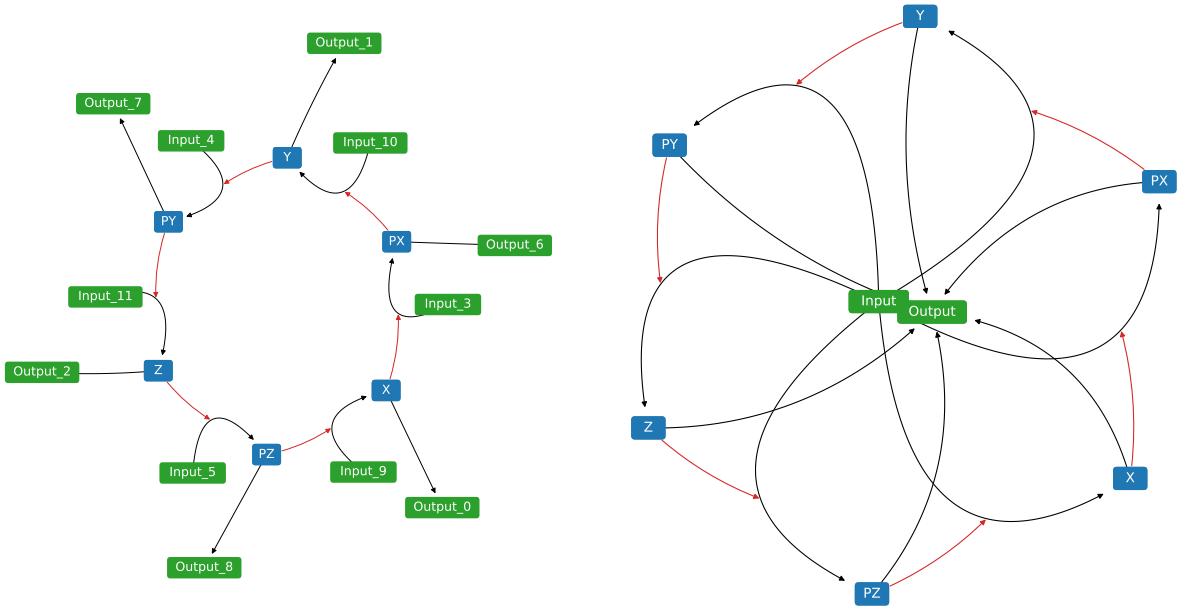


Figure 2.13: Network diagrams of repressilator model [41] with and without separating the boundary species. Visualizing the model after separating shared boundary species makes the diagram easier to understand.

often harder to comprehend because generated layout is not ideal. Figure 2.13 demonstrates how separating the boundary species for each reaction can generate network diagrams that are easier to understand. Figure 2.13 shows the same repressilator model [41] with and without separating the boundary species. The original model uses the same name for the source and the sink for all reactions. The prefix of the names is extracted from the original boundary species to indicate some of the boundary species are actually the same in the model. The code below demonstrates how to separate the boundary species.

```
net = npl.Network(repressilator)
net.scale = 1.5
net.drawReactionNode = False
net.fontsize = 25
```

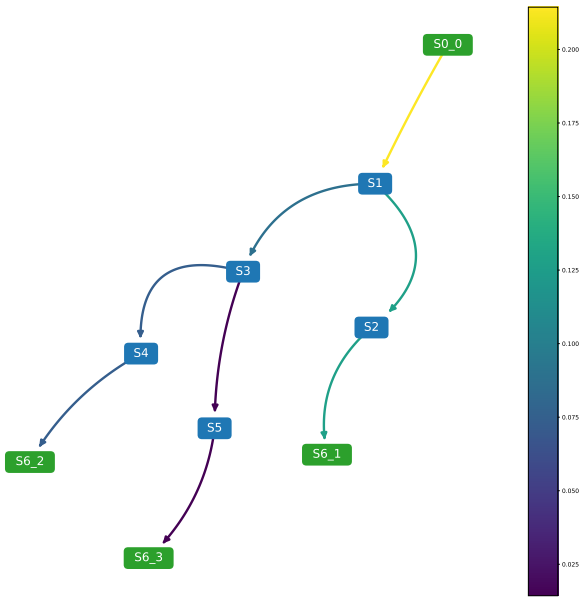


Figure 2.14: A network diagram of simple model of branching networks with flux visualized with colormaps.

```
net3.draw()
net3.breakBoundary = True
net3.draw()
```

Flux Visualization

Fluxes are often sought after for problems involving maximizing or minimizing certain product. It is also a metric that greatly benefits from visualization. Netplotlib can visualize the fluxes of a model. The flux visualization is done by color-coding the edges (reactions). Users can also supply custom colormaps to visualize the flux. Figure 2.14 illustrates fluxes visualization in a simple model of branching pathway. The flow of species can be easily recognized. Note that it is also possible to plot the colorbar corresponding to the supplied colormaps which are scaled to the minimum and maximum values of flux. Below is an example for

generating a network diagram with flux visualized.

```
net = npl.Network(simple_branch)
net.fontsize = 20
net.scale = 1.3
net.edgelw = 3
net.breakBoundary = True
net.drawReactionNode = False
net.analyzeColorScale = True
net.analyzeFlux = True
net.analyzeColorMap = 'viridis'
net.plotColorbar = True
net.draw()
```

Species Rates of Change Visualization

Netplotlib can be used to visualize the rate of change of floating species. The Rate of change visualization is done by color-coding the species nodes. Similar to flux visualization, users can supply custom colormaps to visualize the rate of change of floating species. Figure 2.15 visualizes the rate of change of floating species in the MAPK cascade model [68]. Below is an example for generating a network diagram with the rate of changes of floating species visualized.

```
net = npl.Network(mapkcascade)
net.fontsize = 20
net.drawReactionNode = False
net.analyzeColorScale = True
net.analyzeRates = True
net.analyzeColorMap = 'viridis'
net.simTime = 3000
```

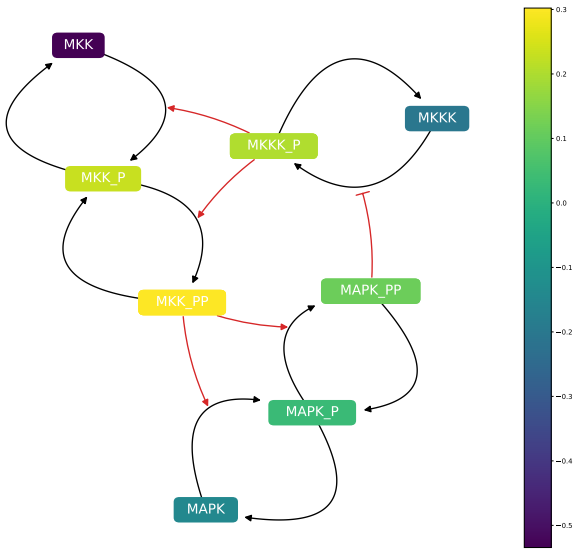


Figure 2.15: A network diagram of MAPK cascade model [68] with species rate of change at $t = 3000$ visualized with colormaps.

```
net.plotColorbar = True
net.draw()
```

Grid Plot

In case there are multiple models to be visualized, netplotlib provides functions to quickly generate a grid plot of network diagrams. Netplotlib has a class called `NetworkEnsemble` which will accept a list of models as the input. Grid plot can be generated via `drawNetworkGrid` function where one can put the desired number of rows and columns. Figure 2.16 shows an example of how the grid plot would look like. The code below demonstrates how grid plot generation can be done in netplotlib.

```
net = npl.NetworkEnsemble(model_list)
net.edgelw = 3
net.fontsize = 25
```

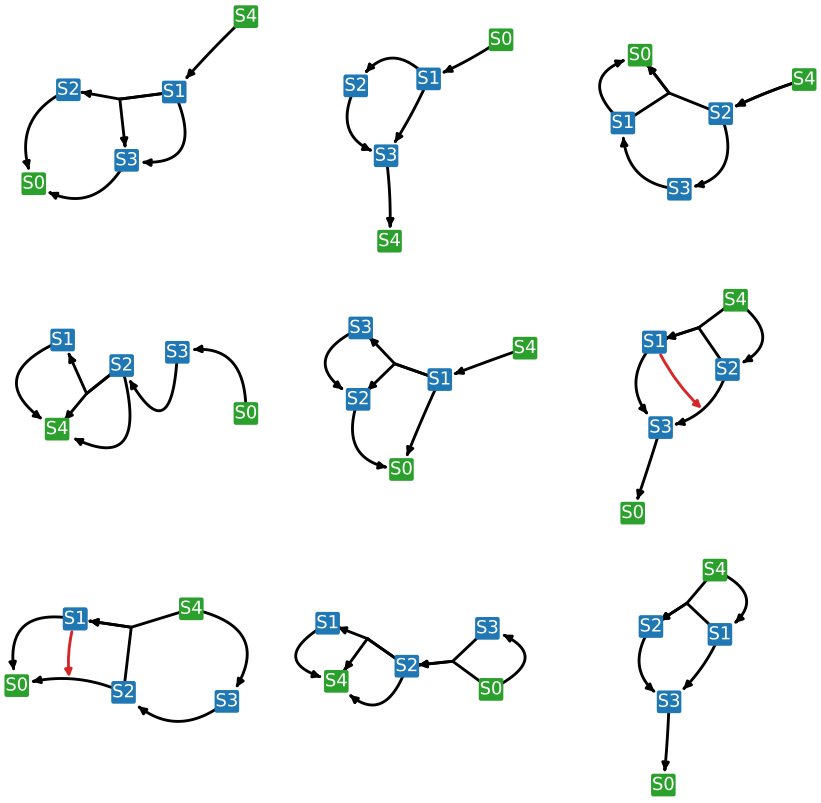
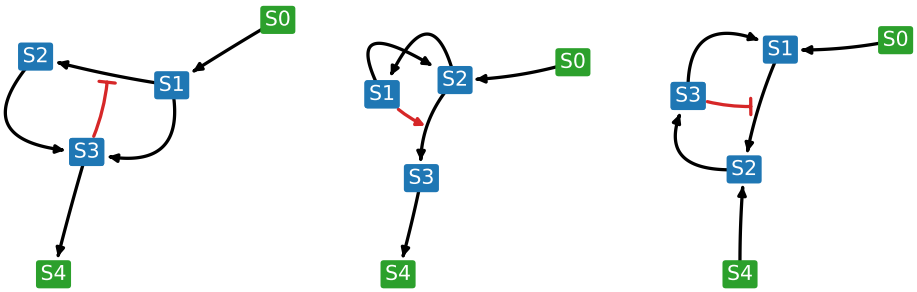


Figure 2.16: A network diagrams of list of models visualized as a grid plot.

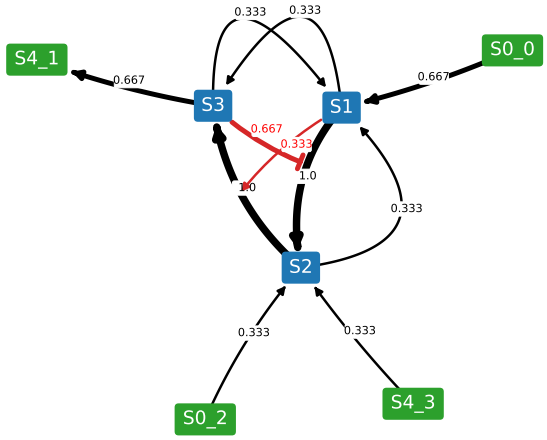
```
net.drawReactionNode = False
net.drawNetworkGrid(3, 3)
```

Weighted Diagram

When dealing with multiple models with common species as in ensemble modeling, model visualization and network analysis can be quite challenging. To help this, netplotlib provides functionalities to combine multiple models into one. This is done by calculating the edge (reaction) frequency and plotting all reactions appearing in the ensemble while weighting the edges accordingly. Consider an ensemble of three models illustrated in Figure 2.17A. These



(A)

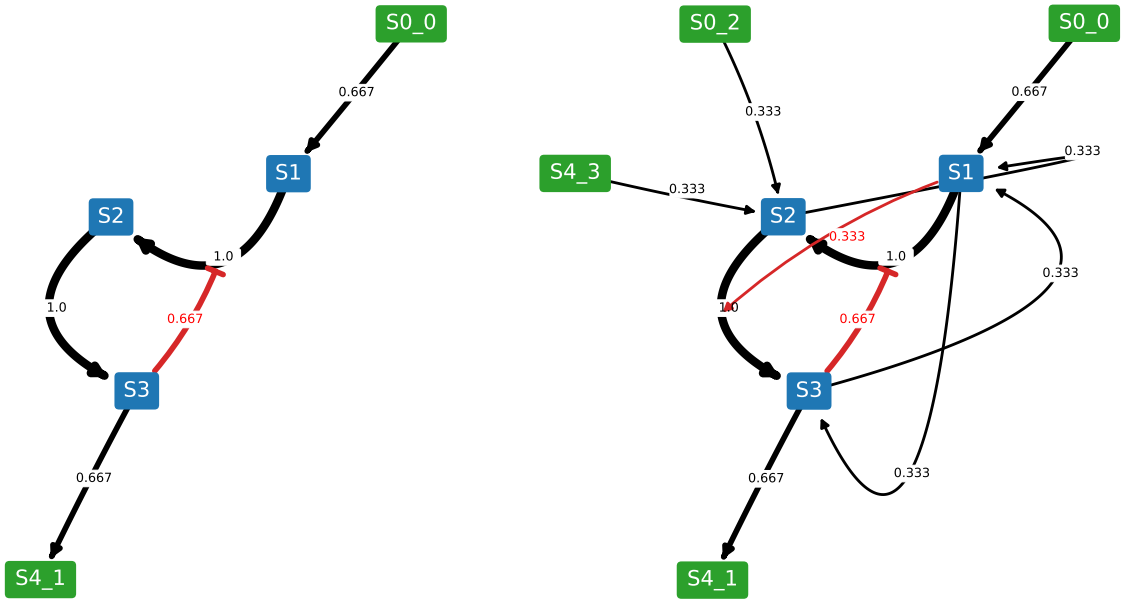


(B)

Figure 2.17: List of models illustrated as (A) a grid of network diagrams and (B) a weighted network diagram.

models can be combined using `drawWeightedDiagram` function which generates Figure 2.17B. Figure 2.17B provides a comprehensive picture of the models in the ensemble, which reactions and regulations are common across the ensemble, and by how much. The value here is if the models in the ensemble share a pattern, the pattern is recognizable through the combined diagram. Later in the manuscript, this will be extensively demonstrated.

Supporting multiple ways to visualize a combined diagram helps with the qualitative



(A) (B)

Figure 2.18: The weighted network diagrams (A) with a threshold and (B) with a threshold but without removing the reactions whose frequency is below the threshold. The network diagrams visualize the same information as Figure 2.17B but with better legibility.

understanding of the model ensemble. Since there is no correct answer when it comes to data visualization, the model should be visualized in several different ways to acquire a comprehensive picture of the system. One useful method when it comes to a complex diagram is reducing the amount of information displayed to the viewers. In many cases, only the most common reactions among the model ensemble are the most relevant to look at. `NetworkEnsemble` provides a way to set a threshold by which only edges with frequencies above the threshold are displayed. Figure 2.18A illustrates the same weighted diagram as Figure 2.17B but with a threshold of 0.5. It is also possible to keep this layout but put back all the reactions below the threshold, which is shown in Figure 2.18B. The width of the edges

are scaled to the frequencies by default, but the transparency of the edges can also be scaled to the frequencies. The code below demonstrates how this can be done in netplotlib.

```
net7 = npl.NetworkEnsemble(model_list)
net7.scale = 1.25
net7.edgelw = 10.
net7.fontsize = 25
net7.edgeLabelFontSize = 15
net7.breakBoundary = True
net7.drawReactionNode = False
net7.drawWeightedDiagram()
net7.removeBelowThreshold = True
net7.plottingThreshold = 0.5
net7.drawWeightedDiagram()
net7.removeBelowThreshold = False
net7.drawWeightedDiagram()
```

Chapter 3

FACILITATING REPRODUCIBILITY IN SYSTEMS AND SYNTHETIC BIOLOGY

The Systems and synthetic biology community has been designing, improving, and applying standards to ensure the exchangeability of computation models. The effort led to various software tools for SBML, support from journals and funding agencies, and adoption from the community in general. However, standards other than SBML are often overlooked, presumably because these standards are relatively new and do not encode information on the biological processes, which is generally considered as the 'model' part of a study.

However, models are not enough to reproduce computational studies. The information on how to run the model is crucial for reproducibility. This information is stored in SED-ML, which is a language that is not well supported by the community as yet. There are several programs that support SED-ML by importing and performing simulation described in a SED-ML file. Other simulators include the SBW Simulation Tool [17] and the Systems Biology Simulation Core Library [67]. Apart from generic XML editors, some others let the user edit or create SED-ML. SED-ML Script [13] defines a script-based language that is closely tied to SED-ML itself: each script function adds a particular SED-ML element to a document, with the arguments of that function setting all of the particular attributes and child elements of that root element. However, no particular attempt is made to hide the complexity or to error-check: all arguments to the script functions are passed as-is to the SED-ML creator, including XPath strings. While simpler than editing the XML directly, this approach leaves a burden on the user to comprehend the details of raw SED-ML.

On the other end of the complexity spectrum, SED-ML creator 'wizard' (`http://sysbioapps.dyndns.org/SED-ML/Web/Tools/`) creates a 'standard' SED-ML file from an uploaded SBML

file with few basic options. Another GUI-based SED-ML creator is SED-ED [2], which lets the user build up a SED-ML file element by element, with error-checking along the way, while providing a graphical overview of the relationships between the elements. It also provides automated methods for creating XPath strings and labeled fields for the necessary SED-ML attributes.

Nonetheless, there is a relatively small number of software tools that support SED-ML and practically none is based on Python. Improving Python support for SED-ML to import, modify, and export SED-ML files would be a valuable contribution to the community.

Another issue is that these standards are rarely transparent. Reproducibility is granted only when the study is both exchangeable and transparent. The standardization effort led by the systems and synthetic biology community addresses the issue of exchangeability, but not the transparency aspect of a study. SBML and SED-ML are designed to be machine readable, but human readability is at the mercy of software developers who might supply tools to help with transparency along with their software. There have been a number of tools developed by the community to address this issue. Antimony [121] language is a human-readable/writable counterpart of SBML. Various visualization tools, such as those integrated to JDesigner [109], provides a qualitative understanding of a model. Efforts towards standardized annotations [92, 93] also improves the transparency by supplying exact meanings of components at various hierarchies.

However, most of these works aim at improving the transparency of the model (SBML) rather than the simulation setup (SED-ML). As a result, SED-ML remains to be hard to comprehend in general. It would be great if the suggested Python tool for importing, modifying, and exporting SED-ML also helps with the transparency of SED-ML.

In this chapter, efforts to build support for SED-ML and SBOL in Python are discussed. The effort led to two SED-ML tools for Python including SED-ML to Python Converter and phraSED-ML, which support importing, modifying, and exporting of SED-ML files and thus allow Python users to control fully SED-ML files. PhraSED-ML is a human-readable/writable counterpart of SED-ML, improving the transparency. Also discussed is

pySBOL, a Python binding for the libSBOL library. All of these tools are included in Tellurium.

3.1 SED-ML to Python Converter

The foundation of a computational standard stems from the development and distribution of software libraries. These software libraries provide application programming interfaces (API) to read, write, and manipulate information encoded in a standardized format. In systems biology, the community have been contributing to maintain such libraries, including libSBML [21] and libSEDML [12]. However, software libraries are only half of the story when it comes to the adoption of a standard. Tools that support these standards are just as important as the software libraries themselves. Without proper tool support, users suffer for limited usability of the standards, leading to insufficient adoption. SBML, being the oldest and widely accepted of the standards discussed here, enjoys extensive third-party support through various softwares and packages [57, 122, 121, 45, 40, 89, 114, 128, 77, 97, 17, 88, 87]. SED-ML, on the other hand, is relatively lacking in this front especially for Python language, which is an issue since Python is increasingly adopted in the scientific community in general.

To facilitate the adoption of SED-ML in Python, one of the most basic functionalities one would need is a proper importing and exporting tools. In SBML, for example, such feature would let user load in models written in SBML to a simulator without manually reading and entering numbers from an SBML file. There are several programs that support SED-ML in such manner, including the SBW Simulation Tool [17] and the Systems Biology Simulation Core Library [67]. However, there is currently no Python-based software that does this, and manually reading and translating SED-ML is not trivial as the simulation setup becomes more complex. This is also problematic for reproducibility because as a result, the information encoded in a SED-ML file is not transparent. As a solution to the problem, I contributed to developing a SED-ML to Python Converter which takes raw SED-ML file or string and automatically generates a Python script compatible with our libRoadRunner simulator [122] and Tellurium simulation environment [31, 84]. The resulting script can be

directly executed to generate the desired outputs specified in the SED-ML file, making SED-ML files actually useful in Python. Not only that, the Python script is generally easier to read, write, and modify, providing better transparency and potential for reuse.

As stated before, SED-ML specification is largely divided into five separate classes: `listOfModels`, which describes models to be used; `listOfSimulations`, which describes the algorithms to be executed; `listOfTasks`, which links a model to a simulation; `listOfDataGenerators`, which captures raw simulation outputs and applies post-processing if defined; `listOfOutputs`, which describes how the output should be presented. SED-ML to Python Converter checks and translates each of these classes into Python while conserving the details by pointing out each step. This is done to add minor readability to the translated script where a user might want to compare to result with the original SED-ML file. For example, information encoded under `listOfDataGenerators` might be of no specific use if there is no specified post-processing step as simulators return array and Python by default supports basic array manipulation routines. However, SED-ML to Python Converter will create a section dedicated to `DataGenerators` objects regardless. SED-ML to Python Converter supports most of the features listed in SED-ML version 1 level 2 specification, except XML modification features and certain algorithms that our default simulator does not support.

As an example, I present following snippets of SED-ML files and their corresponding translations. A typical SED-ML file for `listOfModels` might look like this:

```
<listOfModels>
    <model id="modelId" language="urn:sedml:language:sbml" source="
        ↪ simpleModel.xml" />
</listOfModels>
```

The `listOfModels` class in this case simply points to an SBML file that should be used for the simulation. The SED-ML to Python Converter generates a Tellurium function called `loadSBMLModel` which looks for the model.

```
modelId = te.loadSBMLModel(os.path.join(workingDir, 'simpleModel.xml'))
```

The `listOfSimulations` class wants to run a `uniformtime-course` simulation on this model from time zero to hundred with hundred steps.

```
<listOfSimulations>
    <uniformtime-course id="simId" initialTime="0.0" outputStartTime="0.0"
        ↪ outputEndTime="100.0" numberOfPoints="100">
    </uniformtime-course>
</listOfSimulations>
```

The SED-ML to Python Converter generates a Tellurium function called `simulate` which accepts initial time, end time and the number of steps as keyword arguments.

```
task1[0] = modelId.simulate(start=0.0, end=100.0, steps=100)
```

The full SED-ML file used for this section is available in Appendix A. The full translated Python script is available in Appendix B.

3.2 phraSED-ML

Translation from SED-ML to Python is not difficult but the opposite is not true. Exporting Python script to SED-ML has been proven challenging because Python is a programming language that is imperative. This means that 1) Python script, when executed, will execute line by line from top to bottom, and 2) a state of a variable can change at any time. Consider a simple example demonstrating these characteristics.

```
A = 1
print('A is ' + str(A))


A = 10
print('A is now ' + str(A))
```

Once executed, the output will look like this:

```
A is 1
A is now 10
```

This means the order of which the line is executed affects the state of a variable. Not only that, Python supports other programming paradigms, such as functional programming and object-oriented programming. This means that there are numerous ways to code a Python script to do the same thing. All of these characteristics of Python language are problematic when trying to export to SED-ML, which is a declarative, static language. Unless one design an environment that tracks what is being executed in which order, automatic conversion from a Python script to a SED-ML file is difficult. And if one thing that the snippets of SED-ML in the previous section have proved is that SED-ML is not a language that should be written by humans. Therefore, I have decided to design a human-readable/writable language to be used in Python to easily translate to and from SED-ML.

Here, I introduce phraSED-ML [32], a paraphrased, human-readable adaptation of SED-ML. PhraSED-ML is conceptually similar to Antimony [121], which is a text-based language to define the models. We distribute libraries for the language to integrate it with any other simulation software that can understand SED-ML, providing the users with a new option for SED-ML creation and exploration. Python bindings provide a unified scripting environment that can both execute and export their simulation experiment. The cross-platform library for phraSED-ML is written in Bison (`https://www.gnu.org/software/bison`) and C++, with a simple C API, along with Python bindings. It uses libSEDML [12] to parse and create SED-ML files, and libSBML [21] to parse the SBML files to which the SED-ML documents refer. It also uses the check library (`http://libcheck.github.io/check/`) for unit tests. A standalone command-line tool (phrasedml-convert) is provided for easy conversion between phraSED-ML and SED-ML. Source code, executables, libraries, and documentation are available at `http://phrasedml.sf.net`. From now on, I describe the implementation of phraSED-ML and discuss a few examples demonstrating the usefulness

of phraSED-ML.

3.2.1 Simple Abstractions

The phraSED-ML language is designed to be easily readable and writable without the need to reference the documentation at each turn. The format of each line is declarative, and because the subject of SED-ML is the act of simulation, all keywords are verbs. Model objects are declared with the keyword 'model', simulations with the keyword 'simulate', tasks with the keyword 'run', repeated tasks with the keyword 'repeat', and output with the keyword 'plot' and 'report'. The `DataGenerator` object is abstracted away entirely - objects are created as needed according to user directives parsed in the requested outputs. The following illustrates a typical phraSED-ML code.

```
mod1 = model "sbml_model.xml"
sim1 = simulate uniform(0,10,100)
task1 = run sim1 on mod1
plot time vs S1
```

XPath expressions are avoided on the user end by allowing model elements to be referenced by ID alone (e.g. 'S1'), which the `libphrasedml` library translates to an appropriate XPath string. Element attribute values are similarly translated behind the scenes, made possible by the use of `libsbml` library to parse the referenced model to determine which element attribute must be used. This restricts the scope of phraSED-ML somewhat from the broader abilities of SED-ML, which can perform arbitrary XML transformations. However, this restriction was not deemed too onerous, particularly in light of the simplification it offered.

3.2.2 Allowed Complexity

Some of the more advanced features of SED-ML may still be accessed with phraSED-ML. KiSAO terms are not necessary, but if the user wishes to use backward differentiation for-

mula for solving ordinary differential equation (ODE), for example, they may use the 'bdf' keyword:

```
sim1.algorithm = bdf
```

which will be translated to SED-ML as KiSAO id 288. The KiSAO id may also be used directly:

```
sim1.algorithm = kisao.288
```

which is useful for algorithms for which there are no built-in keywords. Similarly, algorithm parameters may be defined either by keyword or KiSAO id. Here, the relative tolerance (KiSAO id 209) is set:

```
sim1.algorithm.relative_tolerance = 0.001
sim1.algorithm.209 = 0.001
```

SED-ML 'repeated tasks' also have phraSED-ML equivalents:

```
task2 = repeat task1 for S1 in [1, 10, 15]
task3 = repeat task1 for S2 in uniform(0,10,100)
```

which set up tasks looping species S1 through a vector of values, and species S2 through a 'uniformRange' of evenly-spaced values.

3.2.3 Examples

Several examples of application involving phraSED-ML are demonstrated below. We use the aforementioned MAPK cascade model [68] as the model of interest for most of the demonstrations in this section.

Simple Time Course Simulation

The simplest example illustrating the use of phraSED-ML is a simple time course simulation. The code below performs a time course simulation on the model from 0 to 4000 with 1000

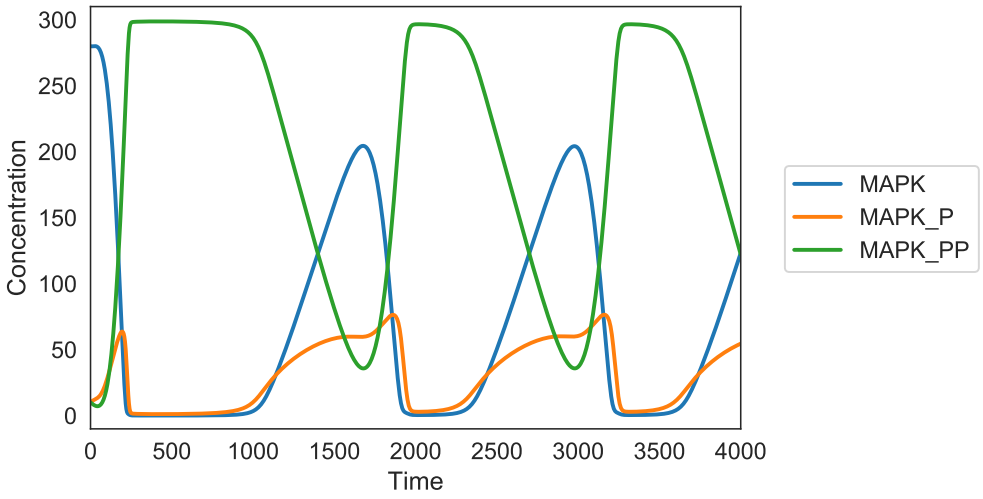


Figure 3.1: Output of a simple time course simulation. Blue line represents MAP kinase, red line represents phosphorylated MAP kinase, and green line represents double phosphorylated MAP kinase.

time points, and plots time versus MAP kinase, phosphorylated MAP kinase, and double phosphorylated MAP kinase as the output.

Listing 3.1: Simple Time Course, see Figure 3.1

```
model1 = model "MAPKcascade"
sim1 = simulate uniform(0,4000,1000)
task1 = run sim1 on model1
plot task1.time vs task1.MAPK, task1.MAPK_P, task1.MAPK_PP
```

The output of the phraSED-ML string is shown in Figure 3.1.

Phase Portrait

The following example shows how it is possible to specify a simple phase portrait. In this case, we run a simulation of the Lorenz attractor [78] which under certain parameter values

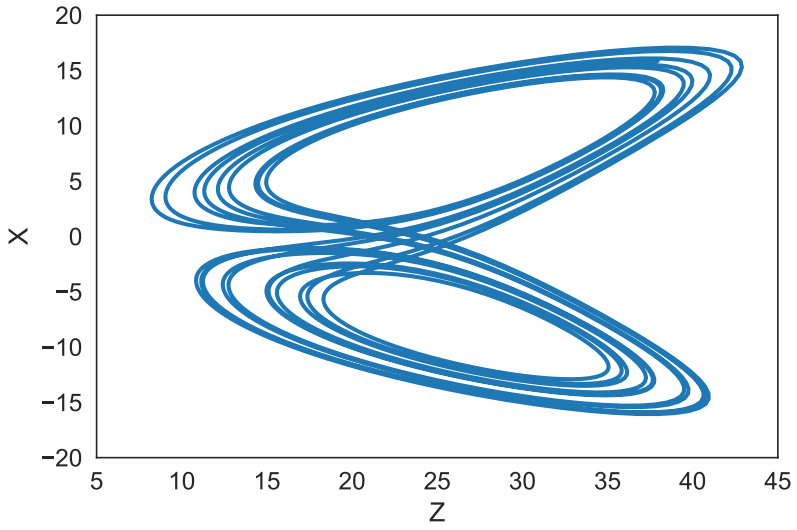


Figure 3.2: Phase plot of Lorenz attractor resulted from running the phraSED-ML code in listing 3.2

exhibits chaotic behavior. In Figure 3.2 we plot the variable z versus x.

Listing 3.2: Lorenz Attractor Phase Plot, see Figure 3.2

```
model1 = model "lorenz"
sim1 = simulate uniform(0, 15, 2000)
task1 = run sim1 on model1
plot task1.z vs task1.x
```

1-Dimensional Parameter Scan

Running parameter scan is simple and intuitive using phraSED-ML by using 'repeat'. The example below runs a 1-Dimensional parameter scan on parameter 'J1_KK2' with values 1, 10, and 100, while resetting the model back to initial condition every time. The output is shown in Figure 3.3.

Listing 3.3: 1-Dimensional Parameter Scan, see Figure 3.3

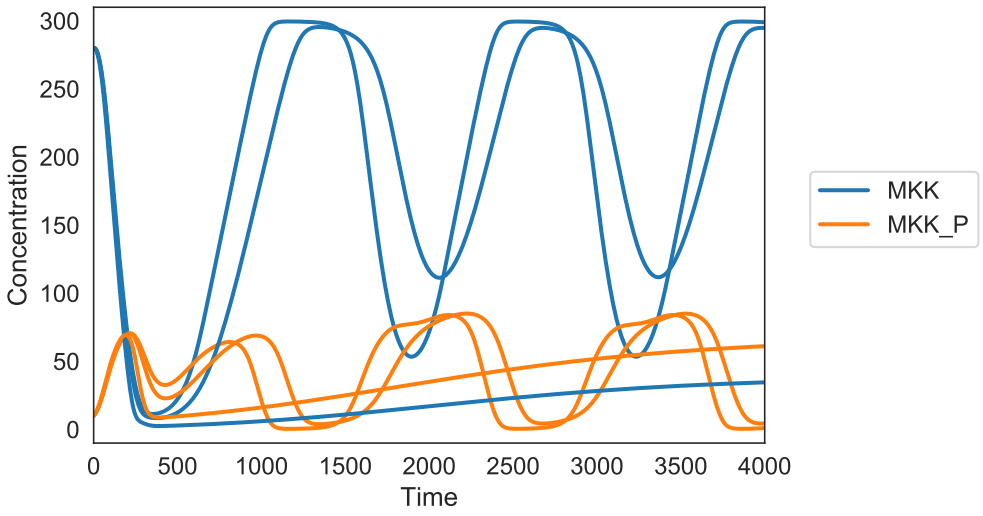


Figure 3.3: Typical output of phraSED-ML string running 1-dimensional parameter scan. The blue lines represent MAP kinase kinase and red lines represent phosphorylated MAP kinase kinase.

```
model1 = model "MAPKcascade"
sim1 = simulate uniform(0,4000,1000)
task1 = run sim1 on model1
repeat1 = repeat task1 for J1_KK2 in [1, 10, 100], reset=true
plot repeat1.time vs repeat1.MKK, repeat1.MKK_P
```

Multi-Dimensional Parameter Scan

Expanding from 1-Dimensional parameter scan, parameter scan on two different parameters can be achieved through repeats of 'repeat'. The code below illustrates the 2-Dimensional parameter scan where the parameter 'J1_KK1' is varied as before while parameter 'J4_KK5' is changed from 0 to 100 in 10 uniform steps. The output is plotted on Figure 3.4.

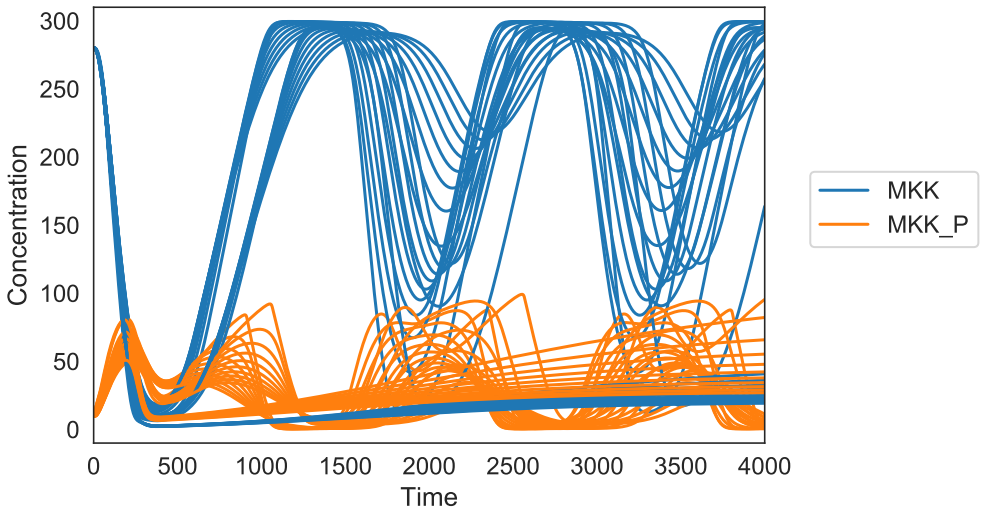


Figure 3.4: Typical output of phraSED-ML string running 2-dimensional parameter scan. The blue lines represent MAP kinase kinase and red lines represent phosphorylated MAP kinase kinase.

Listing 3.4: Multi-Dimensional Parameter Scan, see Figure 3.4

```
model1 = model "MAPKcascade"
sim1 = simulate uniform(0,4000,1000)
task1 = run sim1 on model1
repeat1 = repeat task1 for J1_KK2 in [1, 10, 100], reset=true
repeat2 = repeat repeat1 for J4_KK5 in uniform(1, 100, 10), reset=true
plot repeat2.time vs repeat2.MKK, repeat2.MKK_P
```

Repeated Stochastic Simulations

Another application of phraSED-ML shows how to run repeated stochastic simulations on models. The following phraSED-ML string demonstrates the differences between stochastic simulations with and without a given seed. Typical output of the simulation setup is shown

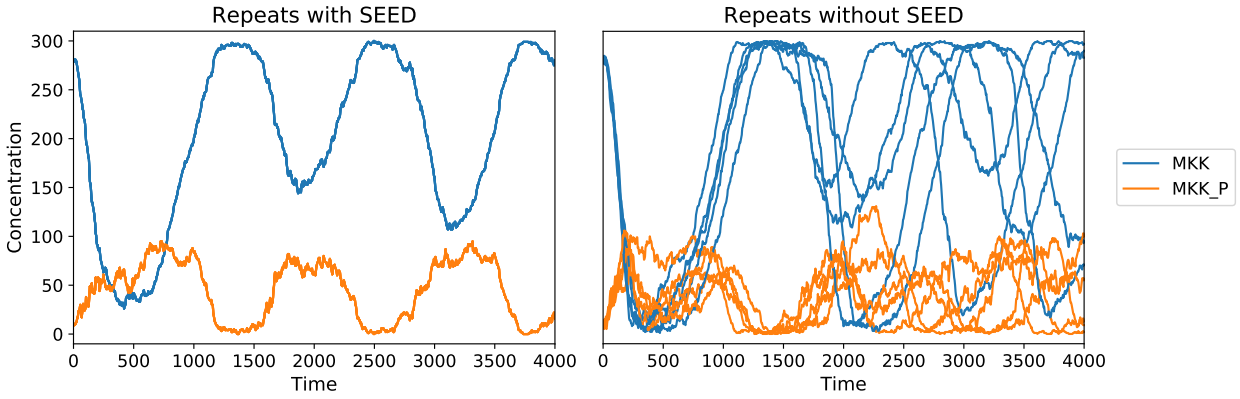


Figure 3.5: The same plot as Figure 3.3 but using the stochastic Gillespie algorithm. The blue lines represent MAP kinase kinase and red lines represent phosphorylated MAP kinase kinase. The left panel shows stochastic simulations with a single seed. The right panel shows stochastic simulations with varying seeds.

in Figure 3.5.

Listing 3.5: Repeated Stochastic Simulations, see Figure 3.5

```
model1 = model "MAPKcascade"
time-course1 = simulate uniform_stochastic(0, 4000, 1000)
time-course1.algorithm.seed = 1003
time-course1.algorithm.variable_step_size = false
time-course2 = simulate uniform_stochastic(0, 4000, 1000)
time-course2.algorithm.variable_step_size = false
task1 = run time-course1 on model1
task2 = run time-course2 on model1
repeat1 = repeat task1 for local.x in uniform(0, 5, 5), reset=true
repeat2 = repeat task2 for local.x in uniform(0, 5, 5), reset=true
plot "Repeats with SEED" repeat1.time vs repeat1.MKK, repeat1.MKK_P
plot "Repeats without SEED" repeat2.time vs repeat2.MKK, repeat2.MKK_P
```

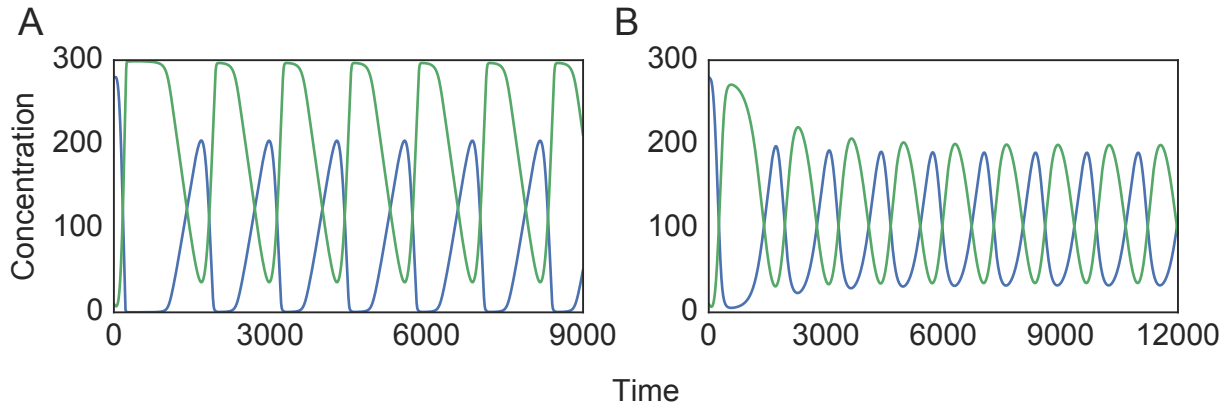


Figure 3.6: Time-course simulation of MAPK and bi-phosphorylated MAPK concentration reproduced from (A) the original COMBINE archive, and (B) the modified combine archive. Blue lines represent the concentration of MAPK and green lines represent the level of bi-phosphorylated MAPK.

3.3 COMBINE Archive Support in Tellurium

A COMBINE archive is a ZIP-based container specifically designed to facilitate reproducibility of computational studies [14]. A COMBINE archive contains all information necessary to reproduce the output such as the model (SBML), simulation experiment (SED-ML), raw data, figures, etc. Now that the support for SED-ML in Python and Tellurium is available, the next step is to expand our support to the COMBINE archive.

Tellurium supports importing and exporting COMBINE archives through a custom class that takes the model and the simulation experiment defined in Antimony and phraSED-ML languages, respectively. To generate a COMBINE archive, first, define a model and a simulation setup in the Antimony and phraSED-ML language. Then an inline OMEX can be created by joining the two string blocks.

```
import tellurium as te
import tempfile, os
```

```
antimony_str = '''
model myModel
    J1: $Xo -> S1; k1*Xo - k2*S1;
    J2: S1 -> S2; k3*S1 - k4*S2;
    J3: S2 -> S3; k5*S2 - k6*S3;
    J4: S3 -> S4; k7*S3 - k8*S4;
    J5: S4 -> S5; k9*S4 - k10*S5;
    J6: S5 -> ; k11*S5;

    Xo = 1.0; S1 = 0.3; S2 = 0.1;
    S3 = 0.2; S4 = 0.1; S5 = 0.2;
    k1 = 3.92; k2 = 2.83; k3 = 0.8;
    k4 = 0.24; k5 = 0.68; k6 = 0.35;
    k7 = 0.82; k8 = 0.47; k9 = 0.37;
    k10 = 0.22; k11 = 0.1;
end
'''

phrasedml_str = '''
  model1 = model "myModel"
  sim1 = simulate uniform(0, 30, 100)
  task1 = run sim1 on model1
  plot "Figure 1" time vs S1, S2, S3, S4, S5
'''
inline_omex = '\n'.join([antimony_str, phrasedml_str])
```

This setup can be directly executed in Python to check the output using `executeInli-`

`neOmex`, which will be identical to the plot shown in Figure 2.3.

```
te.executeInlineOmex(inline_omex)
```

To export an inline OMEX setup, use `exportInlineOmex` function. The code below exports an inline OMEX setup as a COMBINE archive using `exportInlineOmex` to a temporary directory.

```
wDir = tempfile.mkdtemp(suffix="_omex")
te.exportInlineOmex(inline_omex, os.path.join(wDir, 'archive.omex'))
```

Once exported, the COMBINE archive can be re-imported and executed which will reproduce the result.

```
te.executeCombineArchive(os.path.join(wDir, 'archive.omex'))
```

As one other example, the MAPK cascade model [68] is used to demonstrate how easy it is to modify the content of the COMBINE archive. Through the combination of Antimony and phraSED-ML, one can make direct changes to the SBML and SED-ML files embedded in the COMBINE archive. Here, Tellurium is used to generate a different parameterization of the model that reproduces another figure from the original paper [68]. Figure 3.6A illustrates the output of the original COMBINE archive and Figure 3.6B illustrates the output of the modified COMBINE archive. Both figures are presented in the original paper and through the use of reproducibility tools integrated to Tellurium, it is possible to reproduce the original figures. The example illustrates how Tellurium's workflow incorporates automated handling of the exchange of reproducible simulations. The full script is available in Appendix C. The changed parameters and values are listed in Table 3.1.

While it is possible to import COMBINE archives through the use of Python script, Tellurium provides simple GUI-based plug-ins to open a COMBINE archive as well. In Tellurium Spyder, there are options to import SED-ML files and COMBINE archives to Python scripts or phraSED-ML scripts. When there are multiple SED-ML files in a single COMBINE archive, Tellurium will generate multiple scripts corresponding to each and every

Parameters	**Values**
Time	12000
K_i	18
n	2
K_1	50
KK_2	40
KK_3	100
KK_4	100
KK_5	100
KK_6	100
KK_7	100
KK_8	100
V_9	1.25
KK_9	100
V_{10}	1.25
KK_{10}	100

Table 3.1: List of parameters and its changed values in MAPK cascade model.

SED-ML files.

Now that SED-ML to Python converter and phraSED-ML are in place, Python supports all standards necessary to reproduce an output of a simulation experiment performed in the field of systems biology through our Tellurium environment. Tellurium is a valuable tool for the Python community and anyone who wish to reproduce a simulation study done by others. Tellurium encourages reproducible studies by providing an easy way to export a simulation study in a standardized format.

3.4 pySBOL

SBOL [47, 46, 33, 9] is a standard for encoding synthetic designs, specifically designed with exchangeability and reproducibility in mind. SBOL encodes information about the genetic sequence, the components, and their interactions. SBOL also supports hierarchical designs to enable researchers to collaboratively compose genetic elements into entire synthetic organisms. SBOL enables reuse of parts and designs through support from biological parts repositories and design software tools. The language also supports design-build-test-learn workflows for research and engineering purposes.

To use files encoded in SBOL, software library is necessary. A software library allows users to read, write and modify a file encoded in a specific format. SBOL had a software library support for C/C++ called libSBOL, but no software library support was available for Python. To let Python users use existing parts encoded in SBOL and automate designs through SBOL, we designed Python binding called pySBOL [8] for libSBOL. PySBOL provides Python API to use functions built in libSBOL, such as reading and writing SBOL files, handling SBOL objects, etc. The library works with SBOL 1, SBOL 2, GenBank, and FASTA formats. PySBOL is available for Python 2.7 and Python 3.6 and can be installed using pip.

```
pip install pysbol
```

PySBOL is available under open source Apache 2.0 license. The availability of pySBOL complements other programming language support for SBOL, including libSBOL (C/C++), libSBOLj (Java) [133], and sboljs (Javascript) [83]. PySBOL is distributed with Tellurium by default, ensuring exchangeability and reproducibility of synthetic designs and facilitating synthetic design process in Python.

Chapter 4

DESIGNING NOVEL ALGORITHMS FOR ROBUST, RELIABLE MODELS

One of the major goals of systems biology is to construct robust and reliable models. To build a robust, reliable model, one must go through an extensive validation process. However, systems biology today is heading towards multi-scale modeling, building larger and more complex models [91]. As the size and complexity of a model grow, validation becomes more and more difficult. The traditional approach of cycling through designing and testing to build a model can become inefficient as a result.

Thankfully, systems biology have witnessed significant advancements in both experimental and computational techniques. Advancements in proteomics, metabolomics, and genomics provide a large amount of specific, high-resolution data. Scientific computing in general has also enjoyed substantial growth in computational resources. Developments in hardware have made high-performance computing more accessible. Commercial clusters provide a significant amount of computing power as well. All these developments have made high-throughput modeling studies for systems biology feasible.

One way to achieve a rapid, coarse-grained reduction of the model search space is through perturbation studies, similar to what was proposed in [79]. With the advent of CRISPR-Cas9, we now have unprecedented control over selective activation/inhibition of specific genes. Utilizing CRISPRa/i, one can selectively (and combinatorially) perturb the total amount of species or individual reaction kinetics. Coupled with advanced proteomics, CRISPRa/i allows a variety of perturbation studies and having an algorithm that is specifically tailored to these types of data is of interest.

Another area of interest when it comes to the topological search space of a network

is the concept of ensemble modeling. In many cases, lack of knowledge on kinetics and scarcity of experimental data make designing a mechanistic model of a system significantly difficult [71]. In such a case, designing multiple potential candidate models to form a model ensemble is highly desirable for the following reasons: First, it eliminates the necessity for choosing a model out of multiple models all of which perform seemingly well against the data. When multiple models can fit given data equally well, one often faces a dilemma of selecting a model. Second, a model ensemble is more robust than a single model. It has been demonstrated [51, 38, 124] that ensembles as a whole provide better predicting power than individuals in an ensemble. This is true even if the ensemble contains the original topology because a biochemical reaction network model generally has multiple sets of parameters that can result in similarly observed outputs. Third, the ensemble can be analyzed to direct future experiments by identifying the information that can maximally reduce the ensemble. Overall, ensemble modeling can increase the robustness while retaining the reliability [51, 38, 124, 23, 62, 49] and provide a framework to complete the cycle between modeling and experimental efforts.

While ensemble modeling is widely used in various fields of studies such as weather forecasts [100, 104], the concept has also been applied to biological systems such as disordered proteins [125, 80], metabolic pathways [73, 126], and signaling pathways [71]. However, there has been relatively little work done on utilizing perturbation data to generate a model ensemble composed of mechanistic models. Novel algorithms are thus necessary for fitting these specific needs. This chapter presents several algorithms we have designed to aid the modeling endeavor by constructing robust and reliable models. Specifically, two types of algorithm are discussed: one based on qualitative comparison using combinatoric perturbations and the other based on the evolutionary algorithm generating model ensemble using control coefficients. Both types of algorithm utilize perturbation data to collect an ensemble of mechanistic models.

4.1 Network Search Space Reduction

One of the issues in systems biology modeling is that the model search space can be extraordinarily large. Search space for multi-scale models can be especially large, but small models can exhibit the same problem. One must make sure that the topology of the network is correct, that various rate laws and associated regulatory loops for the enzymatic reactions are accurate, and that numerous parameters involved in rate laws are reasonably accurate. When we consider a large scale model which could have more than 60 metabolites and reactions [86], it is evident that modeling efforts with limited prior knowledge of the system can be quite challenging.

Out of all the unknown variables, the network topology is a more important and challenging variable to examine. Until the correct network topology is identified, searching for potential rate laws and parameter values are of little value. At the same time, searching for potential topologies is much more depending. The combinatorics of topologies in general increases superlinearly as the number of nodes grows. Furthermore, topologies are discrete, making it hard to analyze quantitatively and to apply common computational techniques used for continuous variables. For this reason, it would be immensely useful if we could reduce the search space of model topologies to a more manageable scale.

In this section, we present an algorithm that screens models using a minimal amount of qualitative perturbation data and thus collects automatically an ensemble of reliable models [29]. In that way, the algorithm reduces the search space of model topologies. All computations presented in this chapter have been performed through the use of Python. In particular, the Tellurium [31, 84] environment has been used in conjunction with the libRoadRunner solver [122] for model simulations and with the Antimony language [121] for model description.

Before diving into the description of the workflow, let us begin by discussing a suitable data structure for a network model to use for the algorithm.

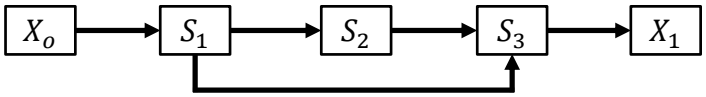


Figure 4.1: Coherent type 1 feed-forward loop (C1-FFL). X_o and X_1 represent the boundary species in the model and are fixed during simulations.

4.1.1 Representation of Network Models

It is important to define how to represent a model for computing purposes, especially when we have limited knowledge of the reaction steps present in a network. In systems biology, a pathway network is usually described by a set of chemical reactions from which a set of ordinary differential equations is derived. The same model will be visualized through a network diagram with arrows representing reactions. A model can have diverse types of motifs ranging from linear chains to dense overlapping regulons (DOR). Reactions can range from simple UNI-UNI [1] reactions to enzyme kinetics.

The problem we would like to solve is how to reduce the model search space and to generate an ensemble of network models based on limited knowledge of the network that is consistent with experimental data. For our solution to be general, it is essential to define a data structure that can describe a network with flexibility enough to account for potentially diverse types of interactions. One of the easiest ways to do this is using matrices where rows represent participants as reactants and columns represent participants as products; a reaction exists between the species specified on the row and column if the value is unity. There are no reactions between the species specified on the row and column if the value is zero. This description is akin to the adjacency matrix used in computer science and the connectivity matrix used in computational neuroscience, except in our case we need also to

[1] UNI-UNI refers to reactions of the type $A \rightarrow B$

take directionality into account (a directed graph). For a simple Coherent type 1 feed-forward loop (C1-FFL) [4] with three floating species and a boundary input/output (Figure 4.1), our description will result in the matrix given by Equation 4.1 (m_{c1ffl}).

$$m_{c1ffl} = \begin{array}{c} \\ X_o \\ X_1 \\ S_1 \\ S_2 \\ S_3 \end{array} \begin{array}{c} \begin{array}{ccccc} X_o & X_1 & S_1 & S_2 & S_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{4.1}$$

The above description is sufficient to describe the majority of network motifs with reversible/irreversible reactions, but there are certain types of reactions that may not be easily expressed in this manner. We can expand the proposed notation farther to incorporate more complex dynamics one might see in, e.g., nonlinear kinetics found in enzyme-catalyzed reactions. One way to approach this is to append the matrix with combinations of individual species. This does increase the computational complexity of the problem. However, if we limit the scope of searches to BI-BI [2] reactions at maximum, which is a reasonable restriction to impose on many systems, the increase in dimensionality might be acceptable, as we need to add a combination of selecting two species out of n total species, $C(n, 2)$, to the number of rows and columns. The total number of combinations of r samples out of n objects is given by

$$C(n, r) = \frac{n!}{r!(n-r)!} \tag{4.2}$$

From the equation, with $r = 2$, it is evident that when the total number of species increases by one, only n rows and columns are added.

For example, consider a moderate-sized network with 10 species in total. In this case, our matrix will be a square matrix with 55 rows and columns (10 species and 45 combinations).

[2] BI-BI refers to reactions of the type $A + B \rightarrow C + D$

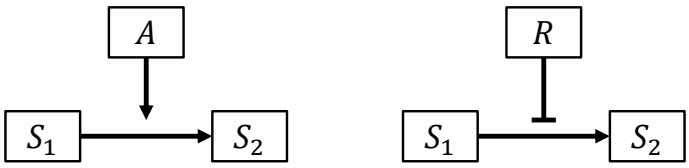


Figure 4.2: Simplest cases of enzymatic activation by activator A and repression by repressor R.

It is possible to add an additional row/column to consider production/degradation (one to the number of rows/columns) of species as well. Something to consider when defining the model is that the scope of a model is arbitrary. If there are well-defined inputs and outputs, or if the system has a steady-state solution, it is entirely possible to break down a large model into small closed systems to apply computational techniques.

The proposed notation is advantageous because it can represent enzymatic reactions with only minor changes. Consider the simplest examples of enzymatic activation and repression shown in Figure 4.2. These reactions can be expressed as a variation of the BI-BI reaction shown below:

$$\mathrm{S}_1 + \mathrm{X} \longrightarrow \mathrm{S}_2 + \mathrm{X} \tag{4.3}$$

However, in order to account for both enzymatic activation and repression, we cannot rely on the binary system (composed of 0s and 1s) where 0 will denote no reaction and 1 will denote activation. Thus, it is inevitable to introduce an additional state to our network representation. The additional state, with the value of -1, will represent enzymatic repression. Then, we can represent enzymatic activation and repression shown in Figure 4.2 as the

following set of matrices:

$$
\begin{array}{c}
\\
A \\
S_1 \\
S_2 \\
A+S_1 \\
A+S_2 \\
S_1+S_2
\end{array}
\begin{array}{c}
\begin{array}{cccccc} A & S_1 & S_2 & A+S_1 & A+S_2 & S_1+S_2 \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{1} & 0 \\
0 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\
0 & 0 & 0 & 0 & \mathbf{0} & 0
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c}
\\
R \\
S_1 \\
S_2 \\
R+S_1 \\
R+S_2 \\
S_1+S_2
\end{array}
\begin{array}{c}
\begin{array}{cccccc} R & S_1 & S_2 & R+S_1 & R+S_2 & S_1+S_2 \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathbf{-1} & 0 \\
0 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\
0 & 0 & 0 & 0 & \mathbf{0} & 0
\end{pmatrix}
\end{array}
\tag{4.4}
$$

Note that a reaction defined in the matrix will be interpreted as an enzymatic activation or repression only when one of the reactants are also present in the products. In the example above, both the reactant and the product contain either an activator or a repressor. This means that only when a reaction with states $+1$ or -1 defined in non-diagonal and non-antidiagonal (similar to a diagonal but runs from top right to bottom left) of BI-BI specific quadrant of the network matrix (bottom right) will be interpreted as an enzymatic activation or repression. In the matrix given by Equation 4.4, bold entries indicate which reactions are treated as enzymatic activation or repression. This also means that where the additional state of -1 can be introduced is limited. (-1) state can only be present in a UNI-UNI reaction (as a type of auto-regulation) or in a BI-BI reaction (as enzymatic repression). This way, we can express diverse types of motifs and reactions through the network matrix.

This is very useful for representing complex signaling cascades. For example, consider a simple example shown in Figure 4.3. This model can be represented by a 10×10 square matrix $m_{cascade}$ shown in Equation 4.5 when using mass action kinetics with boundary signal input S_o integrated into the reaction between species S_1 and S_2 (which is possible since boundary input S_o is fixed) for the purpose of simplification.

Now that we have a concrete structure to represent a network model, we introduce a quick and simple algorithm to reduce the model search space and collect an ensemble of

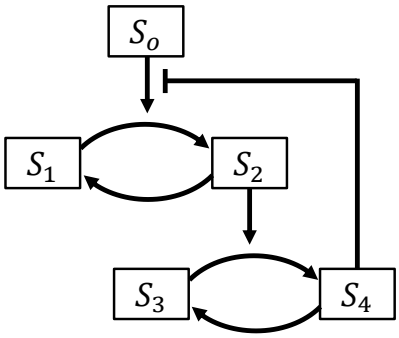


Figure 4.3: A simple cascade model involving four floating species. Species S_2 activates the reaction from species S_3 to S_4. Species S_4 inhibits activation by the boundary input S_o.

network models that are always consistent with perturbation data.

$$m_{cascade} = \begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_1+S_2 \\ S_1+S_3 \\ S_1+S_4 \\ S_2+S_3 \\ S_2+S_4 \\ S_3+S_4 \end{array} \begin{array}{c} \begin{array}{cccccccccc} S_1 & S_2 & S_3 & S_4 & S_1+S_2 & S_1+S_4 & S_1+S_4 & S_2+S_3 & S_2+S_4 & S_3+S_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{4.5}$$

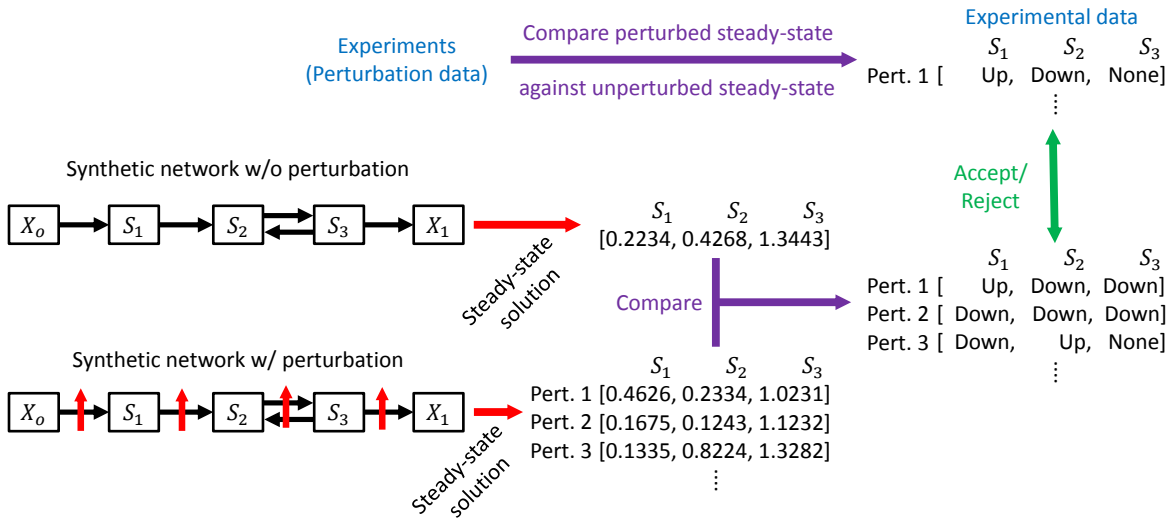


Figure 4.4: Illustration of the network reduction technique. Perturbation data are compared with each other to create an array of trileans; similar steps are taken for synthetic networks, with steady-state solutions calculated in the presence/absence of perturbations. A synthetic network will be accepted if and only if the array of trileans match with that of experimental results. Sets of trileans for combinations of perturbations could be compared if the corresponding experiment should be performed.

4.1.2 Network Reduction Technique

Now that we have a way to represent a model, we describe a method to reduce the search space using perturbation data. Perturbation data contain steady-state solutions of the network in question with and without a specified perturbation on certain reactions or species. We can then compare the steady-state solution in the perturbed state with the solution in the unperturbed state for each and every floating species. One can create an array of three-valued logic (trilean), i.e. $+1$ when the steady-state solution in the presence of perturbation is higher than that in the absence of perturbation, -1 when the steady-state solution in the presence of perturbation is lower than that in the absence, and 0 when the difference in

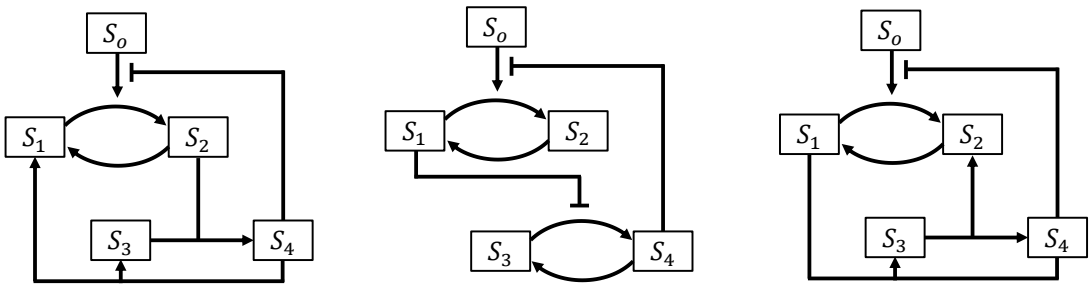


Figure 4.5: Examples of various networks that have survived the selection process. Only the reaction between species S_1 and S_2, completing with repression from species S_4, has been given. In all cases, perturbing the reaction between species S_1 and S_2 results in qualitatively similar steady-state floating species responses. Only mass action kinetics has been used.

steady-state solutions in the presence and absence of perturbation is smaller than a predefined threshold. One thing to keep in mind is that techniques such as CRISPRa/i allow one to perturb combinations of reactions extracting extra information about the network if there are two or more targets that can be perturbed.

Once arrays of trileans are obtained, random synthetic networks can be generated through the use of the proposed network representation method above. An important feature that is necessary for this step is preserving known information, i.e. keeping reactions that are experimentally perturbed and any other reactions known to exist in the network of interest in each and every randomly generated synthetic network. In that way, it is possible to compare the results from the real network and the synthetic network. Known information includes information on any reactions that are known not only to exist but also to be non-existent, as well as information on rate constants and initial species amounts. Knowledge of non-existent reactions is very helpful in removing unwanted reactions which will reduce the search space significantly. After preserving known reactions (and non-reactions) in the network matrix, we explore other reactions, assigning randomly various states to the empty spots in the network

matrix. We also enforce rules against meaningless and nonsensical solutions when generating random synthetic networks: First of all, we make sure all species in the network are involved in at least one reaction, removing incomplete networks. In addition, direct reactions between input and output boundary species are not allowed. Finally, input boundary species remain as inputs and output boundary species remain as outputs.

Once a random synthetic network is generated under these rules, we calculate the steady-state solutions in the presence/absence of perturbations on known reactions to create another set of arrays of trileans. Combinations of perturbations may be applied to the network if the same is done experimentally. Finally, trileans obtained via experimental measurement can be compared with trileans obtained from the synthetic network. The synthetic network will be accepted if and only if the arrays of trileans are identical, and discarded otherwise. At this point, a single iteration is completed and the process starting from the random network generation is repeated. Figure 4.4 illustrates the general workflow of the network reduction technique.

So how well does this technique work? Consider a simple C1-FFL model shown in Figure 4.1. Suppose that a single reaction between species S_1 and S_3 is known to exist, which is the minimal information necessary for using this technique. 10,000 iterations comparing unique and randomly generated networks result in less than 100 accepted networks, indicating more than 99% reduction in the potential network space. The set of accepted networks contains the original network.

What of the cascade model shown in Figure 4.3? In this case, let us assume that we know a single reaction between two floating species but nothing else. After running 10,000 iterations, less than 20 networks including the original network are accepted, indicating indeed more than 99% reduction in the potential network space as before. Figure 4.5 shows some of the other networks that have survived the selection process. In all of the cases, only the reaction between species S_1 and S_2, complete with the repression from species S_4, has been given.

After testing on various types of networks, we suggest that the technique on average

can reduce the search space by more than 95% when an adequate amount of information about the network is given. This technique is appealing for several reasons: The technique makes qualitative comparisons of features. Therefore it is a coarse-grained way of reducing the search space but faster in the sense that parameter optimization is unnecessary in any of the steps. The technique is also continuous. In a highly modularized workflow, this kind of reduction technique can be used as an initial screening step, continuously collecting and passing accepted models down to its downstream processes. The performance is also reasonable. Based on our experience, we expect a million iterations to be carried out in several hours on a typical modern CPU, once the work is parallelized.

The main benefit of such an algorithm is that the resulting search space might become small enough to run a regression analysis on accepted models and to choose the most conceivable model. Furthermore, this algorithm is useful in conjunction with other advanced computational techniques which might benefit from the reduced model search space. One may also analyze the output to infer patterns common across the collected models. For instance, network models illustrated in Figure 4.5 all exhibit cyclic flows of mass between species S_3 and S_4. Some sort of interactions is also expected between two cycles in addition to the negative feedback from species S_4 to the boundary input regulation. This kind of information can be beneficial to understanding the system of interest.

4.2 evoMEG: Evolutionary Algorithm-based Model Ensemble Generation

The first step toward designing a reaction network model is to figure out the correct topology. Once the topology is determined, the rate laws and rate constants can be assessed. However, it is often intriguing to obtain experimental data enough to pinpoint a single topology. In such a case, ensemble modeling is a better approach to assessing a number of potential topologies. Using combinatorics of qualitative perturbation data in general makes the barrier on experiments relatively low. While the information provided by the qualitative comparison of steady-state values would allow a significant reduction in the model search space, the selection criteria might be too simple and coarse-grained to go a step further to generate a

useful model ensemble. Especially for larger models, different, more informative data types might be necessary. Moreover, the brute-force approach taken by the network search space reduction algorithm is far from optimized. While the algorithm is very fast, it might not be highly scalable for applications like whole-cell modeling. These are the specific reasons why the algorithm is branded not as a modeling algorithm but as a search space reduction algorithm even though it technically collects mechanistic models. To meet these requirements, one needs a more advanced algorithm that utilizes other types of data and directive as well.

Here we propose a modified version of an evolutionary algorithm designed to generate an ensemble of detailed mechanistic models from quantitative but scaled perturbation data. In particular, we demonstrate a workflow using control coefficients [112]. A control coefficient, used widely in metabolic control analysis (MCA), describes how the steady-state of a system variable (e.g. a species concentration S or a flux J) is affected by perturbations given to a parameter v_i (e.g. an enzyme concentration E_i or steady-state reaction rate). The concentration control coefficient $C^S_{v_i}$ and the flux control coefficient $C^J_{v_i}$, with respect to a system variable v_i, are defined to be

$$\begin{aligned} C^S_{v_i} &= \frac{v_i}{S}\frac{dS}{dv_i} = \frac{d\ln S}{d\ln v_i} \\ C^J_{v_i} &= \frac{v_i}{J}\frac{dJ}{dv_i} = \frac{d\ln J}{d\ln v_i}. \end{aligned} \tag{4.6}$$

Over the entire system, the unscaled concentration and flux control coefficients are written in the matrix form [56]:

$$\begin{aligned} \boldsymbol{C^S} &= -\boldsymbol{L}\boldsymbol{M}^{-1}\boldsymbol{N_r} \\ \boldsymbol{C^J} &= \boldsymbol{\epsilon_s}\boldsymbol{C^S} + \boldsymbol{I_n}, \end{aligned} \tag{4.7}$$

where $\boldsymbol{M}$ is the Jacobian and $\boldsymbol{L}$ is the link matrix satisfying

$$\boldsymbol{N} = \boldsymbol{L}\boldsymbol{N_r} \tag{4.8}$$

with the full stoichiometry matrix $\boldsymbol{N}$ and the reduced stoichiometery matrix $\boldsymbol{N_r}$, On the second line, $\boldsymbol{\epsilon_s}$ is the elasticity coefficient, a measure of local sensitivity of the reaction rate

v against a given effector, which in this case is the concentration:

$$\boldsymbol{\epsilon_s} = \frac{dv}{dS}. \tag{4.9}$$

Finally, $\boldsymbol{I_n}$ is an identity matrix. Throughout this manuscript, we use scaled concentration control coefficients, the dimensionless and normalized form of concentration control coefficients, as the input data. That is, what is given in Equation 4.6 or Equation 4.7 divided by respective state variables over parameters.

An evolutionary algorithm (EA) is a population-based heuristic optimization algorithm, where the population goes through reproduction, mutation, recombination, and selection processes from generation to generation. Our implementation of the evolutionary algorithm mutates the network topology while running parameter fitting and exploring orthogonal search spaces at the same time. As a population-based algorithm, it naturally results in a population of models. Through selection processes, the algorithm directs the population towards better fitness.

Here we present the Evolutionary Algorithm-based Model Ensemble Generation (evoMEG) algorithm, which utilizes a modified evolutionary algorithm to generate an ensemble of robust and reliable mechanistic reaction network models [30]. We then demonstrate the effectiveness of the algorithm by applying the algorithm to several test cases. Finally, I discuss the usefulness of a model ensemble by demonstrating the effectiveness of aggregated predictions and analyzing the ensemble to direct future experiments. All computation and analysis performed in this section have been done on a combination of Intel i7 4770 processor running at 3.4 GHz with 8GB RAM and AMD Ryzen 1700X running at 3.8 GHz with 16GB RAM.

4.2.1 Implementation

The EvoMEG algorithm is iterative in nature. It starts from generating a population of random biochemical reaction networks with a given number of floating species and reactions. In this and subsequent random network generation routines, models only with unique topologies are generated since the algorithm tracks every topology it encountered and looks

for only novel ones. Within the scope of this research, the algorithm is limited to generating networks with UNI-UNI, UNI-BI, BI-UNI, and BI-BI type reactions. Namely, the reactions are defined as one of the following:

$$\text{UNI-UNI} : A \rightarrow B$$

$$\text{UNI-BI} : A \rightarrow B + C$$

$$\text{BI-UNI} : A + B \rightarrow C$$

$$\text{BI-BI} : A + B \rightarrow C + D$$

All reactions are defined via Michaelis-Menten kinetics. For each reactions type, rate laws are generated according to

$$v = \frac{dP}{dt} = \frac{V_{max} \prod S}{K_M + \prod S}, \tag{4.10}$$

where P and S denote the concentrations of products and of substrates, respectively.

A detailed description of the rate law is unnecessary as long as it can accurately predict the steady-state behavior. In this study, the algorithm will only use the scaled concentration control coefficients, which describe the steady-state responses to perturbations. How detailed the rate law can describe the (transient) dynamics is unimportant. Therefore the rate laws can be considerably simplified in the scope of this study and we use Michaelis-Menten kinetics for completeness although lin-log [110] or even mass action kinetics should be reasonable approximations. Once the topology is approximated, detailed rate laws and values of corresponding rate constants can be figured out.

Once the initial seed of the population is generated, each model in the population goes through a round of global optimization to fit the rate constants. (Since the random network generation only looks for unique topologies, the rate constants are not optimized until the parameter estimation is done.) In this study, we adopt intentionally relatively high tolerance values for the global optimization routine to speed up the compute time. This is proper since we are interested in models with relatively good fitness regardless of the absolute value of the objective function. For the optimization of rate constants, differential evolution (DE) [123] is used.

In fact, the evoMEG algorithm looks for solutions in two orthogonal search spaces, one for the topology and the other for the rate constants. When looking for both solutions, we use the same objective function given below:

$$F_{obj} = \mathrm{N}_F(C^{true} - C^{test}) \left[1 + F_{neq}^{sign}(C^{true}, C^{test})\right] \tag{4.11}$$

with the Frobenius norm of the difference in matrices

$$\mathrm{N}_F(C^{true} - C^{test}) \equiv \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} |C_{ij}^{true} - C_{ij}^{test}|^2}, \tag{4.12}$$

which is conceptually similar to calculating least-squares between two one-dimensional arrays. F_{neq}^{sign} represents the number of values in the calculated concentration control coefficients that differ in sign compared with the reference data. The signs of concentration control coefficients provide a large amount of information [37, 116], and Equation 4.11 heavily penalizes those concentration control coefficients with mismatching signs. We consider the model to have better fitness if the output of Equation 4.11 is smaller.

To obtain the output of the objective function, we need to calculate scaled concentration control coefficients. This means that the model in question must carry a steady state. This restriction is enforced during the random network generation, mutation, and any time when the objective function is computed. For the system of interest, the availability of a steady state is manifested through the ability to measure concentration control coefficients.

Once the global optimization routine converges, the optimized rate constants are used for calculating the scaled concentration control coefficients and subsequently, the objective function. The output of this objective function is treated as the best fitness for the model topology given, by which models in the population are ranked.

The selection process chooses which models to mutate. The process splits randomly the model population to create two sets of models. These two sets of models are compared and models with better fitness are chosen to be mutated. The top ten percent of the models are always selected to be mutated. Models that have not been chosen to be mutated are discarded.

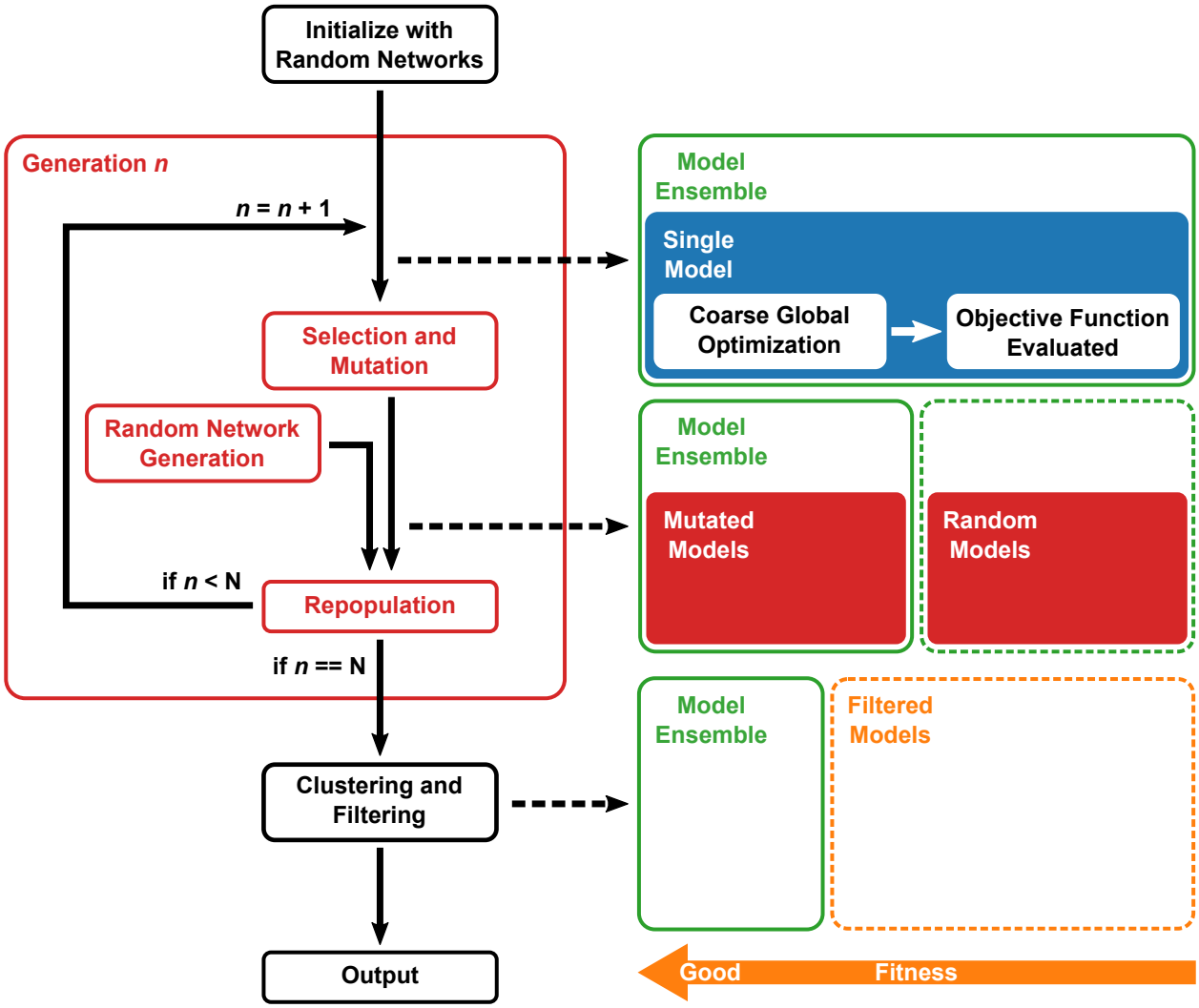


Figure 4.6: Workflow of the evoMEG algorithm. Diagrams on the right illustrate what is happening to the model population at each step. The smaller the output of the objective function, the better the fitness of the model.

The mutation process attempts to mutate the model in the hope of achieving better fitness and involves three types of modifications. First, a reaction is randomly chosen. The probability of choosing a reaction is calculated on the basis of the output of the objective function per reaction. Namely, the algorithm has a higher chance of picking a reaction

that contributes the most to low fitness. Second, a reaction type to replace the reaction is randomly chosen. The probability of choosing one reaction type relative to another is set to be a constant. The UNI-UNI reaction type has the highest probability to be chosen while the BI-BI reaction type has the lowest probability. Third, species to participate as reactants and products are randomly chosen. While doing so, the mutated model is checked for passing the assumption as to the number of correct floating species and the existence of a steady state. Furthermore, the modification is accepted only when the resulting fitness is better than that of the parent. Therefore, our adaptation of the evolutionary algorithm differs from the traditional version of the evolutionary algorithm: Our algorithm is much more aggressive and does not involve recombination. This is due to the fact that defining the concept of recombination for topology is difficult, especially in consideration of the restrictions imposed on the generated models (e.g., a certain number of species must be floating species, there are no identical reactions within the same model, the model must have a steady state, and so on).

When the population is repopulated with mutation and random network generation, the population is passed to the next generation, going through the process of global optimization for rate constants. The entire process is repeated for a set number of generations or until the fitness reaches a certain threshold. Once the end condition is met, the population is filtered systematically to collect only models with good fitness. In this way, models apparently at the global minimum in both topological and parametric search spaces are collected, leaving with a model ensemble composed of evolved models with topologies that fit well to the data given. This is based on the assumption that the initial population size is much larger than the number of topologies that can produce a good fit.

Filtering stage can be done in a few different ways. A simple way is to implement an ideal low-pass filter where models with fitness under the cutoff threshold are collected. The threshold can be relative since the absolute value of fitness is dependent on the type and the scale of the system of interest. For example, the fitness can be analyzed and a threshold can be picked, which results in five percent of the population collected as the ensemble. However,

depending on the system, the low-pass filter might not be the most accurate choice. The low-pass filter performs particularly poor when a subset of the population have fitness values clustered together (see, e.g., Figure D.1A). Ideally, if this cluster is in the regime of the best fitness, all models in the cluster should be collected. However, the low-pass filter provides no solution for this.

Instead, kernel density estimation (KDE) [105, 101] and relative minima calculation are used for the study. Kernel density estimation is intended to be used for estimating a probability density function. However, for univariate multimodal distributions where there are clusters, kernel density estimation smooths the probability density of fitnesses from which relative minima can be calculated. The relative minima corresponds to the turning points. Systematic filtering using kernel density estimation and relative minima can collect a subset of the population by selecting the models below the lowest turning point. The only problem with kernel density estimation is how to select the bandwidth, which is a somewhat arbitrary parameter impacting the outcome. One way to address this issue is to calculate the mean integrated square error along with a range of bandwidth values and determine an approximate 'good' bandwidth value. There are also data-driven bandwidth selection methods reported in literature [118, 63]. Here one should keep in mind that no methodology is error-proof and some amount of manual analysis may thus be required.

Once the filtering stage is done, the algorithm yields a model ensemble along with detailed statistics of the run. Figure 4.6 illustrates the workflow of the evoMEG algorithm. The model ensemble may be analyzed for making either reasonable predictions on the system or decisions on future experiments to reduce the size of the ensemble.

4.2.2 Results

To test the algorithm, we consider four synthetic models as exemplary cases. These include a coherent type-1 feed-forward loop, a linear chain, cycles, and branching pathways, which are shown schematically in Figure 4.7.

Figure 4.8 illustrates some of the models collected by the algorithm in the case of the

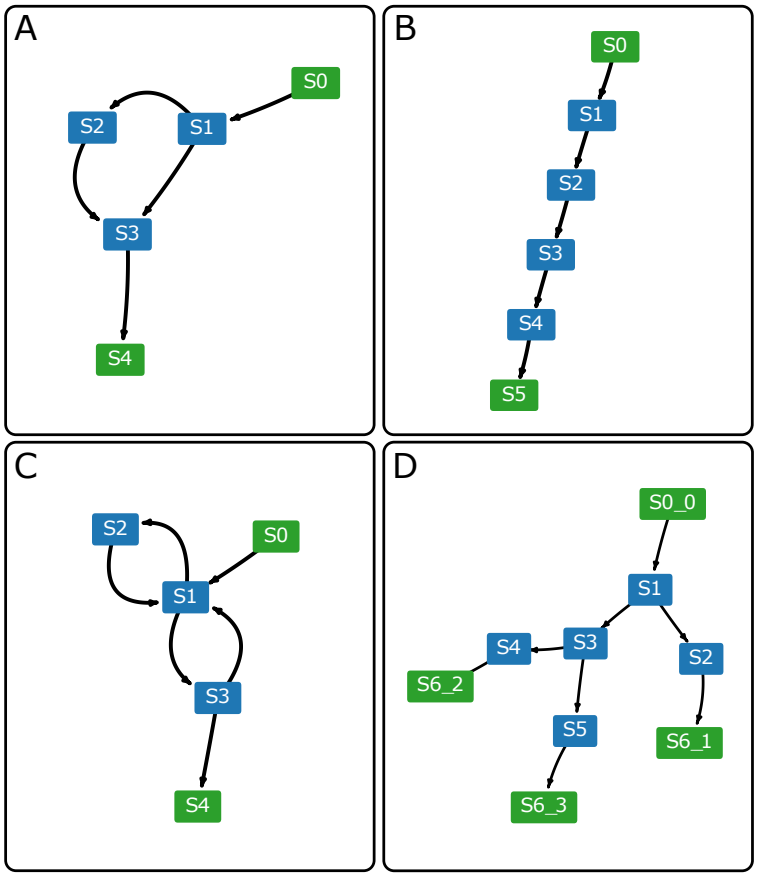


Figure 4.7: Network diagrams of models used as test cases, including (A) a feed-forward loop, (B) a linear chain, (C) cycles, and (D) branching pathways. Nodes in green represent boundary species which are fixed.

feed-forward loop. It turns out that the algorithm collected five models via kernel density estimation for systematic filtering. The algorithm recovered the original model and also collected other models with topologies that generated comparable scaled concentration control coefficients.

For a linear chain, the algorithm generates an ensemble of 24 models including the original model. Interestingly, for the linear chain topology, the scaled concentration control coefficients could not determine different orderings of the species. The algorithm collected

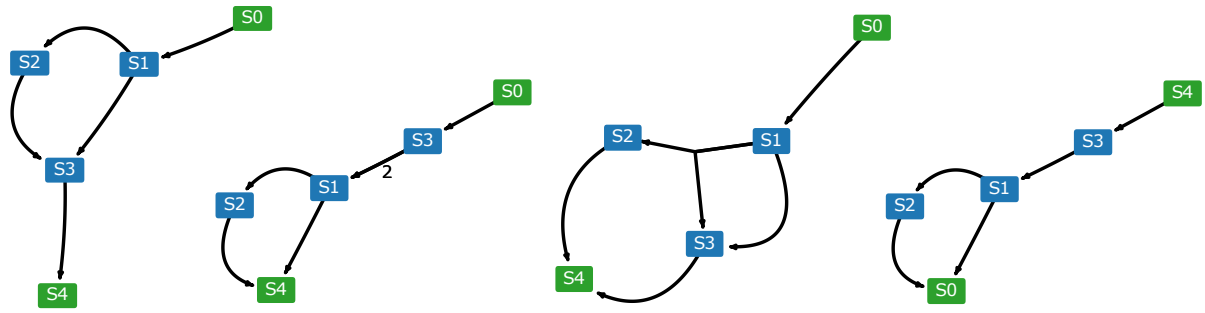


Figure 4.8: Network diagrams of selected models in the ensemble for the feed-forward test case. Nodes in green represent boundary species which are fixed.

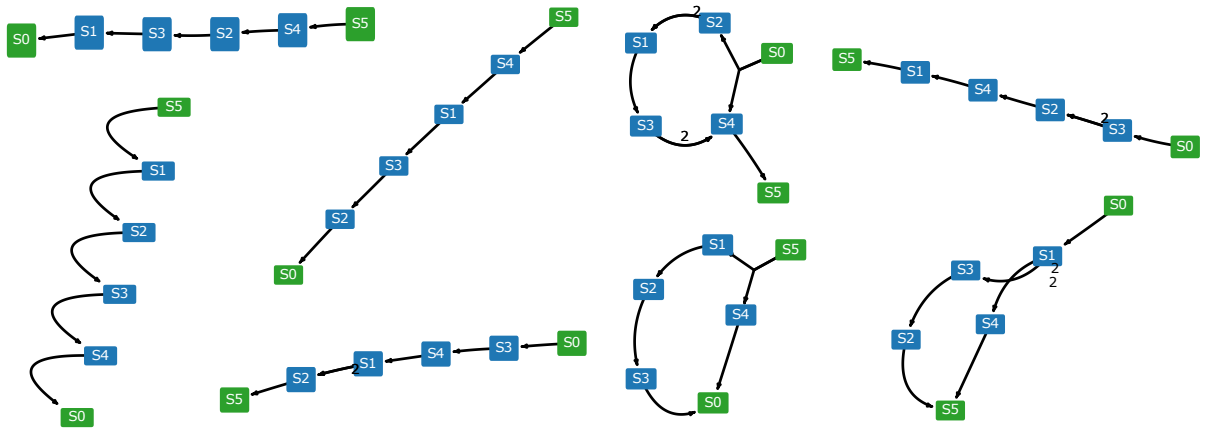


Figure 4.9: Network diagrams of selected models in the ensemble for the linear chain test case. Nodes in green represent boundary species which are fixed.

multiple versions of linear chains with different orderings of the species, as displayed in Figure 4.9. This reflects the unique characteristics of the models composed of irreversible reactions. As long as the reactants of all reactions match with those in the original model, the scaled concentration control coefficients tend to return similar values regardless of the products. Then the scaled concentration control coefficients just suggest linear chain-like topology and the algorithm ends up with collecting models of linear chains with different

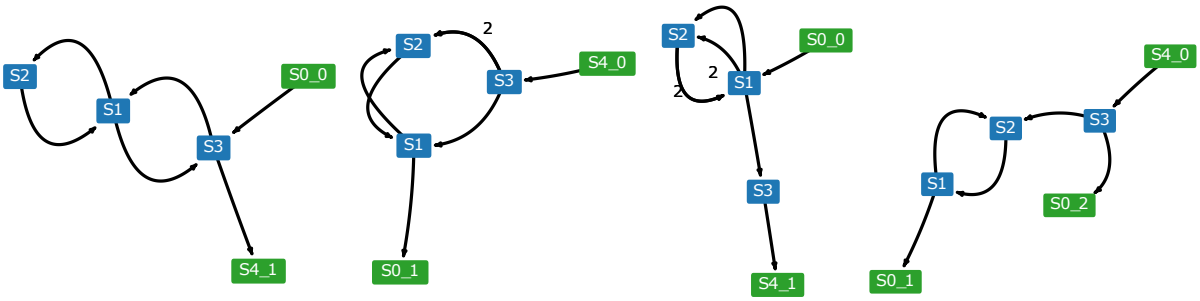


Figure 4.10: Network diagrams of selected models in the ensemble for the cycles test case. Nodes in green represent boundary species which are fixed.

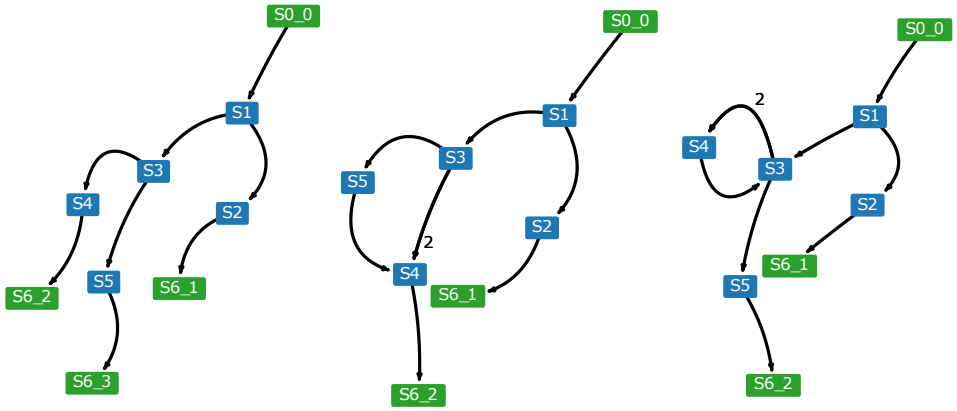


Figure 4.11: Network diagram of selected models in the ensemble for the branched pathway test case. Nodes in green represent boundary species which are fixed.

orderings of the species.

In the case of the cycles, the algorithm collected ten models, some of which are illustrated in Figure 4.10. In all cases, the whole or part of the cycles is observed to be recovered. The algorithm had a hard time recovering the original topology, like the case of a linear chain. The reason is different, though: For cycles, it is rather difficult to specify the boundary inputs and outputs because essentially any part of a cycle can be the input or the output.

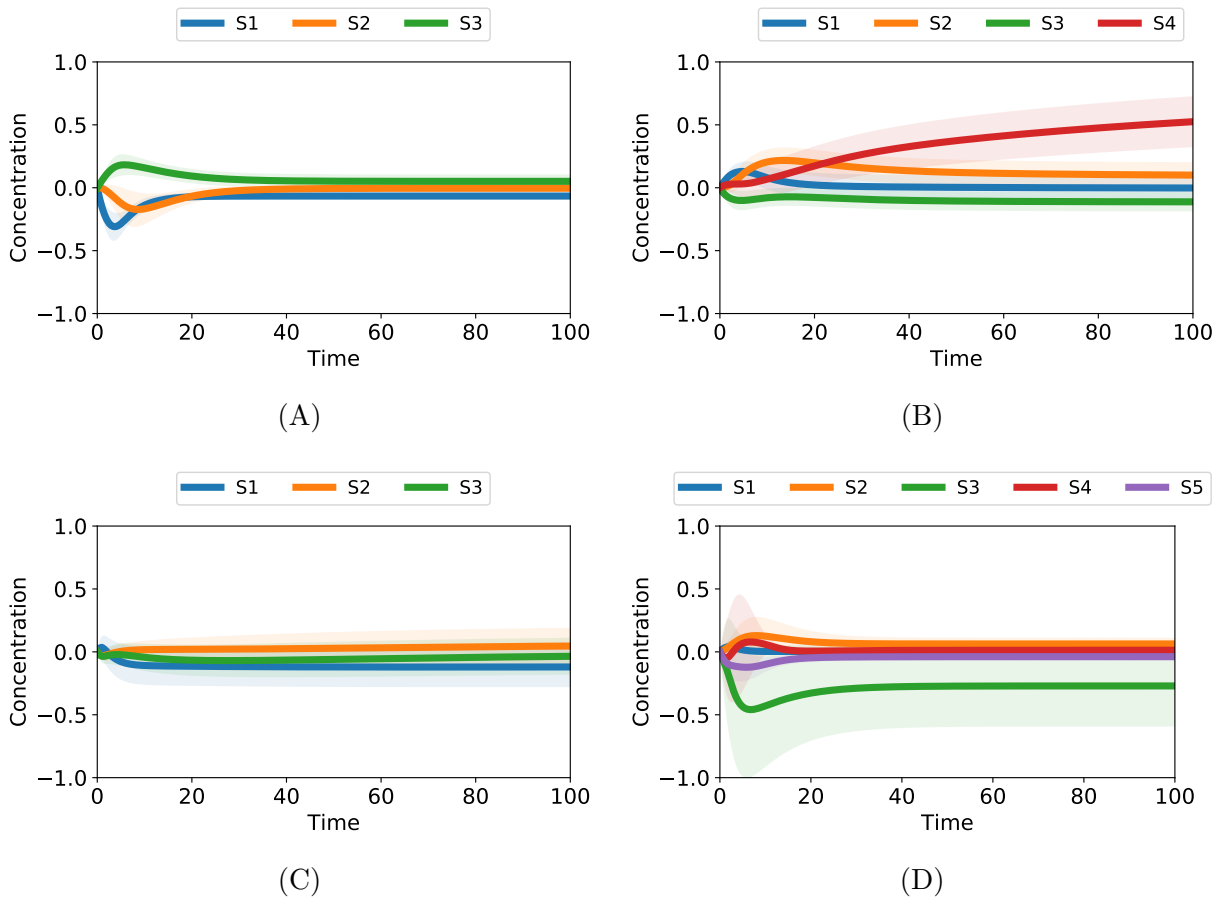


Figure 4.12: Residuals of time-course simulation data between the aggregate predictions made via the ensemble and the corresponding test case: (A) feed-forward loop, (B) linear chain, (C) cycles, and (D) branched pathways. Shaded regions correspond to the standard errors of species concentrations over all models in the model ensemble.

Finally, the algorithm applied to the branched pathway recovered three models shown in Figure 4.11. The original model, which was recovered as well, turns out to have the best fitness.

The model ensembles can be used in two different ways: First, aggregated predictions can be made through the use of the ensemble as a whole. It has been demonstrated [51, 38,

124] that aggregated predictions made via the model ensemble often fares better than the individual predictions made via a model in an ensemble. For biochemical reaction networks, this is true even if the ensemble contains the original topology because many motifs often have issues with parameter identifiability, resulting in similar simulation results while parameters vary.

To demonstrate this point, we calculate residuals in time course traces and display the results in Figure 4.12. The residuals are calculated by comparing the time course traces of the test cases illustrated in Figure 4.7 and the ensemble averaging (bagging) of time course traces of the model ensemble. Shaded regions correspond to the standard errors. Even though most of these ensemble contains topologies and parameters different from those in the original model, the ensemble can make reasonable predictions on the steady-state values and the transients. The linear chain test case performed worse than the others presumably because of the same reason that led the algorithm to collect models with different orderings of the species.

Also compared are the fluxes from ensemble aggregation and the corresponding test cases (see Figure 4.13). Error bars correspond to the standard errors. Values from aggregate predictions are within narrow ranges of the true fluxes from the original models. Interestingly, the fluxes calculated from ensemble aggregation from the inputs (first reactions in the plot) and the outputs (last reactions in the plot) are generally the most deviating from the original models. This is especially true for the topologies that the algorithm had a hard time determining the correct input and output (such as linear chains and cycles). Overall, the Pearson correlation coefficient calculated over all flux values is given by $r = 0.895$ with the p-value as low as $p = 3.443 \times 10^{-9}$. This strongly confirms the presence of correlations between the true flux values and the aggregated predictions.

Second, the ensemble can be analyzed to direct future experiments. The main goal of the evoMEG algorithm is to generate a model ensemble of which individual models can explain the given data equally well. Thus, the ensemble can contain models with various topologies. Even though ensemble aggregation is demonstrated to be useful for making predictions as is,

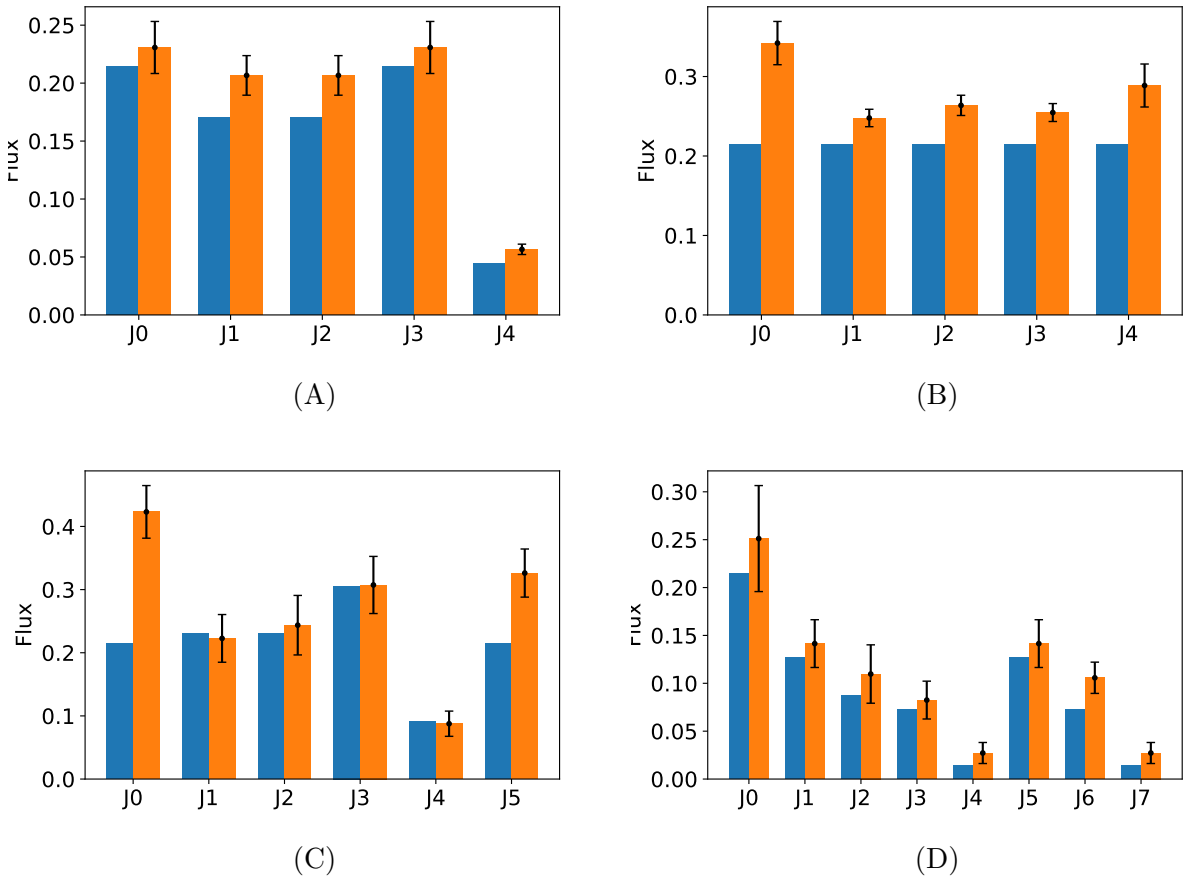


Figure 4.13: Comparison of fluxes between the aggregate predictions via the ensemble and the corresponding test case. Orange bars and blue bars correspond to fluxes calculated from the ensemble aggregation and the test case, respectively: (A) feed-forward loop, (B) linear chain, (C) cycles, and (D) branched pathways. Error bars correspond to the standard errors of fluxes over all models in the model ensemble.

it is often desired to generate models with less ambiguity for the mechanistic understanding of the system. The individual models in the ensemble can be analyzed to identify key information necessary for maximally reducing the ensemble. Consider the model ensemble for the feed-forward loop illustrated in Figure 4.8. The ensemble largely consists of models

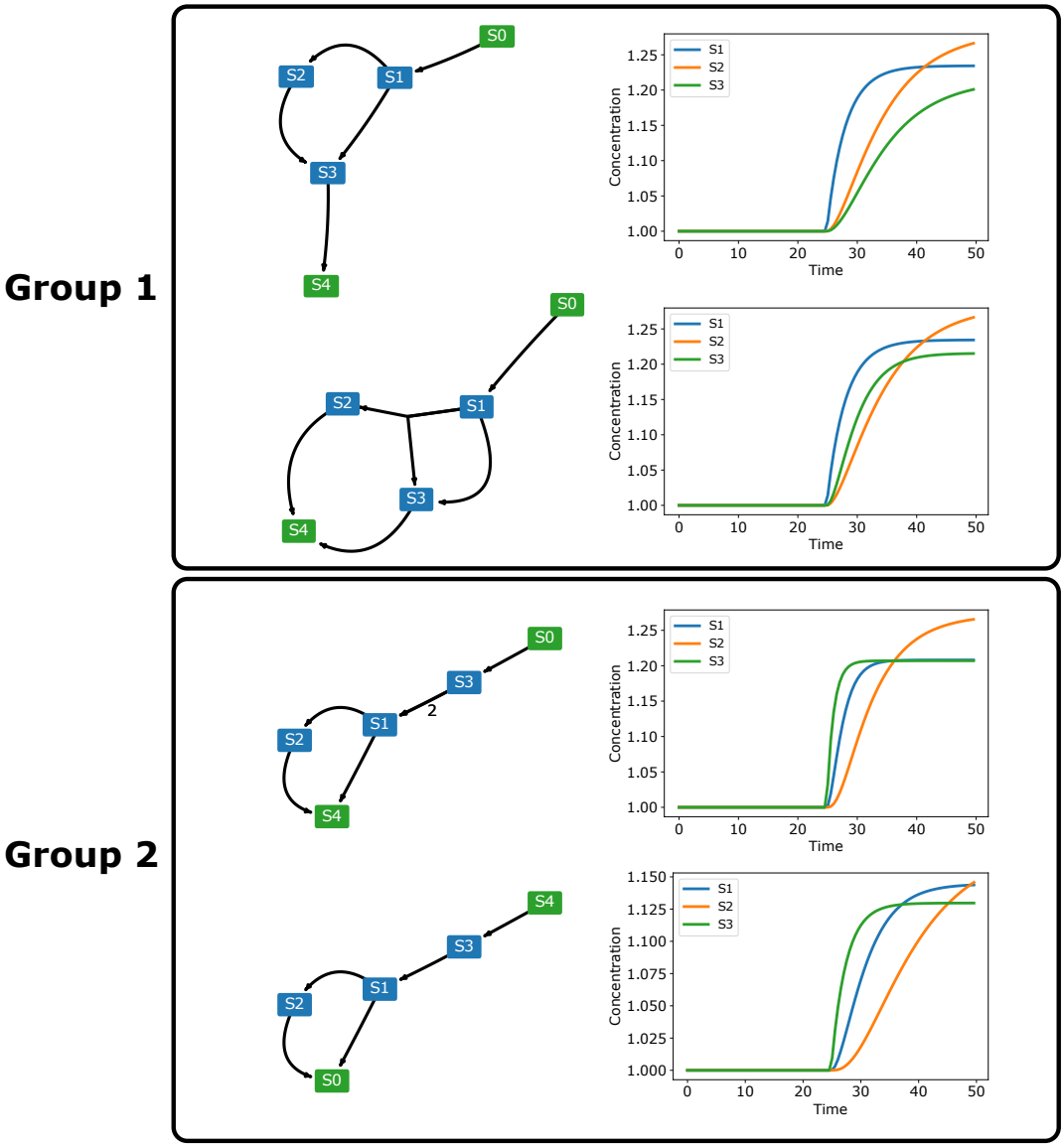


Figure 4.14: Model ensemble generated from the feed-forward loop test case has two groups of models with distinct topologies. When perturbations are applied to the boundary input, two groups show different transient responses, most notably in species $S1$ and $S3$. The concentrations are scaled to the steady-state values. Nodes in green represent boundary species which are fixed.

with two distinct topologies as shown in Figure 4.14. Because of the similarity in steady-state values and fluxes between these two groups, transient responses to perturbations on the boundary input are the information that can exclude one of the groups completely. Figure 4.14 illustrates transient responses in the steady state when the boundary input concentration is doubled. The concentrations are scaled to the steady-state values. Two groups show different transient responses, most notably in species $S1$ and $S3$. In group 1, the concentration of species $S1$ increases faster than that of species $S3$. In group 2, the opposite is true. Because the model ensemble can be simulated, it can be used to identify the set of information that can further reduce the size of the ensemble. Once the transient responses to perturbations are experimentally obtained, one can safely knock out around half of all models in the ensemble, significantly reducing the variance in the topology.

One thing to note is that our study has demonstrated the issue of parameter identifiability. In most of our test cases, the model ensemble contained the original topology. However, the model with the correct topology still made predictions with deviating the steady-state values and the transient responses. Even if the tolerance for parameter estimation was increased, the behavior persisted. Parameter uncertainty is often attributed to the noise, but in our case, it is due to an insufficient amount of data since our data are noise-free. The algorithm is provided with only the scaled concentration control coefficients, which seems not to be enough to identify the correct set of rate constants.

If available, additional experimental data can be supplied to obtain a better set of rate constants. Data such as the steady-state values and fluxes are relatively easy to measure compared with time-course data. In particular, we found that supplying the steady-state values significantly improves the fit of transient responses. For example, if the model ensemble goes through another round of parameter optimization using the objective function shown in Equation 4.13, the deviation in transients observed in the branched pathway becomes much smaller. Figure 4.15 illustrates this point by comparing the residuals of time-course simulations between the original test case and the model collected by the algorithm and the same residuals but after running parameter optimization on the model collected by the

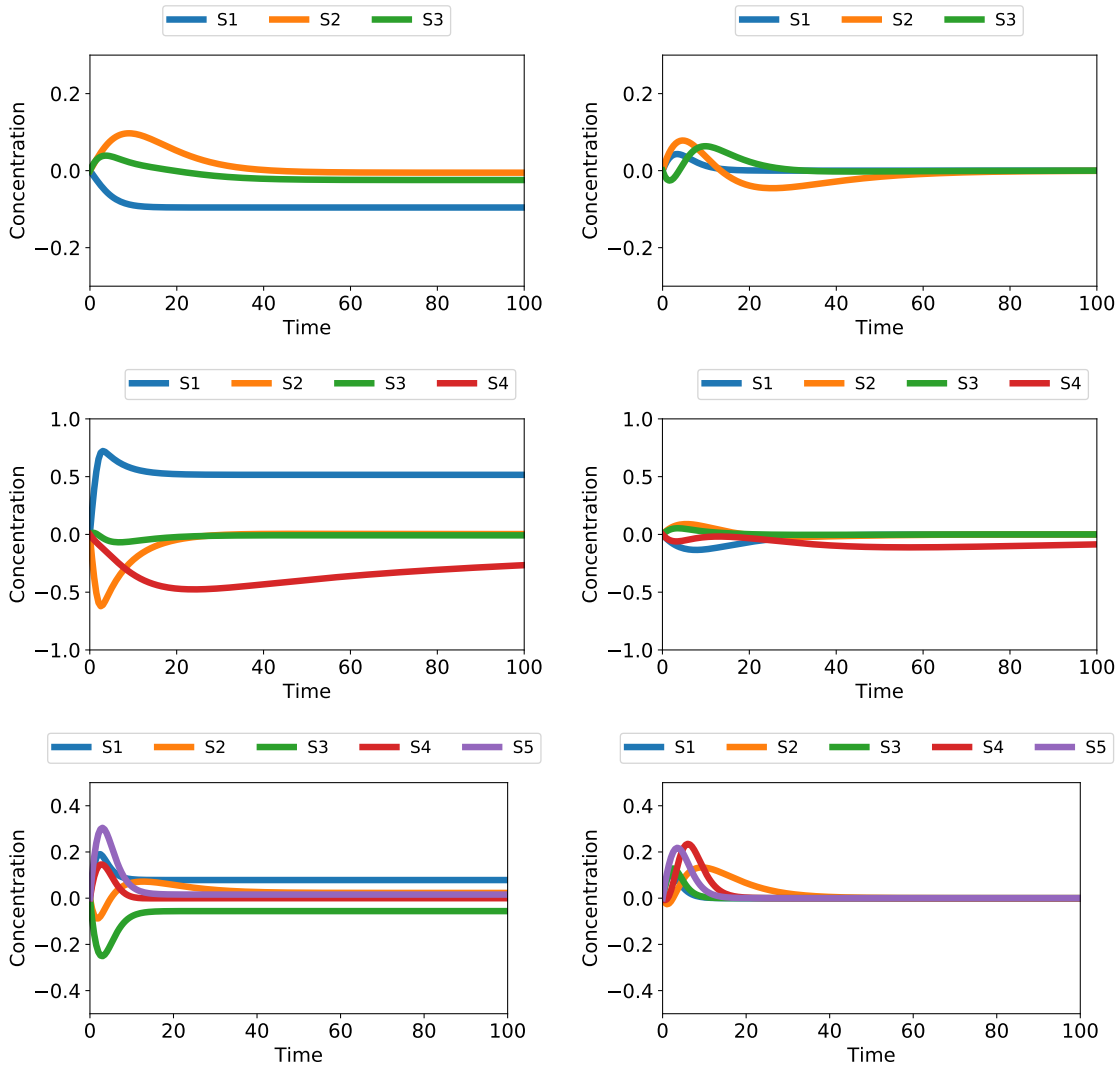


Figure 4.15: Running extra rounds of parameter optimization using steady-state values reduces deviations in time-course simulations. Models with the original topology collected by evoMEG have been used except the case of the cycles. The left column shows the residuals of time-course simulations between the original test case and the model collected by evoMEG. The right column shows the same residuals but after running parameter optimization on the model collected by evoMEG using the updated objective function in Equation 4.13. From top to bottom: feed-forward loop, linear chain, and branched pathway test cases.

algorithm using the updated objective function. The case of cycles is not shown because the algorithm was unable to recover the correct topology with the correct boundary input and output, although it correctly predicted the cycles.

$$F_{obj} = \mathrm{N}_F(S_{true} - S_{test}) \tag{4.13}$$

Because the analysis is done only on models in the ensemble, higher tolerance can be set if desired, compared with the tolerances set for searching topologies. If various types of data are available, going through multiple rounds of optimizations is a valid strategy to improve and potentially to discard models in the ensemble, funneling the best performing models in the end.

The examples in this chapter demonstrate how evoMEG can generate a model ensemble from perturbation data, how the model ensemble can be used to make reasonable predictions, and how ensemble modeling can direct the future experiment, leading to more efficient experiments. The ensemble modeling approach is a good example of closing the loop between the experiments and the modeling endeavors. The histograms of population fitness are presented in Appendix D and the convergence curves in Appendix E.

4.2.3 Case Study for Noisy Data

One concern of using the evoMEG algorithm involves the effects of noisy data on the model ensemble. In the previous section, the ensemble was obtained from synthetic data which were noise-free. In reality, data should have some amount of noise and understanding the noise effects on the algorithm is necessary for applications. For this purpose, the feed-forward loop case was used again at low and high levels of Gaussian noise added to the synthetic data (which are scaled concentration control coefficients). To simulate both the intrinsic noise of the species concentrations and the experimental (measurement) noise, we have considered noise η given by the addition of two Gaussian noises $\mathcal{N}(\mu, \sigma)$ with mean μ set equal to zero and standard deviation σ, one scaled to the values of scaled concentration control coefficients

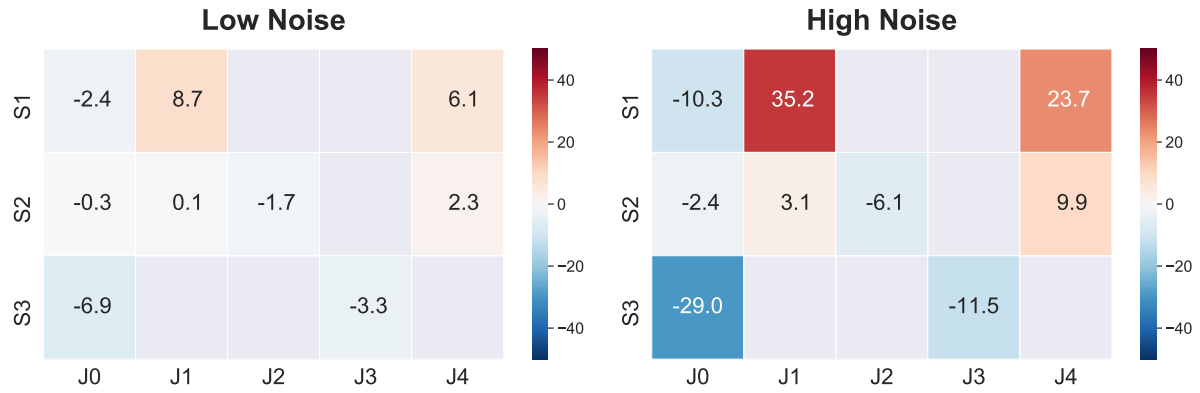


Figure 4.16: Heatmaps of percentage differences in scaled concentration control coefficients between low and high levels of noise and the original values. Some of values are not shown because the original values are zero.

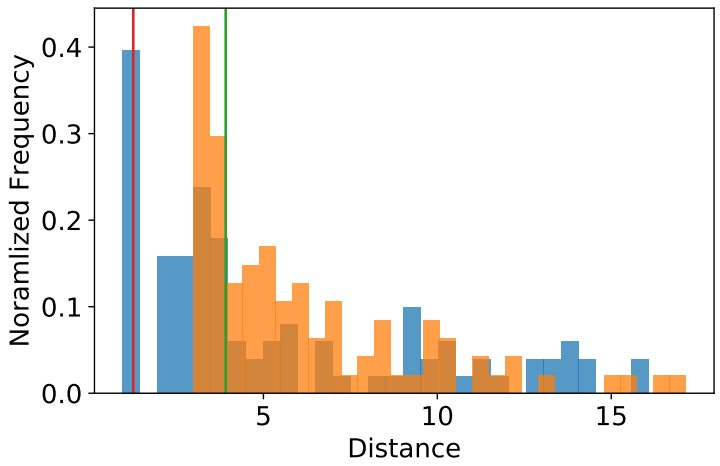


Figure 4.17: Histograms of population fitness under low-noise (blue) and high-noise (orange) conditions. Red and green lines represent values between which kernel density estimation is used to filter the population to generate a model ensemble under low-noise and high-noise conditions, respectively.

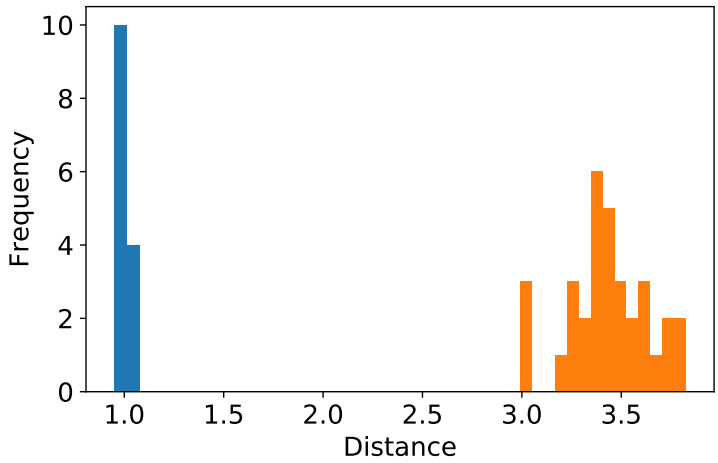


Figure 4.18: Histograms of model ensemble fitness after filtering via kernel density estimation. Blue and orange bars correspond to low and high noise conditions, respectively. The bin size has been scaled to the range of distances and the number of models in the ensemble. The population size for the high-noise condition is larger and the spread of the distribution is much wider.

(intrinsic noise) and the other set to be a constant (experimental noise):

$$\eta_{ij} = \mathcal{N}(0, C_{ij}^{true}\sigma_{rel}) + \mathcal{N}(0, \sigma_{abs}). \tag{4.14}$$

Two different noise conditions are tested: We generate the low-noise condition through the use of $\sigma_{rel} = 0.05$ and $\sigma_{abs} = 0.005$. The high-noise condition is generated from $\sigma_{rel} = 0.2$ and $\sigma_{abs} = 0.005$. Figure 4.16 shows the heatmaps of percentage differences in matrices of scaled concentration control coefficients between low and high levels of noise and the original values. Some of the values are not shown because the original values are zero. The raw values for the differences are available in Appendix F.

Once the algorithm is run, the resulting ensemble have distances illustrated in Figure 4.17. Not surprisingly, the distances measured tend to be larger under the high-noise condition. Also, the minimum distance of the population under the high-noise condition is about three

times larger than that under the low-noise condition. Next, the ensemble is filtered to collect models with good fitness. For systematic filtering of the model ensemble, kernel density estimation with a bandwidth of 0.1 has been used. Appendix G presents samples of models collected after filtering under the low-noise and the high-noise conditions, respectively. Kernel density estimation is observed to result in larger ensemble size at higher noise levels: The ensemble size was given by $N = 13$ and 32 under the low-noise and the high-noise conditions, respectively. This indicates that the algorithm is able to fit more model topologies to given noisy data. The distribution of fitness values in the model ensemble also displays larger means and deviations at higher noise levels. Figure 4.18 demonstrates that the mean and the deviation of the distances are larger under the high-noise condition compared with those under the low-noise condition.

Even in the high-noise condition, the original topology was recovered and remain as part of the filtered ensemble. However, in case that the data become extremely noisy (noisier than the high-noise condition), the algorithm could neither recover the original topology nor generate a reliable model ensemble. The accuracy of an algorithm, after all, depends largely on the quality of the data given.

4.2.4 Reversible Reactions

So far, the algorithm generated models with irreversible reactions only. While many biological processes can be approximated as irreversible reactions, in reality, most biochemical processes are reversible. Reversible reactions are necessary for describing a wider variety of systems. To understand how the evoMEG algorithm responds to models with reversible reactions, we have re-implemented the random network generation algorithm and constructed models with reversible reactions. The rate laws are given by

$$v = \frac{k_f \prod S - k_r \prod P}{K_M + \prod S + \prod P}. \tag{4.15}$$

The algorithm has been tested again on the four exemplary cases described before, except that this time, all reactions have been taken to be reversible (see Figure 4.19). It is seemingly

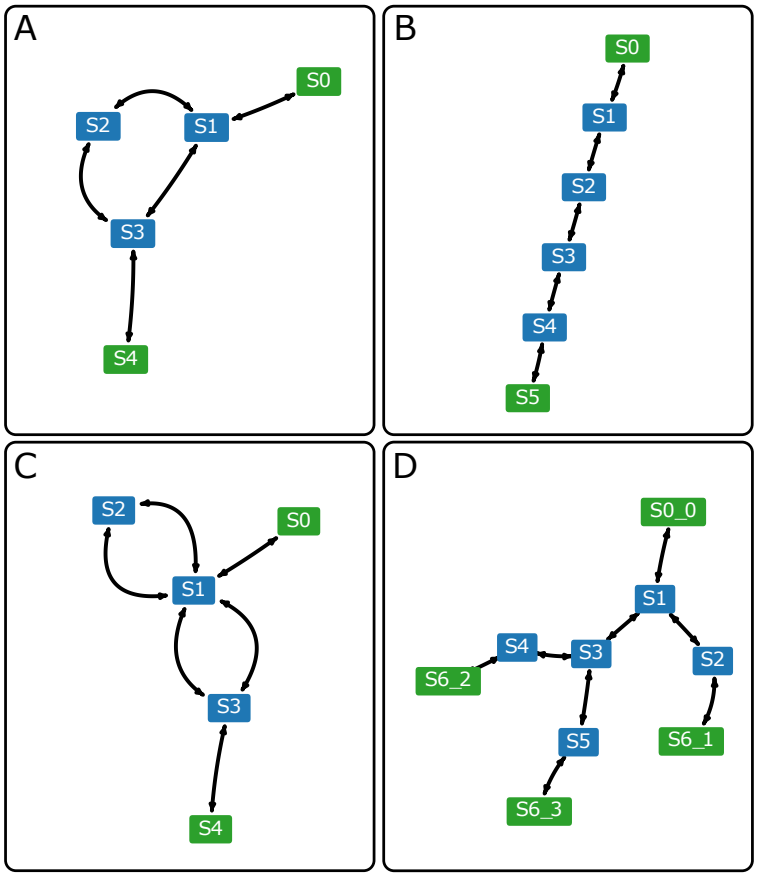


Figure 4.19: Network diagrams of models used as test cases with reversible reactions: (A) feed-forward loop, (B) linear chain, (C) cycles, and (D) branching pathways. Nodes in green represent boundary species which are fixed.

more difficult to solve this problem because the matrix of scaled concentration control coefficients becomes much more complex. When all reactions are reversible, species downstream can affect species upstream. Consider the feed-forward loop case shown in Figures 4.7A and 4.19A. In this case the scaled concentration control coefficients with irreversible reactions

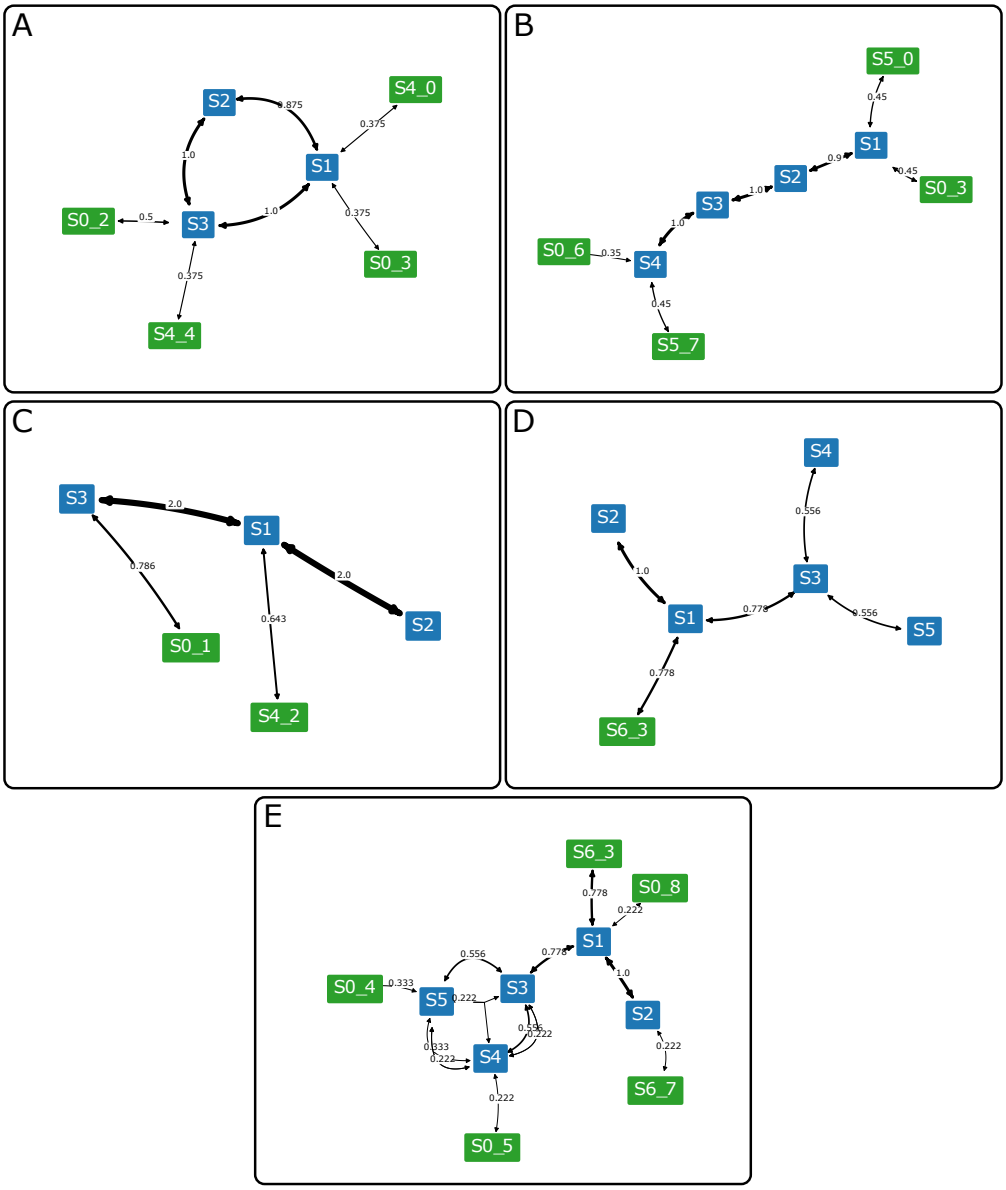


Figure 4.20: Weighted network diagrams of the model ensemble from test cases with reversible reactions: (A) feed-forward loop, (B) linear chain, (C) cycles, (D) and (E) branching pathways. (A) to (D) are generated with threshold of 0.34. (E) is generated with threshold of 0.2. (C) has edges with weight over one because netplotlib treats two reversible reactions in a cycle as an identical reaction. Nodes in green represent boundary species which are fixed.

and with reversible reactions are described, respectively, by

$$C^{true}_{irreversible} = \begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \end{array} \begin{array}{c} \begin{array}{ccccc} J_0 & J_1 & J_2 & J_3 & J_4 \end{array} \\ \begin{pmatrix} 1.42356 & -1.1301 & 0 & 0 & -0.293459 \\ 1.76313 & 0.363461 & -1.76313 & 0 & -0.363461 \\ 1.3941 & 0 & 0 & -1.3941 & 0 \end{pmatrix} \end{array} \tag{4.16}$$

and

$$C^{true}_{reversible} = \begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \end{array} \begin{array}{c} \begin{array}{ccccc} J_0 & J_1 & J_2 & J_3 & J_4 \end{array} \\ \begin{pmatrix} 0.733131 & -0.0258709 & -0.0267344 & -0.310234 & -0.370291 \\ 0.551933 & 0.062784 & -0.0834463 & -0.385827 & -0.145443 \\ 0.349085 & 0.00710745 & 0.00734466 & -0.465266 & 0.101729 \end{pmatrix} \end{array} \tag{4.17}$$

Surprisingly, once the algorithm was run, the model ensemble for test cases with reversible reactions performed better than those with irreversible reactions. To be specific, the algorithm was able to recover the original topology as well as the correct boundary inputs and outputs, practically collecting only the original model with minute topological differences such as additional boundary inputs or outputs. This is unlike models with irreversible reactions, where the orderings of species (e.g. the case of a linear chain) or the boundary inputs and outputs (e.g. the case of cycles) are hard to identify. As we understand, the reason is that reversible reactions provide information on both the reactants and the products of a reaction, unlike irreversible reactions which provide information only on the reactants. The only exception is the branching pathway case, where the downstream reactions are hard to determine. However, once the models in the ensemble are combined, the patterns turn out pretty obvious. Figure 4.20 illustrates resulting model ensembles using weighted network diagrams. The test cases of the feed-forward loop, linear chain, and cycles are spot-on with a cutoff threshold of 0.34; the case of branched pathways seems to miss boundary outputs, but once the threshold of 0.2 is set, the boundary input and outputs are recovered. The only oddity is the direct interaction between species $S4$ and $S5$, which does make sense because both species can technically affect each other through species $S3$.

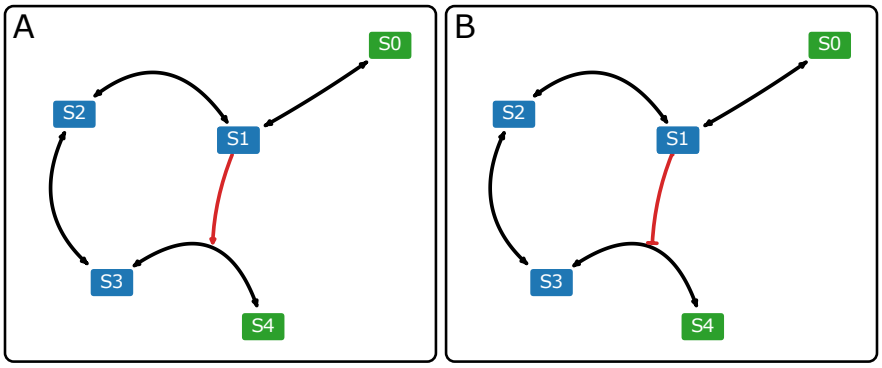


Figure 4.21: Network diagrams of models used as test cases with reversible reactions and regulations: (A) linear chain with activation and (B) linear chain with inhibition. Nodes in green represent boundary species which are fixed.

Unfortunately, unlike the test cases with irreversible reactions, the predictions via the model ensemble generated from reversible reactions generally fared poorly. We suspect the extra degrees of freedom in parameter estimation for rate constants to be responsible for this. Regardless, evoMEG works without any problem when dealing with reversible reactions in terms of generating a model ensemble with correct topologies.

4.2.5 Addition of Regulations

Many biological processes involve regulations of some sort, evident in various metabolic and signaling pathways [86, 68, 20, 25, 3]. It would be of immense benefit if the evoMEG algorithm can figure out potential regulations in addition to irreversible and reversible reactions. To achieve this goal, we should modify the rate law so as to accommodate regulations. However, defining a regulation is not a simple task. There are multiple ways to implement regulations based on allostery, cooperativity, etc.

In the scope of this study, we assume non-competitive inhibition only. The rate law then reads [75, 110]

$$v = \prod \left(\frac{1}{1+\mu_i} \right) \frac{k_f \prod S - k_r \prod P}{1 + \prod S + \prod P}, \tag{4.18}$$

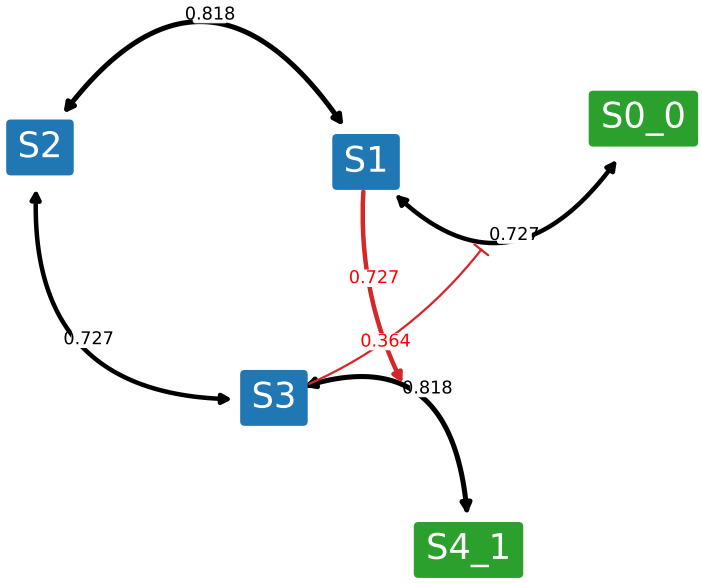


Figure 4.22: Weighted network diagram from the model ensemble generated for the case of a linear chain with activation at the threshold of 0.34. Nodes in green represent boundary species which are fixed.

where μ_i is the inhibitor. For the sake of simplicity, direct binding is assumed, but the generalized form of the Michaelis-Menten rate law (a.k.a common modular) can be substituted without increasing the complexity of the parameter estimation problem.

While this is a gross simplification of real biological processes, it is acceptable since our primary goal is not to find the model with the correct dynamics but to figure out the existence of regulations. If the model ensemble suggests the existence of activation or inhibition, the details on the rate law such as allostery, Hill coefficients, etc. can be analyzed later to get the complete picture. Technically, for the purpose of this study, our rate law simply needs to simulate the effects of a regulator on the control coefficients and may be abstracted accordingly.

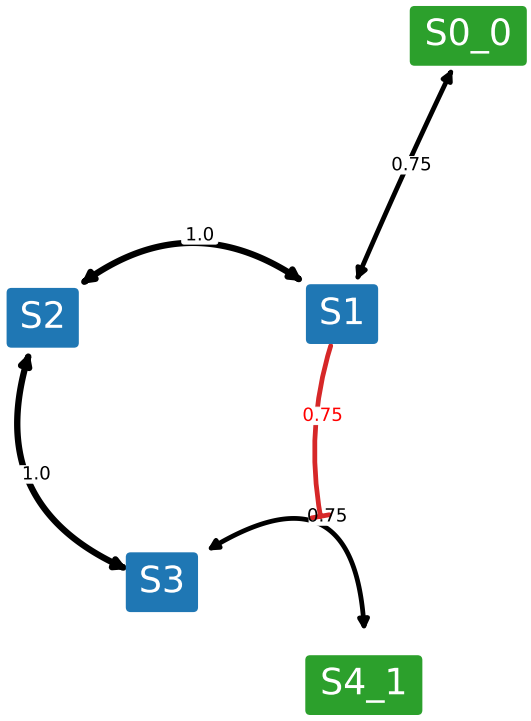


Figure 4.23: Weighted network diagram from the model ensemble generated for the case of a linear chain with inhibition at the threshold of 0.34. Nodes in green represent boundary species which are fixed.

In fact, activation has been implemented in a simplified manner. We have implemented activation in the way that it simulates the behavior of an activator (an increase in the activator level increases the reaction rate and vice versa). In the absence of cooperatvity, the full rate law with both activation and inhibition present is defined as follows:

$$v = \prod (1 + \mu_a) \prod \left(\frac{1}{1 + \mu_i} \right) \frac{k_f \prod S - k_r \prod P}{1 + \prod S + \prod P}, \tag{4.19}$$

where μ_a denotes the activator. The algorithm is updated in such a way that regulations are added according to given probability while generating random networks and mutating the existing reactions. In this study, only a maximum of two regulations (a pair of activation

and inhibition) have been allowed per reaction. Appendix H explores the validity of this assumption further.

To test the updated algorithm, we have used two cases based on linear chains (see Figure 4.21). In both test cases, all reactions are reversible. Figures 4.22 and 4.23 illustrate the weighted network diagrams generated from the model ensemble for linear chains with activation and with inhibition, respectively, at the threshold of 0.34. In both cases, the original regulation has been recovered. Overall, we have successfully demonstrated how the evoMEG algorithm can be used to generate a model ensemble for systems with irreversible reactions, reversible reactions, and regulations, how the model ensemble can be used to make reasonable predictions and direct future experiments, and how the algorithm responds to noise. We believe the evoMEG algorithm to be a useful tool for understanding the topology of the system from perturbation data.

4.3 metaMEG: evoMEG for Metabolic Networks

The evoMEG algorithm is primarily aimed to discover models where many of the interactions are unknown or unclear. However, researchers often have more information than simply the number of species participating in the network. For example, a large part of the carbon backbone is known for metabolic pathways but regulation via allosteric control may be uncertain. In such a case, it makes little sense to waste computational resources in searching through different topologies while only variations in regulations are required. Here, we demonstrate a modified version of our algorithm that aims to look specifically for models that have a fixed carbon backbone but with uncertain allosteric regulation called metaMEG (metabolic pathway Model Ensemble Generation). All computation and analysis performed in this section have been done on a combination of Intel i7 4770 processor running at 3.4 GHz with 8GB RAM and AMD Ryzen 1700X running at 3.8 GHz with 16GB RAM.

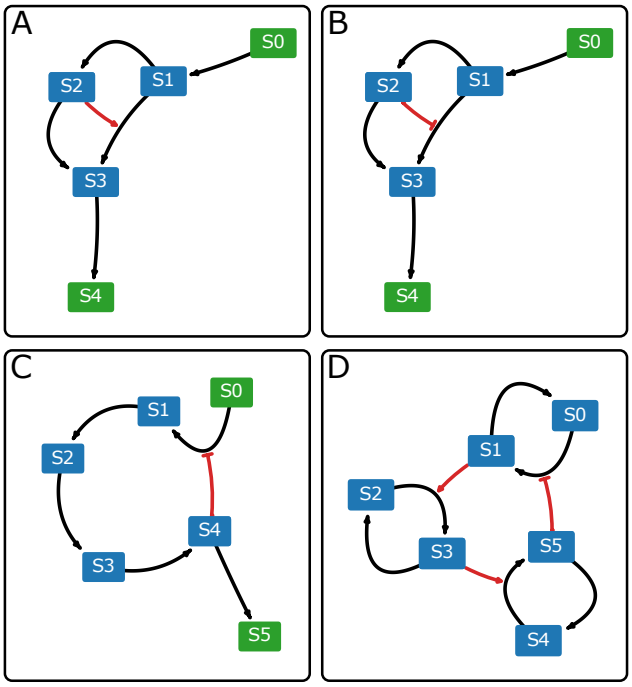


Figure 4.24: Network diagrams of models used as test cases: (A) feed-forward loop with activation, (B) feed-forward loop with inhibition, (C) linear chain with negative feedback, and (D) synthetic cascade. Nodes in green represent boundary species which are fixed.

4.3.1 Implementation

The metaMEG algorithm is similar to evoMEG, except for that the overall topology of the model is not changed through mutation or during random network generation. Instead, the algorithm attempts to generate unique models by adding regulations to the backbones. Regulations can be either activatory or inhibitory. Similar to evoMEG, only scaled concentration control coefficients are used to calculate the objective function. Only models with regulations are collected, meaning that the original model with a backbone is not counted. Overall, the metaMEG algorithm leads to faster convergence.

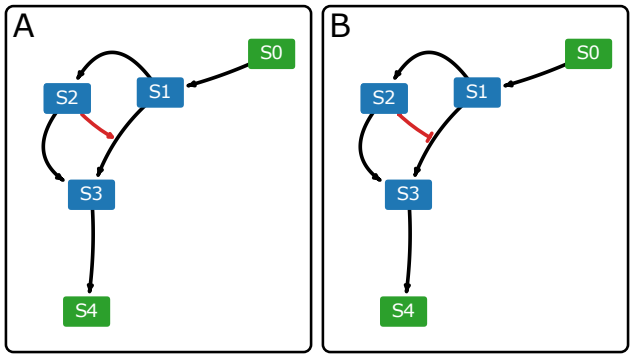


Figure 4.25: Network diagrams from the ensemble generated via metaMEG for feed-forward loops with activation or inhibition. Only the original models lead to good fit. Nodes in green represent boundary species which are fixed.

4.3.2 Results

We have tested the algorithm in several cases to measure its performance. Figure 4.24 illustrates some of the models we have used to test the algorithm, which include feed-forward loops with activation or inhibition, a simple linear chain with negative feedback, and a synthetic cascade. The test cases were used to generate scaled concentration control coefficients, which are in turn used as the synthetic experimental data. However, unlike evoMEG, the information on the reactions without regulation are also extracted to be used as the known backbone information to generate versions of the model with various regulations. The histograms of population fitness are presented in Appendix L and the convergence curves in Appendix M.

Feed-forward Loop with Activation or Inhibition

For feed-forward loops with activation or inhibition, the original models were recovered, and only the original models lead to the good fit (see Figure 4.25). Other models in the ensemble have distances more than four times larger than that of the original models. Appendix I presents weighted network diagrams for feed-forward loops with activation or inhibition with

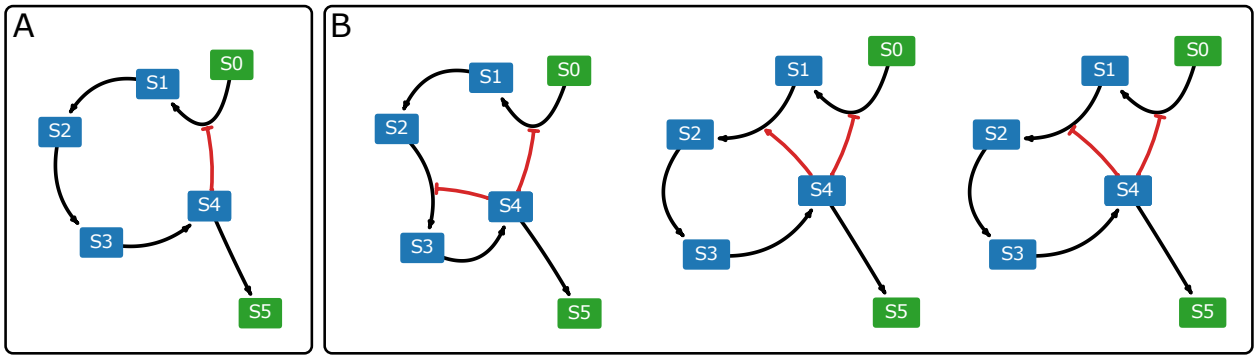


Figure 4.26: Network diagrams from the ensemble generated via metaMEG for the linear chain with a negative feedback. (A) The original model gives the best fit. (B) The algorithm also collected variations of the original model where additional regulations target downstream reactions. However, all models with acceptable fitness have the original negative feedback. Nodes in green represent boundary species which are fixed.

the threshold of 0.34.

Linear Chain with a Negative Feedback

For the linear chain with negative feedback, the original model has been recovered as shown in Figure 4.26A. The original topology has the best fitness, but unlike the case of feed-forward loops with activation or inhibition, some of the models in the population have acceptable distances, less than twice of that of the original model. The population has been filtered through kernel density estimation, which results in a model ensemble with 22 models. Figure 4.26B illustrates some of the other models in the ensemble with acceptable fitness. All of the models in the ensemble have the original negative feedback, as shown in Figure 4.27, but the models also have species $S4$ regulating some of the downstream reactions (see Figure 4.26).

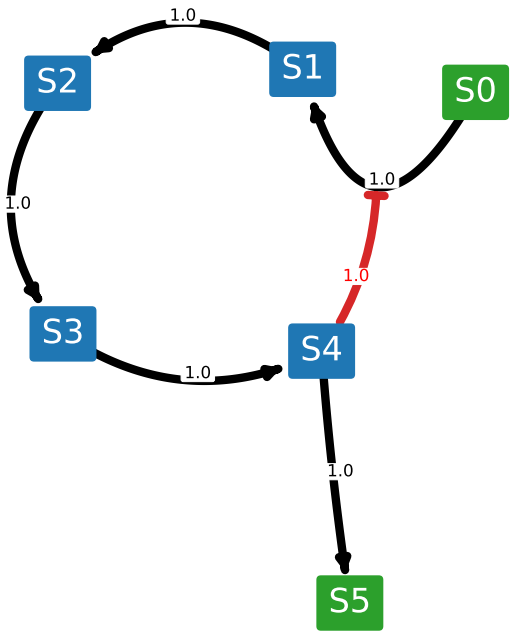


Figure 4.27: Weighted network diagram from the ensemble generated via metaMEG for the linear chain with a negative feedback at the threshold of 0.34. The negative feedback from species $S4$ to the reaction between species $S0$ and species $S1$ is present in all models in the ensemble. Nodes in green represent boundary species which are fixed.

Synthetic Cascade

For the synthetic cascade, the algorithm has generated an ensemble with various models with similar values of fitness. The population has been filtered through kernel density estimation, which results in a model ensemble with 25 models. When the weighted network diagram has been generated using the models with good fit and for a threshold of 0.34, the result looks generally consistent with the original model (see Figure 4.28). The interactions between cycles have two variations that could produce similar outputs. If a species is working as an activator in one part of a cycle, it can also work as an inhibitor in the other part of the cycle. The algorithm, however, was able to somewhat partially recover the activation by species

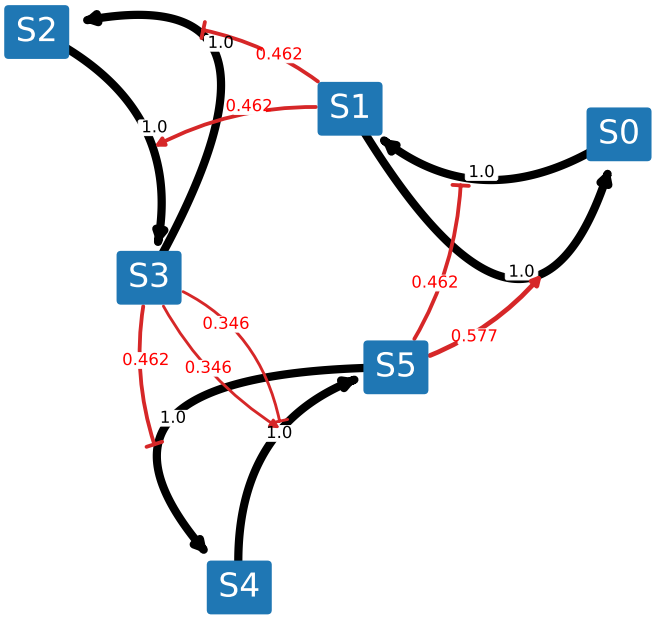


Figure 4.28: Weighted network diagram from the ensemble generated via metaMEG for the synthetic cascade at the threshold of 0.34. If a species works as an activator in one part of a cycle, it can also work as an inhibitor in the other part of the cycle.

$S3$. Instead, the algorithm could determine only that species $S3$ was somehow regulating the cycle between species $S4$ and $S5$.

Models without Regulations

We have also tested versions of the cases without any regulations, which serve as the negative control. The purpose of this test is to see how the algorithm responds in models without regulations. When the system of interest does not have regulations, the models collected by the algorithm should give bad fit. The models, which are illustrated in Figure 4.29, are identical to the test cases illustrated in Figure 4.24 except for that all regulations are removed.

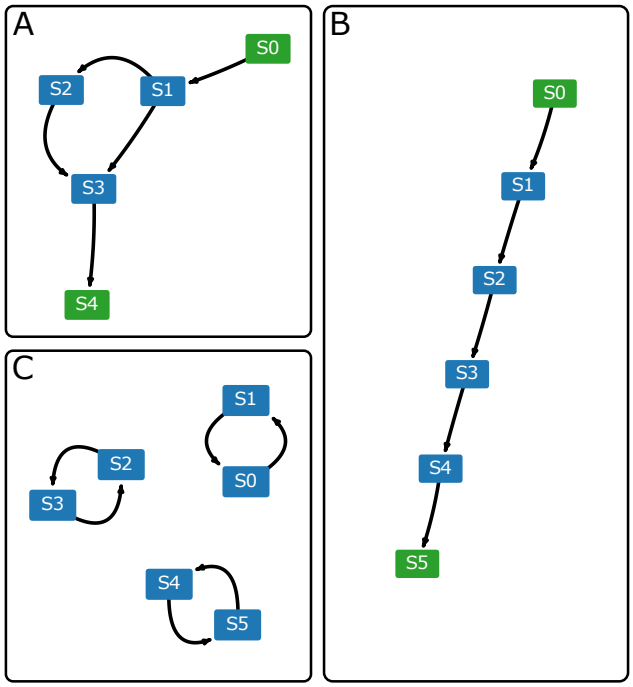


Figure 4.29: Network diagrams of models used for negative control. Models are identical to those illustrated in Figure 4.24 except for that all regulations has been removed. Nodes in green represent boundary species which are fixed.

When the test cases are presented to the algorithm, none of the models in the resulting ensemble give good fit as we expected. Moreover, when weighted network diagrams are plotted, no distinguishable patterns emerge (see Appendix K), unlike when regulations are actually present in the model as shown in Figures 4.27 and 4.28. See Appendix J for comparison with full weighted network diagrams from which Figures 4.27 and 4.28 are generated. The result tells us that when the algorithm runs on a system without regulations, the resulting ensemble has poor fitness with no discernible patterns; this should be taken as an indication that the system does not contain regulations.

When the backbone information is known, the standard procedure should be to run the algorithm twice: one with the experimental data and the other with a synthetic data generated from the model with backbone information. Then we can compare the model

ensembles to make sure the system of interest does contain regulations.

The metaMEG algorithm is a good example of how to integrate the prior knowledge of the system and to optimize topological search studies. Similarly, the evoMEG algorithm can integrate the prior knowledge not only of the entire backbone but also of a partial backbone and/or regulations.

Chapter 5

CONCLUSION

In this manuscript, we have discussed the current state of computational systems biology with regards to simulation environments, software tools, reproducibility, and modeling algorithms. In order to set a groundwork for improving reproducibility, robustness, and reliability of computational systems biology models, we identified bottlenecks and attempted to fix the issue. First, considering the issue of lack of high-performance and extensible modeling and simulation environments in Python, we developed Tellurium accordingly. We also noticed lack of Python-based network diagram visualization and analysis tools, and built the netplotlib package. Next, we have presented software tools to improve reproducibility, especially for simulation experiments and synthetic designs. Specifically, we built Python to the SED-ML Converter and phraSED-ML, making it possible to control fully SED-ML files in Python. As a result, Python users would be able to access the COMBINE archive format, which is a container with all information necessary for reproducing an output. We also built pySBOL, a Python binding for libSBOL, to support the SBOL standard. Finally, to improve the robustness and reliability of biochemical reaction network models, we designed two algorithms: the network search space reduction algorithm and the evoMEG algorithm. Both algorithms utilize recent developments in experimental techniques for high-resolution, combinatorial perturbation studies. We have demonstrated that the network search space reduction algorithm, making use of qualitative and combinatoric steady-state information, provides a fast and substantial reduction in the model search space. We have also showcased the evoMEG algorithm, which is an evolutionary algorithm-based algorithm that generates a model ensemble through the use of scaled concentration control coefficients. Applying the algorithm to various exemplary cases, we have confirmed that the algorithm can recover

topologies and generate a reliable model ensemble. It has also been demonstrated that the model ensemble can be used to make reasonable predictions on the system of interest and further to direct future experiments. In this fashion, evoMEG increases the efficiency of experiments by bridging the disparity between data-driven modeling and modeling-driven data collection. We have then explored the properties of evoMEG such as how it performs against noisy data, reversible rate laws, and regulations, and designed a metabolic pathway-specific variation of evoMEG, which we call metaMEG. The metaMEG algorithm, which searches for regulations while keeping the carbon backbone intact, turns out to provide a more efficient way of building a model ensemble when prior knowledge of the system is available. These developments, all of which are made available as open source software (Appendix N), should present exciting new opportunities for computational systems biology.

BIBLIOGRAPHY

[1] John H Abel, Brian Drawert, Andreas Hellander, and Linda R Petzold. Gillespy: a python package for stochastic model building and simulation. *IEEE life sciences letters*, 2(3):35–38, 2016.

[2] Richard R Adams. Sed-ed, a workflow editor for computational biology experiments written in sed-ml. *Bioinformatics*, 28(8):1180–1181, 2012.

[3] Reka Albert. Scale-free networks in cell biology. *Journal of cell science*, 118(21):4947–4957, 2005.

[4] Uri Alon. Network motifs: theory and experimental approaches. *Nature Rev Genet*, 8(6):450–461, 2007.

[5] David Angeli, James E. Ferrell, and Eduardo D. Sontag. Detection of multistability, bifurcations, and hysteresis in a large class of biological positive-feedback systems. *Proc. Natl. Acad. Sci. USA*, 101(7):1822–1827, 2004.

[6] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, 2016.

[7] Lorena A Barba. Terminologies for reproducible research. *arXiv preprint arXiv:1802.03311*, 2018.

[8] Bryan A Bartley, Kiri Choi, Meher Samineni, Zach Zundel, Tramy Nguyen, Chris J Myers, and Herbert M Sauro. pysbol: A python package for genetic design automation and standardization. *ACS synthetic biology*, 2018.

[9] Jacob Beal, Robert Sidney Cox, Raik Grünberg, James McLaughlin, Tramy Nguyen, Bryan Bartley, Michael Bissell, Kiri Choi, Kevin Clancy, Chris Macklin, et al. Synthetic biology open language (sbol) version 2.1. 0. *Journal of Integrative Bioinformatics*, 13(3):30–132, 2016.

[10] Yosef Bedaso, Frank T. Bergmann, Kiri Choi, Kyle Medley, and Herbert M. Sauro. A portable structural analysis library for reaction networks. *Biosystems*, 169-170:20 – 25, 2018.

[11] C. Glenn Begley and Lee M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, Mar 2012.

[12] Frank Bergmann, David Nickerson, Vincent Nol, and Kyle Medley. fbergmann/libsedml: libsedml 0.4.3, September 2017.

[13] Frank T Bergmann. Sed-ml script language. *Nature Precedings*, 2011.

[14] Frank T. Bergmann, Richard Adams, Stuart Moodie, Jonathan Cooper, Mihai Glont, Martin Golebiewski, Michael Hucka, Camille Laibe, Andrew K. Miller, David P. Nickerson, Brett G. Olivier, Nicolas Rodriguez, Herbert M. Sauro, Martin Scharm, Stian Soiland-Reyes, Dagmar Waltemath, Florent Yvon, and Nicolas Le Novère. Combine archive and omex format: one file to share all information to reproduce a modeling project. *BMC bioinformatics*, 15:369, Dec 2014.

[15] Frank T. Bergmann and Sarah M. Keating. libCombine: a C++ API library supporting the COMBINE Archive, September 2016.

[16] Frank T Bergmann, David Nickerson, Dagmar Waltemath, and Martin Scharm. Sed-ml web tools: generate, modify and export standard-compliant simulation studies. *Bioinformatics*, 33(8):1253–1254, 2017.

[17] Frank T Bergmann and Herbert M Sauro. Sbw-a modular framework for systems biology. In *Proceedings of the 38th conference on Winter simulation*, pages 1637–1645. Winter Simulation Conference, 2006.

[18] Frank T Bergmann, Ravishankar R Vallabhajosyula, and Herbert M Sauro. Computational tools for modeling protein networks. *Current Proteomics*, 3(3):181–197, 2006.

[19] Richard Bonneau, David J Reiss, Paul Shannon, Marc Facciotti, Leroy Hood, Nitin S Baliga, and Vesteinn Thorsson. The inferelator: an algorithm for learning parsimonious regulatory networks from systems-biology data sets de novo. *Genome Biol*, 7(5):R36, 2006.

[20] Nikolay Borisov, Edita Aksamitiene, Anatoly Kiyatkin, Stefan Legewie, Jan Berkhout, Thomas Maiwald, Nikolai P Kaimachnikov, Jens Timmer, Jan B Hoek, and Boris N Kholodenko. Systems-level interactions between insulin–egf networks amplify mitogenic signaling. *Molecular systems biology*, 5(1):256, 2009.

[21] Benjamin J Bornstein, Sarah M Keating, Akiya Jouraku, and Michael Hucka. Libsbml: an api library for sbml. *Bioinformatics*, 24(6):880–881, 2008.

[22] Eugene C Butcher, Ellen L Berg, and Eric J Kunkel. Systems biology in drug discovery. *Nat Biotechnol*, 22(10):1253–1259, 2004.

[23] Saikat Chakrabarti and Anna R Panchenko. Ensemble approach to predict specificity determinants: benchmarking and validation. *BMC bioinformatics*, 10(1):207, 2009.

[24] Deepak Chandran, Frank T Bergmann, and Herbert M Sauro. Tinkercell: modular cad tool for synthetic biology. *Journal of biological engineering*, 3(1):19, 2009.

[25] Christophe Chassagnole, Naruemol Noisommit-Rizzi, Joachim W Schmid, Klaus Mauch, and Matthias Reuss. Dynamic modeling of the central carbon metabolism of escherichia coli. *Biotechnology and bioengineering*, 79(1):53–73, 2002.

[26] Alejandro Chavez, Jonathan Scheiman, Suhani Vora, Benjamin W Pruitt, Marcelle Tuttle, Eswar PR Iyer, Shuailiang Lin, Samira Kiani, Christopher D Guzman, Daniel J Wiegand, et al. Highly efficient cas9-mediated transcriptional programming. *Nat Methods*, 12(4):326–328, 2015.

[27] Albert W Cheng, Haoyi Wang, Hui Yang, Linyu Shi, Yarden Katz, Thorold W Theunissen, Sudharshan Rangarajan, Chikdu S Shivalila, Daniel B Dadon, and Rudolf Jaenisch. Multiplexed activation of endogenous genes by crispr-on, an rna-guided transcriptional activator system. *Cell Res*, 23(10):1163–1171, 2013.

[28] Vijay Chickarmane, Carl Troein, Ulrike A Nuber, Herbert M Sauro, and Carsten Peterson. Transcriptional dynamics of the embryonic stem cell switch. *PLoS Comput. Biol.*, 2(9):e123, 09 2006.

[29] Kiri Choi. Robust approaches to generating reliable predictive models in systems biology. In *Systems Biology*, pages 301–312. Springer, 2018.

[30] Kiri Choi, Joseph Hellerstein, Steven Wiley, and Herbert M Sauro. Inferring reaction networks using perturbation data. *bioRxiv*, page 351767, 2018.

[31] Kiri Choi, J. Kyle Medley, Matthias Knig, Kaylene Stocking, Lucian Smith, Stanley Gu, and Herbert M. Sauro. Tellurium: An extensible python-based modeling environment for systems and synthetic biology. *Biosystems*, 171:74 – 79, 2018.

[32] Kiri Choi, Lucian P Smith, J Kyle Medley, and Herbert M Sauro. phrased-ml: A paraphrased, human-readable adaptation of sed-ml. *Journal of bioinformatics and computational biology*, 14(06):1650035, 2016.

[33] Robert Sidney Cox, Curtis Madsen, James Alastair McLaughlin, Tramy Nguyen, Nicholas Roehner, Bryan Bartley, Jacob Beal, Michael Bissell, Kiri Choi, Kevin Clancy, et al. Synthetic biology open language (sbol) version 2.2. 0. *Journal of integrative bioinformatics*, 15(1), 2018.

[34] Bryan C Daniels, Yan-Jiun Chen, James P Sethna, Ryan N Gutenkunst, and Christopher R Myers. Sloppiness, robustness, and evolvability in systems biology. *Current opinion in biotechnology*, 19(4):389–395, 2008.

[35] Bryan C Daniels and Ilya Nemenman. Efficient inference of parsimonious phenomenological models of cellular dynamics using s-systems and alternating regression. *PLoS ONE*, 10(3):e0119821, 2015.

[36] John W Davey, Paul A Hohenlohe, Paul D Etter, Jason Q Boone, Julian M Catchen, and Mark L Blaxter. Genome-wide genetic marker discovery and genotyping using next-generation sequencing. *Nature Rev Genet*, 12(7):499–510, 2011.

[37] Pedro de Atauri, Adrian Benito, Pedro Vizán, Miriam Zanuy, Ramón Mangues, Silvia Marín, and Marta Cascante. Carbon metabolism and the sign of control coefficients in metabolic adaptations underlying k-ras transformation. *Biochimica et Biophysica Acta (BBA)-Bioenergetics*, 1807(6):746–754, 2011.

[38] Dennis DeCoste. Collaborative prediction using ensembles of maximum margin matrix factorizations. In *Proceedings of the 23rd international conference on Machine learning*, pages 249–256. ACM, 2006.

[39] E. J. Doedel. Auto: a program for the automatic bifurcation analysis of autonomous systems. In *Proc. Manitoba Conf. Num. Math. Comput., 10th, Winnipeg, Canada*, 1981. [Congressus Numeratium, 30:265–284].

[40] Ali Ebrahim, Joshua A Lerman, Bernhard O Palsson, and Daniel R Hyduke. Cobrapy: Constraints-based reconstruction and analysis for python. *BMC systems biology*, 7(1):74, 2013.

[41] M. B Elowitz and S Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403:335–338, 2000.

[42] G. Bard Ermentrout and David H Terman. *Mathematical foundations of neuroscience*. Interdisciplinary Applied Mathematics (Book 35). New York Springer, xvi, 2010.

[43] Jasmin Fisher and Steven Woodhouse. Program synthesis meets deep learning for decoding regulatory networks. *Curr Opin Syst Biol*, 4:64–70, 2017.

[44] Akira Funahashi, Yukiko Matsuoka, Akiya Jouraku, Mineo Morohashi, Norihiro Kikuchi, and Hiroaki Kitano. Celldesigner 3.5: A versatile modeling tool for biochemical networks. *Proc. IEEE*, 96(8):1254–1265, 2008.

[45] Akira Funahashi, Mineo Morohashi, Hiroaki Kitano, and Naoki Tanimura. Celldesigner: a process diagram editor for gene-regulatory and biochemical networks. *Biosilico*, 1(5):159–162, 2003.

[46] Michal Galdzicki, Kevin P Clancy, Ernst Oberortner, Matthew Pocock, Jacqueline Y Quinn, Cesar A Rodriguez, Nicholas Roehner, Mandy L Wilson, Laura Adam, J Christopher Anderson, et al. The synthetic biology open language (sbol) provides a community standard for communicating designs in synthetic biology. *Nature biotechnology*, 32(6):545–550, 2014.

[47] Michal Galdzicki, Cesar Rodriguez, Deepak Chandran, Herbert M. Sauro, and John H. Gennari. Standard biological parts knowledgebase. *PLoS ONE*, 6(2):e17005, 02 2011.

[48] Alan Garny and Peter J Hunter. Opencor: a modular and interoperable approach to computational biology. *Frontiers in physiology*, 6:26, 2015.

[49] Vasileios L Georgiou, Philipos D Alevizos, and Michael N Vrahatis. Novel approaches to probabilistic neural networks through bagging and evolutionary estimating of prior probabilities. *Neural Processing Letters*, 27(2):153–162, 2008.

[50] Luke A Gilbert, Max A Horlbeck, Britt Adamson, Jacqueline E Villalta, Yuwen Chen, Evan H Whitehead, Carla Guimaraes, Barbara Panning, Hidde L Ploegh, Michael C Bassik, et al. Genome-scale crispr-mediated control of gene repression and activation. *Cell*, 159(3):647–661, 2014.

[51] Luke J Gosink, Emilie A Hogan, Trenton C Pulsipher, and Nathan A Baker. Bayesian model aggregation for ensemble-based estimates of protein pka values. *Proteins: Structure, Function, and Bioinformatics*, 82(3):354–363, 2014.

[52] Ryan N Gutenkunst, Joshua J Waterfall, Fergal P Casey, Kevin S Brown, Christopher R Myers, and James P Sethna. Universally sloppy parameter sensitivities in systems biology models. *PLoS computational biology*, 3(10):e189, 2007.

[53] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[54] W. J. Hedley, N. R. Melanie, D. P. Bullivant, and P. F. Nielson. A Short Introduction to CellML. *Phil. Trans. Roy. Soc. London A*, 359:1073–1089, 2001.

[55] David Henriques, Alejandro F Villaverde, Miguel Rocha, Julio Saez-Rodriguez, and Julio R Banga. Data-driven reverse engineering of signaling pathways using ensembles of dynamic models. *PLoS Comput Biol*, 13(2):e1005379, 2017.

[56] Jan-Hendrik S Hofmeyr. Metabolic control analysis in a nutshell. In *Proceedings of the 2nd International conference on systems biology*, pages 291–300. Omnipress Madison, Wisconsin, 2001.

[57] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. Copasia complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[58] Michael Hucka, Andrew Finney, Herbert M. Sauro, Hamid Bolouri, John C. Doyle, Hiroaki Kitano, Adam P. Arkin, Benjamin J. Bornstein, Dennis Bray, Athel Cornish-Bowden, Autumn A. Cuellar, Sergey Dronov, Ernst D. Gilles, Martin Ginkel, Victoria Gor, Igor I. Goryanin, Warren J. Hedley, T. Charles Hodgman, Jan-Hendrik S. Hofmeyr, Peter J. Hunter, Nick S. Juty, Jay L. Kasberger, Andreas Kremling, Ursula Kummer, Nicolas Le Novre, Leslie M. Loew, Daniel Lucio, Pedro Mendes, Eric Minch, Eric D. Mjolsness, Yoichi Nakayama, Melanie R. Nelson, Poul F. Nielsen, Takeshi Sakurada, James C. Schaff, Bruce E. Shapiro, Thomas S. Shimizu, Hugh D. Spence, Jrg Stelling, Kouichi Takahashi, Masaru Tomita, John M. Wagner, Jian Wang, and the rest of the SBML Forum. The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[59] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.

[60] B. P. Ingalls. A Frequency Domain Approach to Sensitivity Analysis of Biochemical Systems. *Journal of Physical Chemistry B*, 108:1143–1152, 2004.

[61] Wolf Jana and Reinhart Heinrich. Effect of cellular interaction on glycolytic oscillations in yeast: a theoretical investigation. *Biochemical Journal*, 345(2):321–334, 2000.

[62] Jianhua Jia, Zi Liu, Xuan Xiao, Bingxiang Liu, and Kuo-Chen Chou. psuc-lys: predict lysine succinylation sites in proteins with pseaac and ensemble random forest approach. *Journal of theoretical biology*, 394:223–230, 2016.

[63] M Chris Jones, James S Marron, and Simon J Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American statistical association*, 91(433):401–407, 1996.

[64] H. Kacser and J. A. Burns. The Control of Flux. In D. D. Davies, editor, *Rate Control of Biological Processes*, volume 27 of *Symp. Soc. Exp. Biol.*, pages 65–104. Cambridge University Press, 1973.

[65] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7 – 15, 1989.

[66] Jonathan R Karr, Jayodita C Sanghvi, Derek N Macklin, Miriam V Gutschow, Jared M Jacobs, Benjamin Bolival, Nacyra Assad-Garcia, John I Glass, and Markus W Covert. A whole-cell computational model predicts phenotype from genotype. *Cell*, 150(2):389–401, 2012.

[67] Roland Keller, Alexander Dörr, Akito Tabira, Akira Funahashi, Michael J Ziller, Richard Adams, Nicolas Rodriguez, Nicolas L Novère, Noriko Hiroi, Hannes Planatscher, et al. The systems biology simulation core algorithm. *BMC systems biology*, 7(1):55, 2013.

[68] Boris N Kholodenko. Negative feedback and ultrasensitivity can bring about oscillations in the mitogen-activated protein kinase cascades. *European Journal of Biochemistry*, 267(6):1583–1588, 2000.

[69] Hiroaki Kitano. Computational systems biology. *Nature*, 420(6912):206–210, 2002.

[70] Hiroaki Kitano. Systems biology: a brief overview. *Science*, 295(5560):1662–1664, 2002.

[71] Lars Kuepfer, Matthias Peter, Uwe Sauer, and Jörg Stelling. Ensemble modeling for analysis of cell signaling dynamics. *Nature biotechnology*, 25(9):1001, 2007.

[72] Petr Kuzmi. Program dynafit for the analysis of enzyme kinetic data: Application to hiv proteinase. *Anal. Biochem.*, 237(2):260 – 273, 1996.

[73] Yun Lee, Jimmy G Lafontaine Rivera, and James C Liao. Ensemble modeling for robustness analysis in engineering non-native metabolic pathways. *Metabolic engineering*, 25:63–71, 2014.

[74] Shuzhao Li, Youngja Park, Sai Duraisingham, Frederick H Strobel, Nooruddin Khan, Quinlyn A Soltow, Dean P Jones, and Bali Pulendran. Predicting network activity from high throughput metabolomics. *PLoS Comput Biol*, 9(7):e1003123, 2013.

[75] Wolfram Liebermeister, Jannis Uhlendorf, and Edda Klipp. Modular rate laws for enzymatic reactions: thermodynamics, elasticities and implementation. *Bioinformatics*, 26(12):1528–1534, 2010.

[76] Catherine M Lloyd, Matt DB Halstead, and Poul F Nielsen. Cellml: its future, present and past. *Progress in biophysics and molecular biology*, 85(2):433–450, 2004.

[77] Carlos F Lopez, Jeremy L Muhlich, John A Bachman, and Peter K Sorger. Programming biological models in python using pysb. *Molecular systems biology*, 9(1):646, 2013.

[78] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[79] Niall M Mangan, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Inferring biological networks by sparse identification of nonlinear dynamics. *IEEE Trans Mol Biol Multi-Scale Commun*, 2(1):52–63, 2016.

[80] Joseph A Marsh and Julie D Forman-Kay. Ensemble modeling of protein disordered states: experimental restraint contributions and validation. *Proteins: Structure, Function, and Bioinformatics*, 80(2):556–572, 2012.

[81] Kevin A McGoff, Xin Guo, Anastasia Deckard, Christina M Kelliher, Adam R Leman, Lauren J Francey, John B Hogenesch, Steven B Haase, and John L Harer. The local edge machine: inference of dynamic models of gene regulation. *Genome Biol*, 17(1):214, 2016.

[82] James Alastair McLaughlin, Chris J Myers, Zach Zundel, Goksel Mısırlı, Michael Zhang, Irina Dana Ofiteru, Angel Goni-Moreno, and Anil Wipat. Synbiohub: A standards-enabled design repository for synthetic biology. *ACS synthetic biology*, 7(2):682–688, 2018.

[83] James Alastair McLaughlin, Chris J Myers, Zach Zundel, Nathan Wilkinson, Christian Atallah, and Anil Wipat. sboljs: Bringing the synthetic biology open language to the web browser. *ACS synthetic biology*, 8(1):191–193, 2018.

[84] J. Kyle Medley, Kiri Choi, Matthias Knig, Lucian Smith, Stanley Gu, Joseph Hellerstein, Stuart C. Sealfon, and Herbert M. Sauro. Tellurium notebooksan environment for reproducible dynamical modeling in systems biology. *PLOS Computational Biology*, 14(6):1–24, 06 2018.

[85] Nathan Mih, Elizabeth Brunk, Ke Chen, Edward Catoiu, Anand Sastry, Erol Kavvas, Jonathan M Monk, Zhen Zhang, and Bernhard O Palsson. ssbio: a python framework for structural systems biology. *Bioinformatics*, 34(12):2155–2157, 2018.

[86] Pierre Millard, Kieran Smallbone, and Pedro Mendes. Metabolic regulation is sufficient for global and robust coordination of glucose uptake, catabolism, energy production and growth in escherichia coli. *PLoS Comput Biol*, 13(2):e1005396, 2017.

[87] Ion I Moraru, James C Schaff, Boris M Slepchenko, ML Blinov, Frank Morgan, Anuradha Lakshminarayana, Fei Gao, Yuhua Li, and Leslie M Loew. Virtual cell modelling and simulation software environment. *IET Syst. Biol.*, 2(5):352–362, 2008.

[88] C. R. Myers, R. N. Gutenkunst, and J. P. Sethna. Python Unleashed on Systems Biology. *ArXiv e-prints.*, April 2007.

[89] Chris J. Myers, Nathan Barker, Kevin Jones, Hiroyuki Kuwahara, Curtis Madsen, and Nam-Phuong D. Nguyen. ibiosim: a tool for the analysis and design of genetic circuits. *Bioinformatics*, 25(21):2848–2849, 2009.

[90] Christopher R. Myers, Ryan N. Gutenkunst, and James P. Sethna. Python Unleashed on Systems Biology. *Computing in Science and Engg.*, 9(3):34–37, 2007.

[91] Joseph L Natale, David Hofmann, Damián G Hernández, and Ilya Nemenman. Reverse-engineering biological networks from large data sets. *arXiv preprint arXiv:1705.06370*, 2017.

[92] Maxwell L Neal, Christopher T Thompson, Karam G Kim, Ryan C James, Daniel L Cook, Brian E Carlson, and John H Gennari. Semgen: a tool for semantics-based annotation and composition of biosimulation models. *Bioinformatics*, 2018.

[93] Maxwell Lewis Neal, Matthias König, David Nickerson, Göksel Mısırlı, Reza Kalbasi, Andreas Dräger, Koray Atalag, Vijayalakshmi Chelliah, Michael T Cooling, Daniel L Cook, et al. Harmonizing semantic annotations for computational models in biology. *Briefings in bioinformatics*, 20(2):540–550, 2018.

[94] J. A. Nelder and R. Mead. A simplex method for function minimization. *Comput. J.*, 7(4):308–313, 1965.

[95] Matthew Newville, Till Stensitzki, Daniel B. Allen, and Antonino Ingargiola. Lmfit: Non-linear least-square minimization and curve-fitting for python, September 2014.

[96] Chris J Oates, Frank Dondelinger, Nora Bayani, James Korkola, Joe W Gray, and Sach Mukherjee. Causal network inference using biochemical kinetics. *Bioinformatics*, 30(17):i468–i474, 2014.

[97] Brett G. Olivier, Johann M. Rohwer, and Jan-Hendrik S. Hofmeyr. Modelling cellular systems with pysces. *Bioinformatics*, 21(4):560–561, 2005.

[98] Brett G Olivier and Jacky L Snoep. Web-based kinetic modelling using jws online. *Bioinformatics*, 20(13):2143–2144, 2004.

[99] Wei Pan, Ye Yuan, Jorge Gonçalves, and Guy-Bart Stan. A sparse bayesian approach to the identification of nonlinear state-space systems. *IEEE Trans Automat Contr*, 61(1):182–187, 2016.

[100] Wendy S Parker. Ensemble modeling, uncertainty and robust predictions. *Wiley Interdisciplinary Reviews: Climate Change*, 4(3):213–223, 2013.

[101] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[102] Florian Prinz, Thomas Schlange, and Khusru Asadullah. Believe it or not: how much can we rely on published data on potential drug targets? *Nat Rev Drug Discov*, 10(9):712–712, Sep 2011.

[103] Lei S Qi, Matthew H Larson, Luke A Gilbert, Jennifer A Doudna, Jonathan S Weissman, Adam P Arkin, and Wendell A Lim. Repurposing crispr as an rna-guided platform for sequence-specific control of gene expression. *Cell*, 152(5):1173–1183, 2013.

[104] Paul J Roebber, David M Schultz, Brian A Colle, and David J Stensrud. Toward improved prediction: High-resolution and ensemble modeling systems in operations. *Weather and Forecasting*, 19(5):936–949, 2004.

[105] Murray Rosenblatt. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, pages 832–837, 1956.

[106] Yvan Saeys, Thomas Abeel, and Yves Van de Peer. Robust feature selection using ensemble feature selection techniques. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 313–325. Springer, 2008.

[107] H M Sauro. *Systems Biology: An Introduction to Pathway Modeling*. Ambrosius Publishing, Seattle, 2014.

[108] H. M. Sauro and B. Ingalls. Conservation analysis in biochemical networks: computational issues for software writers. *Biophys Chem*, 109(1):1–15, Apr 2004.

[109] Herbert M. Sauro. Jdesigner: A simple biochemical network designer. *Available via the World Wide Web at http://members.tripod.co.uk/sauro/biotech.htm*, 2001.

[110] Herbert M Sauro. *Enzyme kinetics for systems biology.* Ambrosius Publishing, 2 edition, 2012.

[111] Herbert M Sauro. *Systems Biology: Introduction to Pathway Modeling.* Ambrosius Publishing, 1st edition, 2014.

[112] Herbert M. Sauro. Control and regulation of pathways via negative feedback. *Journal of The Royal Society Interface*, 14(127), 2017.

[113] Herbert M Sauro and John Barrett. In vitro control analysis of an enzyme system: Experimental and analytical developments. *Mol. Cell. Biochem.*, 145(2):141–150, 1995.

[114] Herbert M Sauro and DA Fell. Jarnac: a system for interactive metabolic analysis. In *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*, pages 221–228. Stellenbosch University Press, 2000.

[115] Jörg Schaber and Edda Klipp. Model-based inference of biochemical parameters and dynamic properties of microbial signal transduction networks. *Current Opinion in Biotechnology*, 22(1):109–116, 2011.

[116] Asok K Sen. On the sign pattern of metabolic control coefficients. *Journal of theoretical biology*, 182(3):269–275, 1996.

[117] Daniel C Sévin, Tobias Fuhrer, Nicola Zamboni, and Uwe Sauer. Nontargeted in vitro metabolomics for high-throughput identification of novel enzymes in escherichia coli. *Nat Methods*, 14(2):187–194, 2017.

[118] Simon J Sheather and Michael C Jones. A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society: Series B (Methodological)*, 53(3):683–690, 1991.

[119] Tujin Shi, Thomas L Fillmore, Xuefei Sun, Rui Zhao, Athena A Schepmoes, Mahmud Hossain, Fang Xie, Si Wu, Jong-Seo Kim, Nathan Jones, et al. Antibody-free, targeted mass-spectrometric approach for quantification of proteins at low picogram per milliliter levels in human plasma/serum. *PNAS*, 109(38):15395–15400, 2012.

[120] Tujin Shi, Mario Niepel, Jason E McDermott, Yuqian Gao, Carrie D Nicora, William B Chrisler, Lye M Markillie, Vladislav A Petyuk, Richard D Smith, Karin D Rodland, et al. Conservation of protein abundance patterns reveals the regulatory architecture of the egfr-mapk pathway. *Sci Signal*, 9(436):rs6, 2016.

[121] Lucian P Smith, Frank T Bergmann, Deepak Chandran, and Herbert M Sauro. Antimony: a modular model definition language. *Bioinformatics*, 25(18):2452–2454, 2009.

[122] Endre T Somogyi, Jean-Marie Bouteiller, James A Glazier, Matthias König, J Kyle Medley, Maciej H Swat, and Herbert M Sauro. libroadrunner: a high performance sbml simulation and analysis library. *Bioinformatics*, 31(20):3315–3321, 2015.

[123] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[124] James W Taylor and Roberto Buizza. Neural network load forecasting with weather ensemble predictions. *IEEE Transactions on Power systems*, 17(3):626–632, 2002.

[125] Tsuyoshi Terakawa and Shoji Takada. Multiscale ensemble modeling of intrinsically disordered proteins: p53 n-terminal domain. *Biophysical journal*, 101(6):1450–1458, 2011.

[126] Linh M Tran, Matthew L Rizk, and James C Liao. Ensemble modeling of metabolic networks. *Biophysical journal*, 95(12):5606–5617, 2008.

[127] Erwin L Van Dijk, Hélène Auger, Yan Jaszczyszyn, and Claude Thermes. Ten years of next-generation sequencing technology. *Trends Genet*, 30(9):418–426, 2014.

[128] Marc T. Vass, Clifford A. Shaffer, Naren Ramakrishnan, Layne T. Watson, and John J. Tyson. The jigcell model builder: a spreadsheet interface for creating biochemical reaction network models. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 3(2):155–164, 2006.

[129] Alejandro F Villaverde, David Henriques, Kieran Smallbone, Sophia Bongard, Joachim Schmid, Damjan Cicin-Sain, Anton Crombach, Julio Saez-Rodriguez, Klaus Mauch, Eva Balsa-Canto, et al. Biopredyn-bench: a suite of benchmark problems for dynamic modelling in systems biology. *BMC systems biology*, 9(1):8, 2015.

[130] Dagmar Waltemath, Richard Adams, Frank T Bergmann, Michael Hucka, Fedor Kolpakov, Andrew K Miller, Ion I Moraru, David Nickerson, Sven Sahle, Jacky L Snoep, et al. Reproducible computational biology experiments with sed-ml-the simulation experiment description markup language. *BMC systems biology*, 5(1):1, 2011.

[131] Jinyuan Yan, Maxime Deforet, Kerry E Boyle, Rayees Rahman, Raymond Liang, Chinweike Okegbe, Lars EP Dietrich, Weigang Qiu, and Joao B Xavier. Bow-tie signaling in c-di-gmp: Machine learning in a simple biochemical network. *PLoS Comput Biol*, 13(8):e1005677, 2017.

[132] Michael Zhang, James Alastair McLaughlin, Anil Wipat, and Chris J Myers. Sboldesigner 2: an intuitive tool for structural genetic design. *ACS synthetic biology*, 6(7):1150–1160, 2017.

[133] Zhen Zhang, Tramy Nguyen, Nicholas Roehner, Göksel Misirli, Matthew Pocock, Ernst Oberortner, Meher Samineni, Zach Zundel, Jacob Beal, Kevin Clancy, et al. libsbolj 2.0: a java library to support sbol 2.0. *IEEE life sciences letters*, 1(4):34–37, 2015.

Appendix A

THE EXAMPLE SED-ML FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<sedML xmlns="http://sed-ml.org/sed-ml/level1/version2" xmlns:math="http://
   ↪ www.w3.org/1998/Math/MathML" level="1" version="2">
  <!--This file was generated by jlibsedml, version 2.2.3.-->
  <listOfSimulations>
    <uniformtime-course id="simId" initialTime="0.0" outputStartTime="0.0"
       ↪ outputEndTime="100.0" numberOfPoints="100">
    </uniformtime-course>
  </listOfSimulations>
  <listOfModels>
    <model id="modelId" language="urn:sedml:language:sbml" source="
       ↪ simpleModel.xml" />
  </listOfModels>
  <listOfTasks>
    <task id="task1" modelReference="modelId" simulationReference="simId" />
  </listOfTasks>
  <listOfDataGenerators>
    <dataGenerator id="time_dg" name="time">
      <listOfVariables>
        <variable id="time" taskReference="task1" symbol="urn:sedml:symbol:
           ↪ time" />
      </listOfVariables>
```

```
    <math:math>
      <math:ci>time</math:ci>
    </math:math>
  </dataGenerator>
  <dataGenerator id="S1_dg" name="S1">
    <listOfVariables>
      <variable id="S1" name="S1" taskReference="task1" target="/sbml:sbml
          ↪ /sbml:model/sbml:listOfSpecies/sbml:species[@id='S1']" />
    </listOfVariables>
    <math:math>
      <math:ci>S1</math:ci>
    </math:math>
  </dataGenerator>
  <dataGenerator id="S2_dg" name="S2">
    <listOfVariables>
      <variable id="S2" name="S2" taskReference="task1" target="/sbml:sbml
          ↪ /sbml:model/sbml:listOfSpecies/sbml:species[@id='S2']" />
    </listOfVariables>
    <math:math>
      <math:ci>S2</math:ci>
    </math:math>
  </dataGenerator>
</listOfDataGenerators>
<listOfOutputs>
  <plot2D id="graph" name="Figure 1">
    <listOfCurves>
```

```
      <curve id="c_S1" name="S1" logX="false" logY="false" xDataReference
         ↪ ="time_dg" yDataReference="S1_dg" />
      <curve id="c_S2" name="S2" logX="false" logY="false" xDataReference
         ↪ ="time_dg" yDataReference="S2_dg" />
    </listOfCurves>
   </plot2D>
  </listOfOutputs>
</sedML>
```

Appendix B

PYTHON SCRIPT TRANSLATED FROM THE EXAMPLE SED-ML FILE

```
import tellurium as te
from roadrunner import Config
from tellurium.sedml.mathml import *
from tellurium.sedml.tesedml import process_trace, terminate_trace,
    ↪ fix_endpoints

import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
try:
    import tesedml as libsedml
except ImportError:
    import libsedml
import pandas
import os.path
Config.LOADSBMLOPTIONS_RECOMPILE = True

workingDir = r'C:\Users\Kiri Choi\Desktop\temp'

# ----------------------------------------------------------
# Models
```

```
# ----------------------------------------------------------
# Model <modelId>
modelId = te.loadSBMLModel(os.path.join(workingDir, 'simpleModel.xml'))




# ----------------------------------------------------------
# Tasks
# ----------------------------------------------------------
# Task <task1>
# Task: <task1>
task1 = [None]
modelId.setIntegrator('cvode')
if modelId.conservedMoietyAnalysis == True: modelId.conservedMoietyAnalysis
    ↪ = False
modelId.time-courseSelections = ['[S2]', 'time', '[S1]']
modelId.reset()
task1[0] = modelId.simulate(start=0.0, end=100.0, steps=100)

# ----------------------------------------------------------
# DataGenerators
# ----------------------------------------------------------
# DataGenerator <time_dg>
__var__time = np.concatenate([sim['time'] for sim in task1])
if len(__var__time.shape) == 1:
    __var__time.shape += (1,)
```

```
time_dg = __var__time
# DataGenerator <S1_dg>
__var__S1 = np.concatenate([sim['[S1]'] for sim in task1])
if len(__var__S1.shape) == 1:
    __var__S1.shape += (1,)
S1_dg = __var__S1
# DataGenerator <S2_dg>
__var__S2 = np.concatenate([sim['[S2]'] for sim in task1])
if len(__var__S2.shape) == 1:
    __var__S2.shape += (1,)
S2_dg = __var__S2


# ------------------------------------------------------------
# Outputs
# ------------------------------------------------------------
# Output <graph>
_stacked = False
if _stacked:
    tefig = te.getPlottingEngine().newStackedFigure(title='Figure 1', xtitle
        ↪ ='time')
else:
    tefig = te.nextFigure(title='Figure 1', xtitle='time')


for k in range(time_dg.shape[1]):
    extra_args = {}
    if k == 0:
        extra_args['name'] = 'S1'
```

```
    tefig.addXYDataset(time_dg[:,k], S1_dg[:,k], color='#1f77b4', tag='tag0'
        ↪ , logx=False, logy=False, **extra_args)
for k in range(time_dg.shape[1]):
    extra_args = {}
    if k == 0:
        extra_args['name'] = 'S2'
    tefig.addXYDataset(time_dg[:,k], S2_dg[:,k], color='#ff7f0e', tag='tag1'
        ↪ , logx=False, logy=False, **extra_args)
if te.tiledFigure():

    if te.tiledFigure().renderIfExhausted():

        te.clearTiledFigure()

else:

    fig = tefig.render()
```

Appendix C

MODIFYING MAPK CASCADE MODEL FOR DIFFERENT PARAMETERIZATION

```
import tellurium as te
import tempfile, os

antimony_str1 = '''
model BorisEJB1

  // Compartments and Species:
  compartment compartment_;
  species MKKK in compartment_, MKKK_P in compartment_, MKK in
     ↪ compartment_;
  species MKK_P in compartment_, MKK_PP in compartment_, MAPK in
     ↪ compartment_;
  species MAPK_P in compartment_, MAPK_PP in compartment_;

  // Reactions:
  J0: MKKK => MKKK_P; (J0_V1*MKKK)/((1 + (MAPK_PP/J0_Ki)^J0_n)*(J0_K1 +
     ↪ MKKK));
  J1: MKKK_P => MKKK; (J1_V2*MKKK_P)/(J1_KK2 + MKKK_P);
  J2: MKK => MKK_P; (J2_k3*MKKK_P*MKK)/(J2_KK3 + MKK);
  J3: MKK_P => MKK_PP; (J3_k4*MKKK_P*MKK_P)/(J3_KK4 + MKK_P);
  J4: MKK_PP => MKK_P; (J4_V5*MKK_PP)/(J4_KK5 + MKK_PP);
```

```
J5: MKK_P => MKK; (J5_V6*MKK_P)/(J5_KK6 + MKK_P);
J6: MAPK => MAPK_P; (J6_k7*MKK_PP*MAPK)/(J6_KK7 + MAPK);
J7: MAPK_P => MAPK_PP; (J7_k8*MKK_PP*MAPK_P)/(J7_KK8 + MAPK_P);
J8: MAPK_PP => MAPK_P; (J8_V9*MAPK_PP)/(J8_KK9 + MAPK_PP);
J9: MAPK_P => MAPK; (J9_V10*MAPK_P)/(J9_KK10 + MAPK_P);

// Species initializations:
MKKK = 90;
MKKK_P = 10;
MKK = 280;
MKK_P = 10;
MKK_PP = 10;
MAPK = 280;
MAPK_P = 10;
MAPK_PP = 10;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
J0_V1 = 2.5;
J0_Ki = 9;
J0_n = 1;
J0_K1 = 10;
J1_V2 = 0.25;
J1_KK2 = 8;
J2_k3 = 0.025;
```

```
  J2_KK3 = 15;
  J3_k4 = 0.025;
  J3_KK4 = 15;
  J4_V5 = 0.75;
  J4_KK5 = 15;
  J5_V6 = 0.75;
  J5_KK6 = 15;
  J6_k7 = 0.025;
  J6_KK7 = 15;
  J7_k8 = 0.025;
  J7_KK8 = 15;
  J8_V9 = 0.5;
  J8_KK9 = 15;
  J9_V10 = 0.5;
  J9_KK10 = 15;

  // Other declarations:
  const compartment_, J0_V1, J0_Ki, J0_n, J0_K1, J1_V2, J1_KK2, J2_k3,
     ↪ J2_KK3;
  const J3_k4, J3_KK4, J4_V5, J4_KK5, J5_V6, J5_KK6, J6_k7, J6_KK7, J7_k8;
  const J7_KK8, J8_V9, J8_KK9, J9_V10, J9_KK10;
end
'''

antimony_str2 = '''
model BorisEJB2
```

```
// Compartments and Species:
compartment compartment_;
species MKKK in compartment_, MKKK_P in compartment_, MKK in
   ↪ compartment_;
species MKK_P in compartment_, MKK_PP in compartment_, MAPK in
   ↪ compartment_;
species MAPK_P in compartment_, MAPK_PP in compartment_;

// Reactions:
J0: MKKK => MKKK_P; (J0_V1*MKKK)/((1 + (MAPK_PP/J0_Ki)^J0_n)*(J0_K1 +
   ↪ MKKK));
J1: MKKK_P => MKKK; (J1_V2*MKKK_P)/(J1_KK2 + MKKK_P);
J2: MKK => MKK_P; (J2_k3*MKKK_P*MKK)/(J2_KK3 + MKK);
J3: MKK_P => MKK_PP; (J3_k4*MKKK_P*MKK_P)/(J3_KK4 + MKK_P);
J4: MKK_PP => MKK_P; (J4_V5*MKK_PP)/(J4_KK5 + MKK_PP);
J5: MKK_P => MKK; (J5_V6*MKK_P)/(J5_KK6 + MKK_P);
J6: MAPK => MAPK_P; (J6_k7*MKK_PP*MAPK)/(J6_KK7 + MAPK);
J7: MAPK_P => MAPK_PP; (J7_k8*MKK_PP*MAPK_P)/(J7_KK8 + MAPK_P);
J8: MAPK_PP => MAPK_P; (J8_V9*MAPK_PP)/(J8_KK9 + MAPK_PP);
J9: MAPK_P => MAPK; (J9_V10*MAPK_P)/(J9_KK10 + MAPK_P);

// Species initializations:
MKKK = 90;
MKKK_P = 10;
MKK = 280;
MKK_P = 10;
MKK_PP = 10;
```

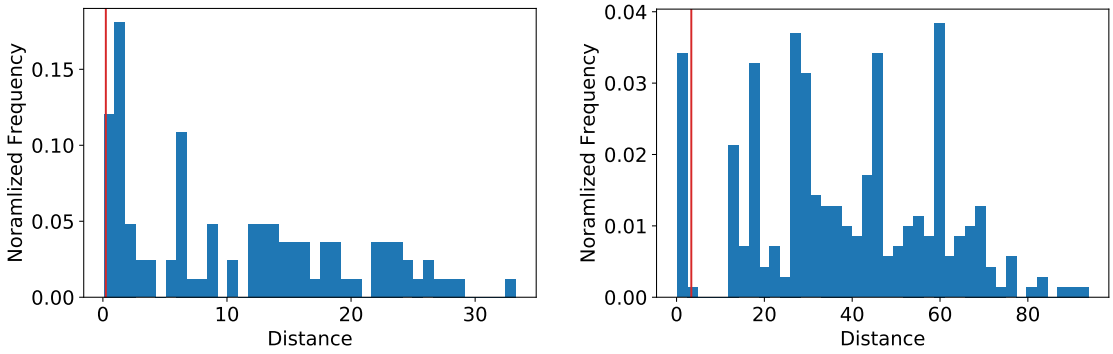
```
MAPK = 280;
MAPK_P = 10;
MAPK_PP = 10;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
J0_V1 = 2.5;
J0_Ki = 18;
J0_n = 2;
J0_K1 = 50;
J1_V2 = 0.25;
J1_KK2 = 40;
J2_k3 = 0.025;
J2_KK3 = 100;
J3_k4 = 0.025;
J3_KK4 = 100;
J4_V5 = 0.75;
J4_KK5 = 100;
J5_V6 = 0.75;
J5_KK6 = 100;
J6_k7 = 0.025;
J6_KK7 = 100;
J7_k8 = 0.025;
J7_KK8 = 100;
J8_V9 = 1.25;
```

```
  J8_KK9 = 100;
  J9_V10 = 1.25;
  J9_KK10 = 100;

  // Other declarations:
  const compartment_, J0_V1, J0_Ki, J0_n, J0_K1, J1_V2, J1_KK2, J2_k3,
     ↪ J2_KK3;
  const J3_k4, J3_KK4, J4_V5, J4_KK5, J5_V6, J5_KK6, J6_k7, J6_KK7, J7_k8;
  const J7_KK8, J8_V9, J8_KK9, J9_V10, J9_KK10;
end
'''

phrasedml_str1 = '''
  model1 = model "BorisEJB1"
  sim1 = simulate uniform(0, 9000, 9000)
  task1 = run sim1 on model1
  plot "Figure 1" time vs MAPK, MAPK_PP
'''
phrasedml_str2 = '''
  model1 = model "BorisEJB2"
  sim1 = simulate uniform(0, 12000, 12000)
  task1 = run sim1 on model1
  plot "Figure 1" time vs MAPK, MAPK_PP
'''

inline_omex1 = '\n'.join([antimony_str1, phrasedml_str1])
inline_omex2 = '\n'.join([antimony_str2, phrasedml_str2])
```

```
te.executeInlineOmex(inline_omex1)
te.executeInlineOmex(inline_omex2)
```

Appendix D

EVOMEG DISTANCE HISTOGRAMS



(A) (B)

(C) (D)

Figure D.1: Histograms of population fitness for evoMEG test cases with irreversible reactions. Red lines represent values which kernel density estimation used to filter the population to generate a model ensemble. (A) Feed-forward loop, (B) linear chain, (C) cycles, and (D) branched pathways.

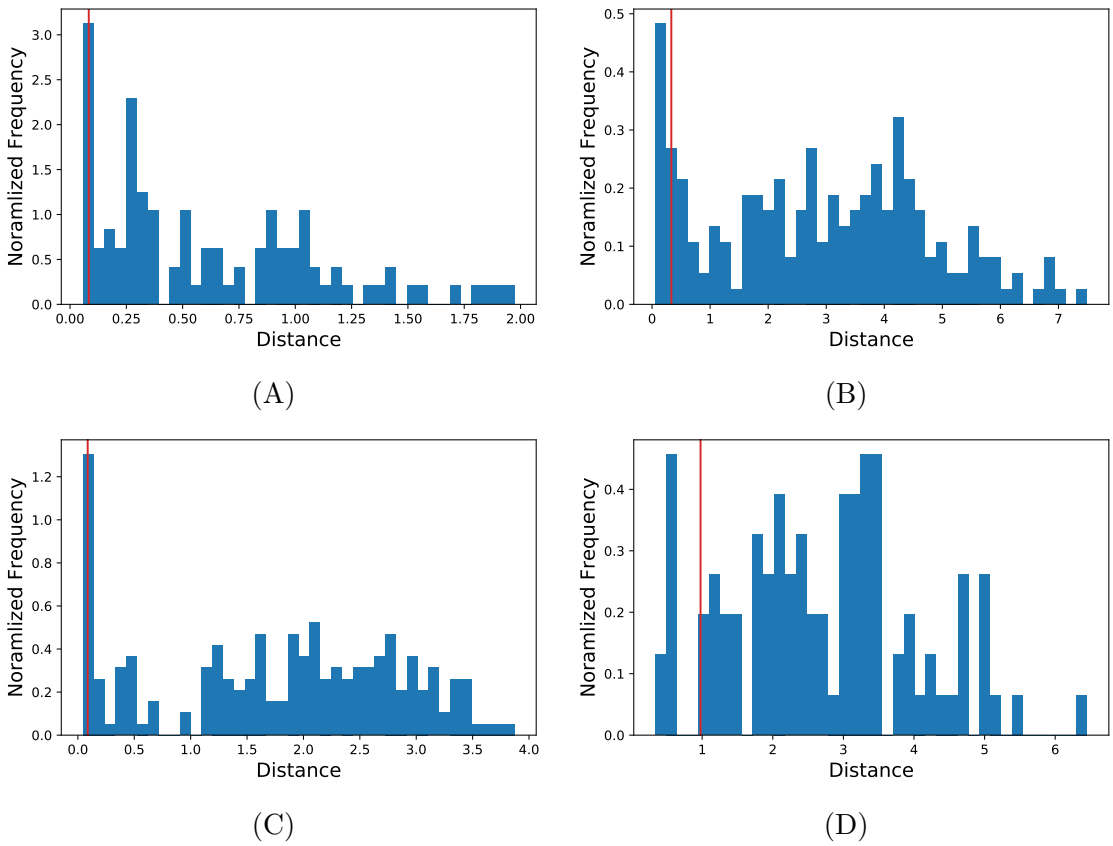
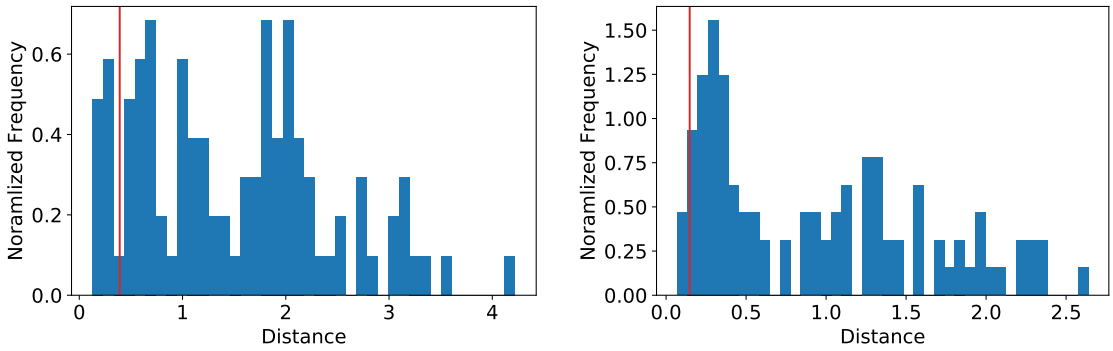


Figure D.2: Histograms of population fitness for evoMEG test cases with reversible reactions. Red lines represent values which kernel density estimation used to filter the population to generate a model ensemble. (A) Feed-forward loop, (B) linear chain, (C) cycles, and (D) branched pathways.



(A) (B)

Figure D.3: Histograms of population fitness for evoMEG test cases with reversible reactions and regulations. Red lines represent values which kernel density estimation used to filter the population to generate a model ensemble. (A) Linear chain with activation and (B) linear chain with inhibition.

Appendix E

EVOMEG CONVERGENCE CURVES

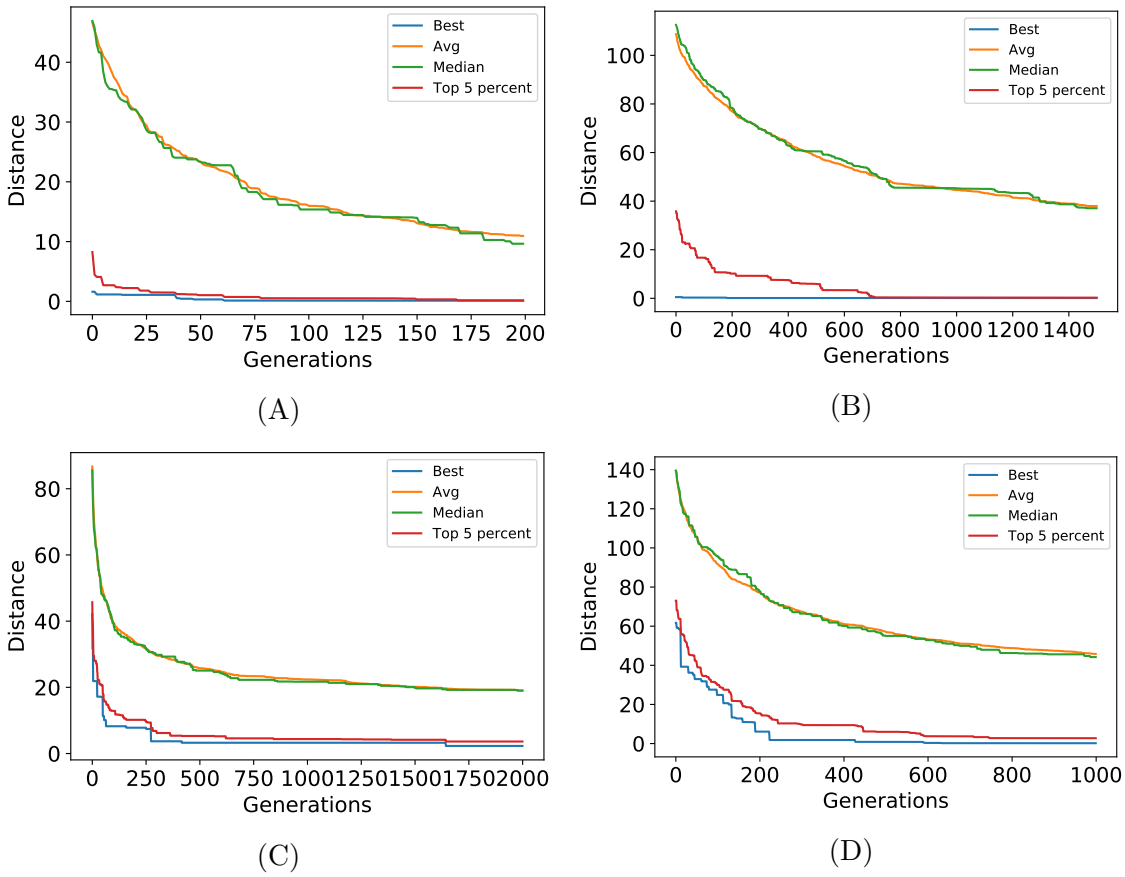


Figure E.1: Convergence curves for evoMEG runs using the test cases with irreversible reactions. (A) Feed-forward loop, (B) linear chain, (C) cycles, and (D) branched pathways.

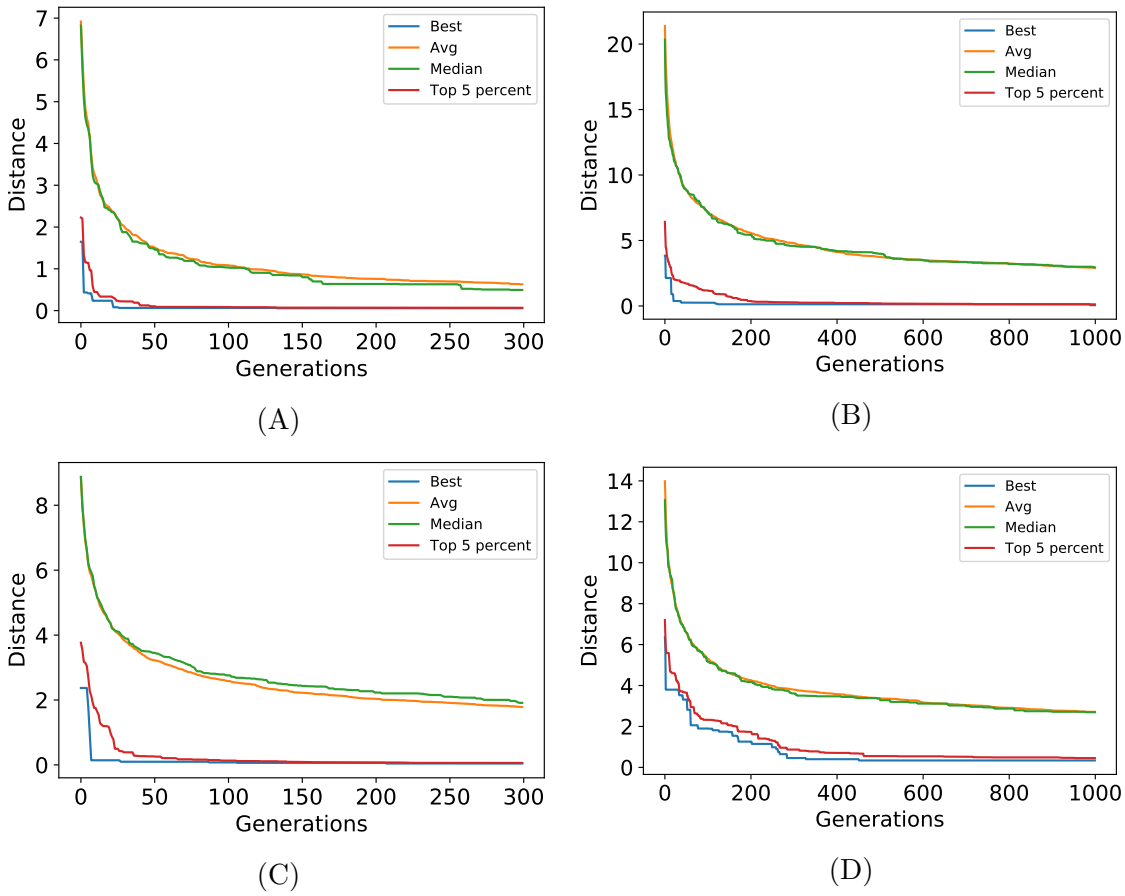


Figure E.2: Convergence curves for evoMEG runs using the test cases with reversible reactions. (A) Feed-forward loop, (B) linear chain, (C) cycles, and (D) branched pathways.

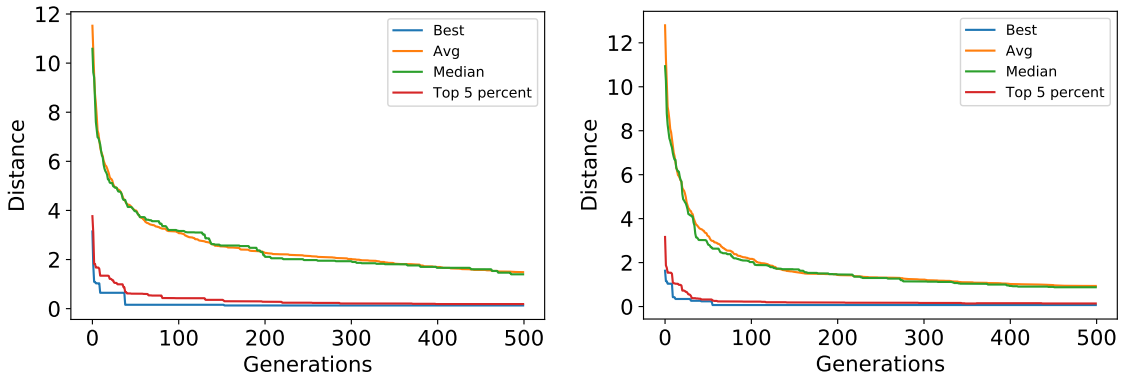


Figure E.3: Convergence curves for evoMEG runs using the test cases with reversible reactions and regulations. (A) Linear chain with activation and (B) linear chain with inhibition.

Appendix F

SCALED CONCENTRATION CONTROL COEFFICIENTS WITH NOISE

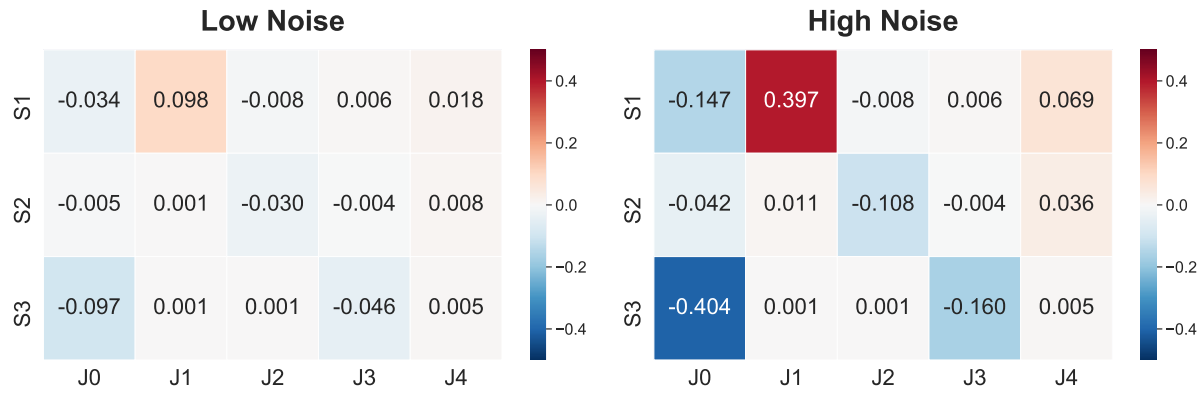


Figure F.1: Heatmaps of differences in scaled concentration control coefficients between low and high levels of noise and the original values. Some of values are identical because the original values are zero and only the measurement noises are applied while using the same seed.

Appendix G

SELECTED MODELS FROM ENSEMBLES UNDER LOW AND HIGH NOISE CONDITION

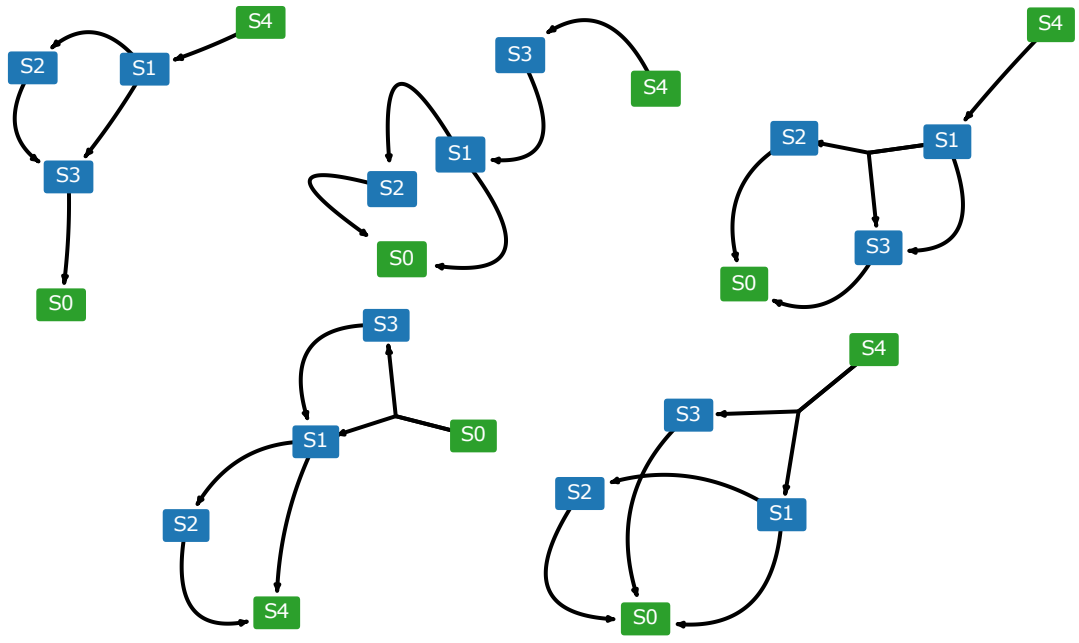


Figure G.1: Network diagrams of selected models in the ensemble under low noise condition. Nodes in green represent boundary species which are fixed.

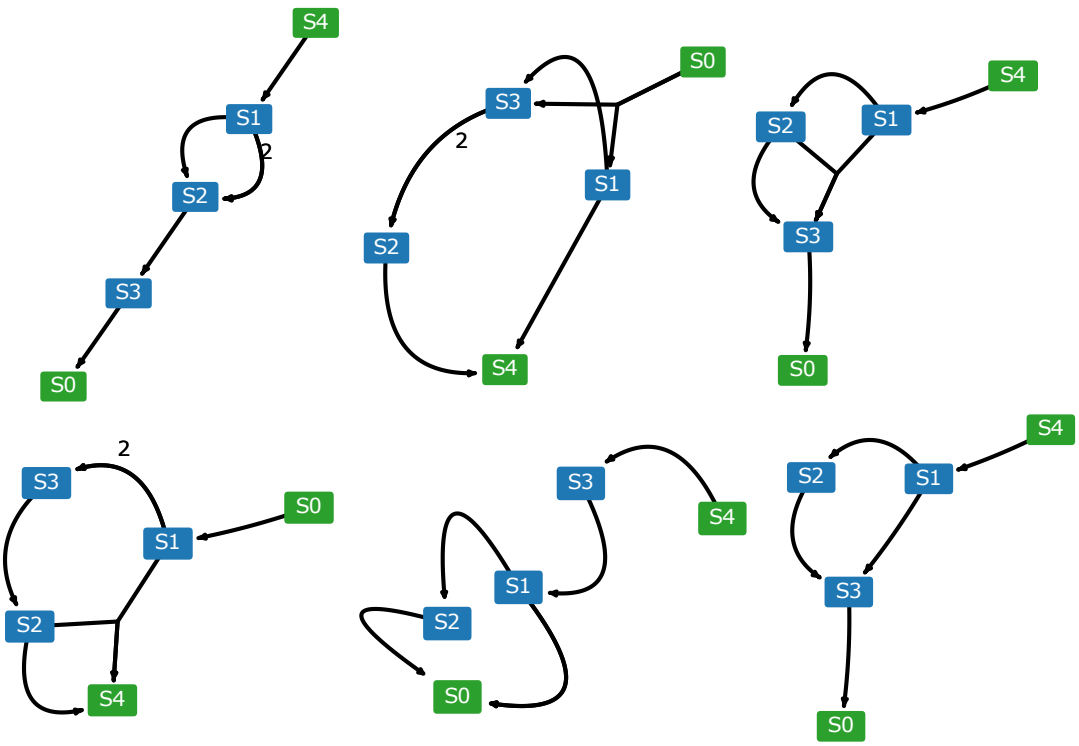


Figure G.2: Network diagrams of selected models in the ensemble under high noise condition. Nodes in green represent boundary species which are fixed.

Appendix H

VALIDATING THE ASSUMPTIONS OF ALLOSTERIC REGULATIONS

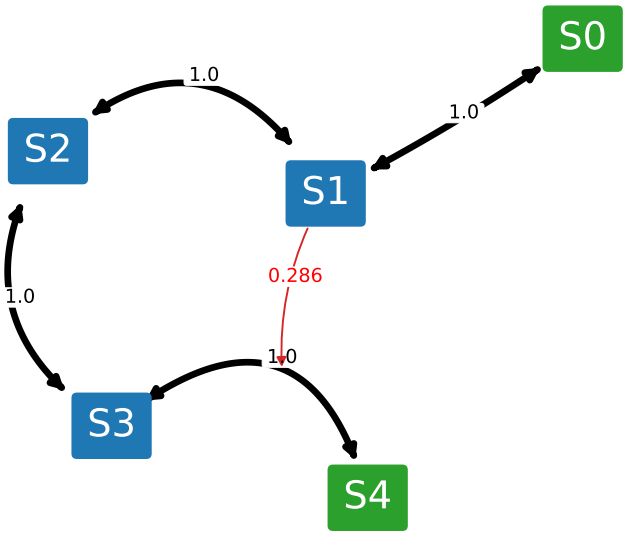


Figure H.1: Weighted network diagram from the model ensemble generated using linear chain with non-allosteric activation test case by applying the threshold of 0.25. Nodes in green represent boundary species which are fixed.

For the scope of the study, allosteric regulations are assumed for evoMEG algorithm when models with regulations are tested. To demonstrate that this assumption is reasonable when it comes to looking for topologies based on scaled concentration control coefficients, the algorithm is run on synthetic data generated from test cases with non-allosteric regulations while the algorithm is generating models with allosteric regulations.

Figure H.1 and Figure H.2 show the weighted network diagrams of model ensembles

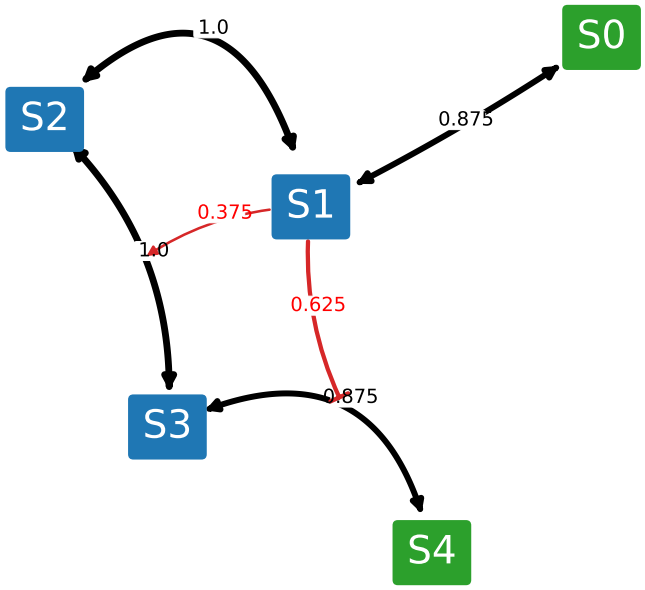
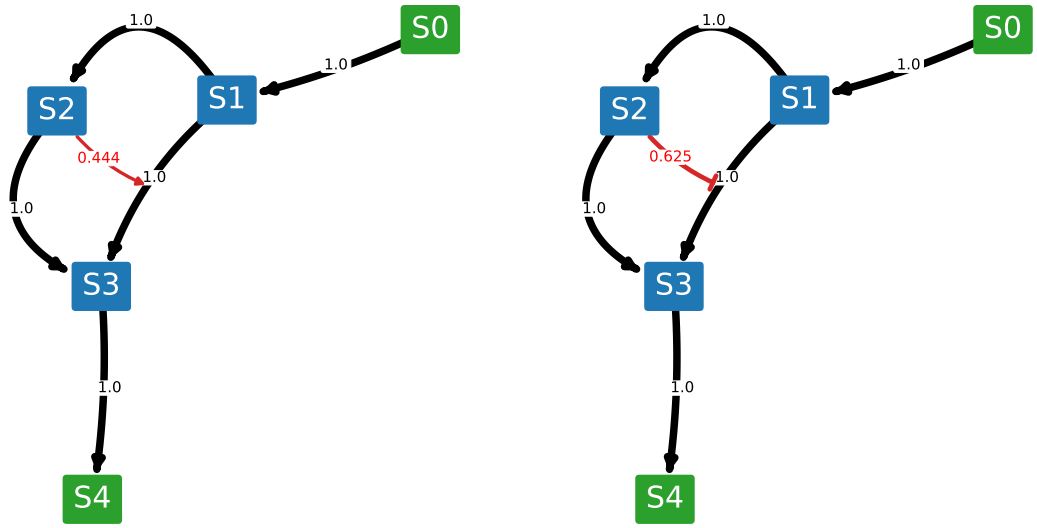


Figure H.2: Weighted network diagram from the model ensemble generated using linear chain with non-allosteric inhibition test case by applying the threshold of 0.25. Nodes in green represent boundary species which are fixed.

generated by evoMEG algorithm with a threshold of 0.25 while using models similar to those illustrated in Figure 4.21 but with non-allosteric regulations. While the pattern is weaker than the test cases with allosteric regulations, both the original activation and inhibition is observed after combining the model ensembles. Non-allosteric activation test case resulted in seven models and non-allosteric inhibition test case resulted in eight models.

Appendix I

WEIGHTED NETWORK DIAGRAMS FOR FEED-FORWARD LOOPS WITH ACTIVATION OR INHIBITION



(A) (B)

Figure I.1: Weighted network diagrams from ensemble generated by metaMEG for feed-forward loops with (A) activation or (B) inhibition with applying the threshold of 0.34.

Appendix J

FULL WEIGHTED NETWORK DIAGRAMS WITH REGULATIONS

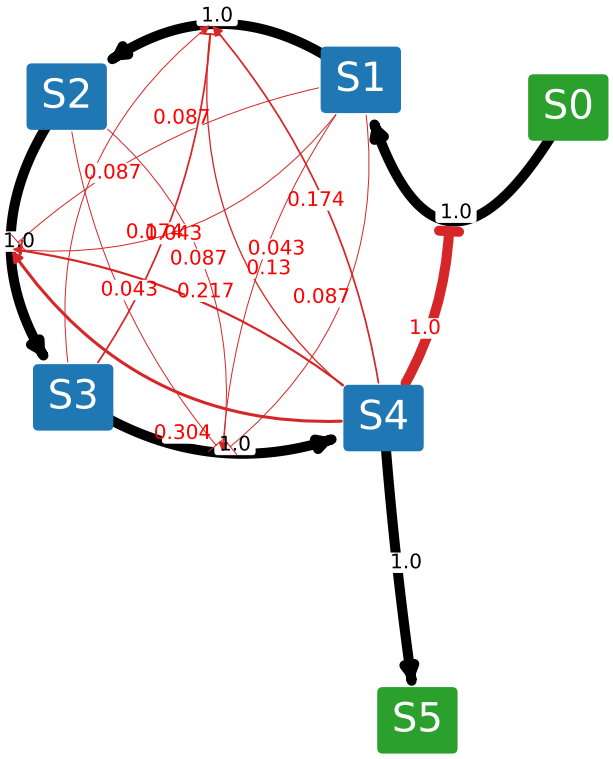


Figure J.1: Weighted network diagram from ensemble generated by metaMEG for the linear chain with a negative feedback without applying the threshold.

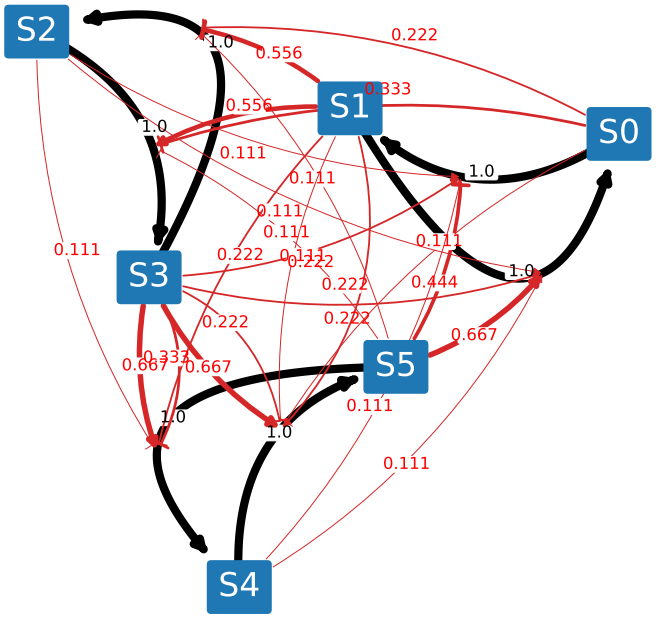


Figure J.2: Weighted network diagram from ensemble generated by metaMEG for the synthetic cascade without applying the threshold.

Appendix K

FULL WEIGHTED NETWORK DIAGRAMS WITHOUT REGULATIONS

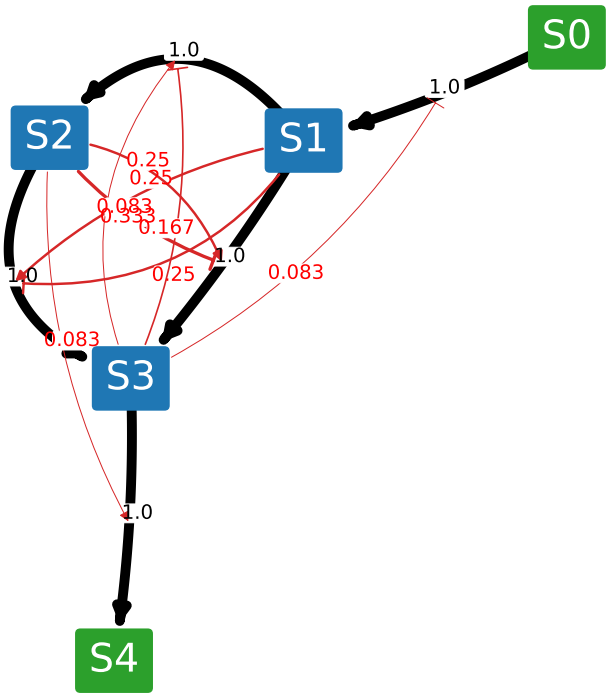


Figure K.1: Weighted network diagram from ensemble generated by metaMEG for the feed-forward loop without applying the threshold.

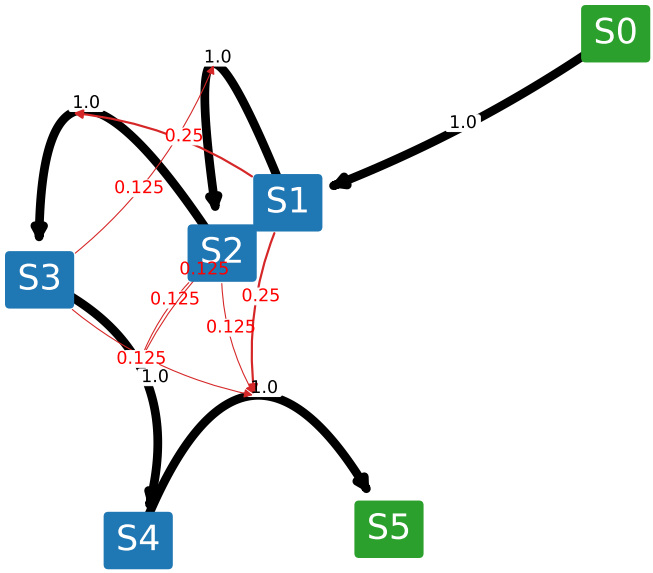


Figure K.2: Weighted network diagram from ensemble generated by metaMEG for the linear chain without applying the threshold.

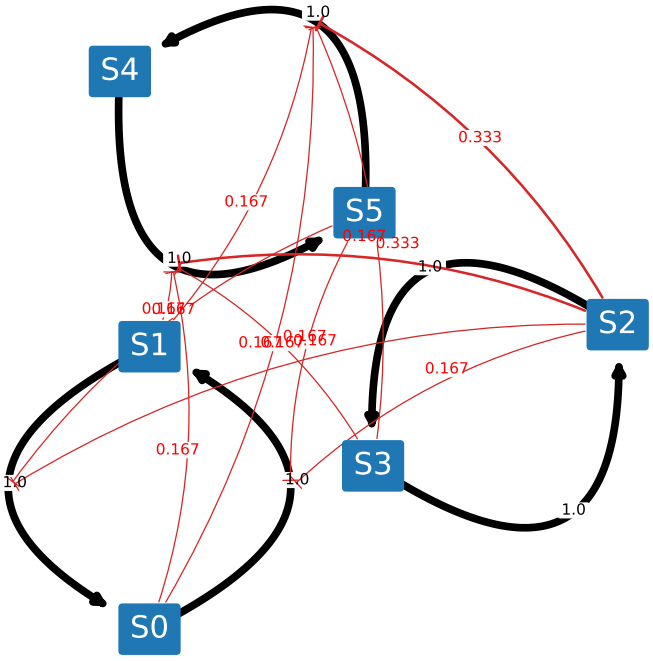


Figure K.3: Weighted network diagram from ensemble generated by metaMEG for the disconnected cycles without applying the threshold.

Appendix L

METAMEG DISTANCE HISTOGRAMS

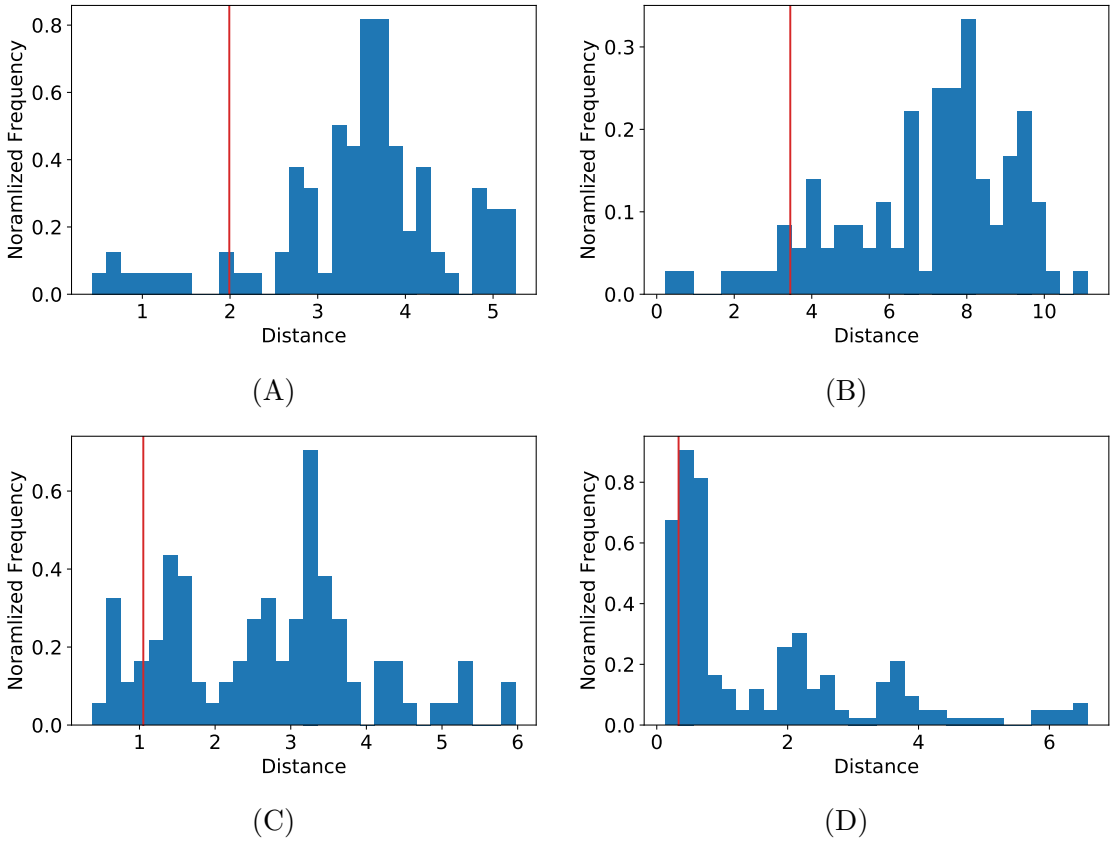


Figure L.1: Histograms of population fitness for metaMEG test cases with regulations. Red lines represent values which kernel density estimation used to filter the population to generate a model ensemble. (A) Feed-forward loop with activation, (B) feed-forward loop with inhibition, (C) linear chain with a negative feedback, and (D) synthetic cycle.

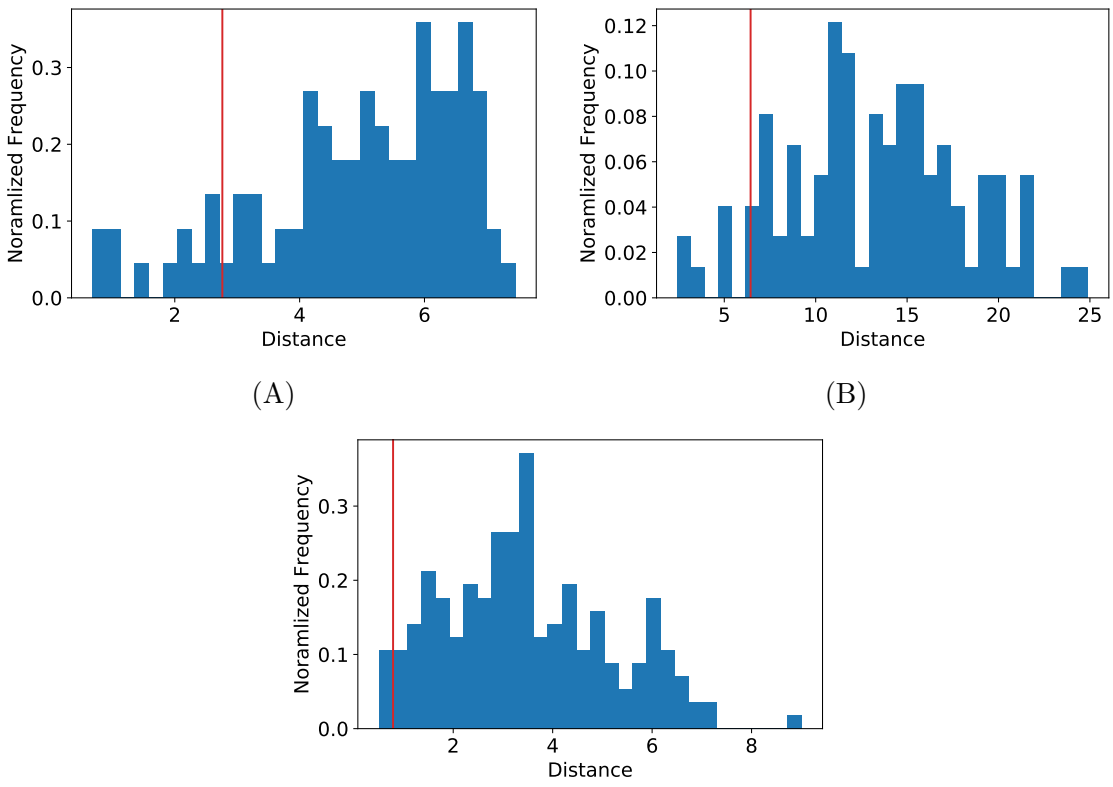


Figure L.2: Histograms of population fitness for metaMEG test cases without regulations. Red lines represent values which kernel density estimation used to filter the population to generate a model ensemble. (A) Feed-forward loop, (B) linear chain, and (C) disconnected cycles.

Appendix M

METAMEG CONVERGENCE CURVES

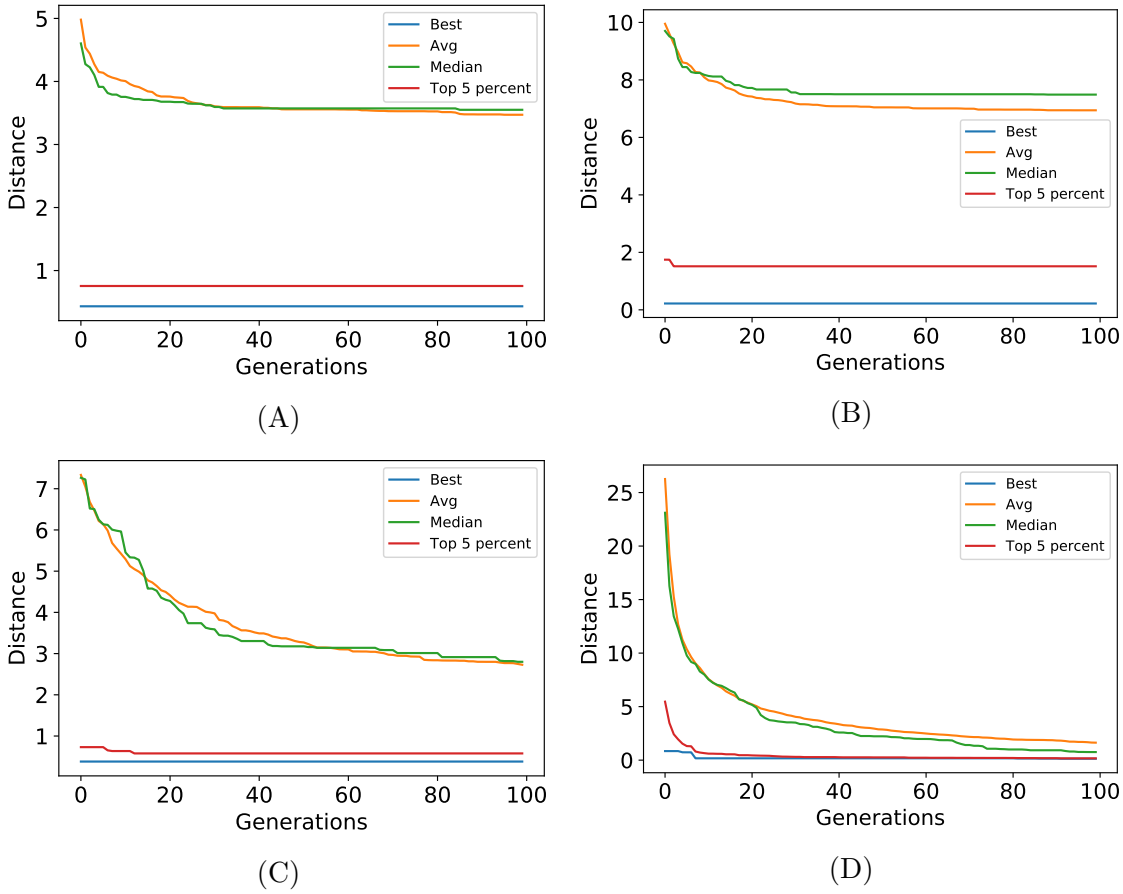


Figure M.1: Convergence curves for metaMEG runs using the test cases. (A) Feed-forward loop with activation, (B) feed-forward loop with inhibition, (C) linear chain with a negative feedback, and (D) synthetic cycle.

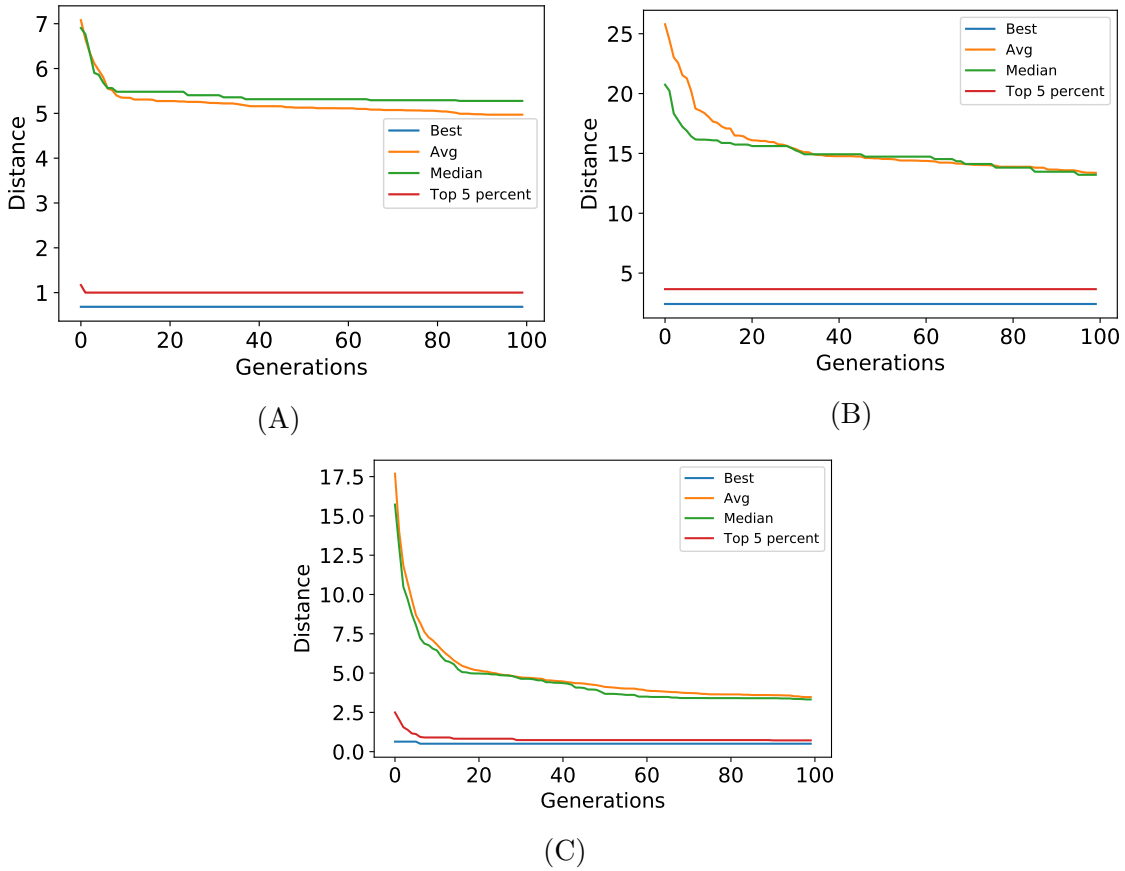


Figure M.2: Convergence curves for metaMEG runs using negative controls. (A) Feed-forward loop, (B) linear chain, and (C) disconnected cycles.

Appendix N

AVAILABILITY

All software, library, and codes presented in this paper are open-source and freely available over the web.

- evoMEG: The full source code is available at `https://github.com/kirichoi/CCR` under MIT license.
- metaMEG: The full source code is available at `https://github.com/kirichoi/metaMEG` under MIT license.
- netplotlib: The full source code, binary installers, and documentations are available at `https://github.com/kirichoi/netplotlib` under MIT license.
- Network Search Space Reduction: The full source code is available at `https://github.com/kirichoi/NSSR` under MIT license.
- phraSED-ML: The full source code, binaries, language specification, and documentations are available at `https://github.com/sys-bio/phrasedml` under BSD license.
- pySBOL: The full source code, binaries, language specification, and documentations are available at `http://sbolstandard.org` under Apache 2.0 license.
- Tellurium: The full source code, binary installers, and documentations are available at `http://tellurium.analogmachine.org` under Apache 2.0 license.