An introduction to whole-cell modeling

Release 0.0.1

Jonathan Karr
Arthur Goldberg

Jun 26, 2020

Contents

Despite extensive research into individual molecules and pathways, we still do not have a complete understanding of cell biology. To comprehensively understand cells, we must build whole-cell (WC) computational models that predict phenotype from genotype by representing all of the biochemical activity inside cells. In addition to helping researchers gain novel insights into biology, WC models could also help bioengineers design microorganisms and help physicians personalize medicine. Recently, we and others demonstrated that WC models are feasible by leveraging recent advances in computational and experimental technology to build the first model that represents every characterized gene in a cell. Although substantial work remains to develop comprehensive WC models, we believe that WC models can be achieved by systemizing cell modeling and coordinating the cell modeling community.

The goals of this primer are to teach researchers about the motivation, goals, principles, and methods of WC modeling and to prepare researchers to contribute to WC modeling. Toward this goal, the primer includes introductory readings and hands-on tutorials on WC modeling, the fundamental principles and methods of cell modeling, computer programming and software engineering, Linux computer systems, and scientific communication. Section 1 introduces the motivation, goals, and fundamental challenges of WC modeling; describes why WC modeling is becoming feasible; summarizes the principles and methods of WC modeling; reviews the latest WC models and their limitations; outlines the major bottlenecks to WC modeling; proposes a plan for achieving WC models as a community; and describes ongoing work to advance WC modeling. Section 2 introduces several foundational concepts and skills for WC modeling including computer programming, Linux computer systems, and scientific communication. Section 3 outlines the theoretical foundations for WC modeling. Section 4 describes the principles and methods of WC modeling. Section 5 outlines several funding opportunities for WC modeling.

Please note, this primer is under active development. Over time, we aim to develop a complete primer with detailed examples and tutorials. We welcome suggestions and feedback.

CHAPTER 1

Introduction

A central goal of biological science is to quantitatively understand how genotype influences phenotype. However, despite decades of research, a growing wealth of experimental data, and extensive knowledge of individual molecules and individual pathways, we still do not understand how biological behavior emerges from the molecular level. For example, we do not understand how transcription factors, non-coding RNA, localization signals, degradation tags, and other regulatory systems interact to control protein expression.

Consequently, physicians still cannot interpret the pathophysiological consequences of genetic variation and bioengineers still cannot rationally design microorganisms. Instead, patients often have to try multiple drugs to find a single effective drug, which exposes patients to unnecessary drugs, prolongs disease, and increases costs. Similarly, bioengineers often have to rely on time-consuming and expensive trial and error methods such as directed evolution [4][5].

Many engineering fields use mechanistic models to help understand and design complex systems such as cars [6], buildings [7], and transportation networks [8]. In particular, mechanistic models can help researchers conduct experiments with complete control, precision, and reproducibility.

To comprehensively understand cells, we must develop whole-cell (WC) computational models that predict cellular behavior by representing all of the biochemical activity inside cells [9][10][11][12]. WC models could accelerate biological science by helping researchers unify our knowledge of cell biology, identify gaps in our understanding, and conduct complex experiments that would be infeasible in vitro. WC models could also help bioengineers design microorganisms and help physicians personalize medicine.

Since the 1950's, researchers have been using modeling to understand cells. This has led to numerous models of individual pathways, including models of cell cycle regulation, chemical and electrical signaling, circadian rhythms, metabolism, and transcriptional regulation. Collectively, these efforts have used a wide range of mathematical formalisms including Boolean networks, flux balance analysis (FBA) [13][14][15], ordinary differential equations (ODEs), partial differential equations (PDEs), and stochastic simulation [16].

Over the last 20 years, researchers have begun to build more comprehensive models that represent multiple pathways [17][18][19][20][21][22][23]. Many of these models have built by combining multiple mathematically-dissimilar submodels of individual pathways into a single multi-algorithmic model [24][3].

Although we do not yet have all of the data and methods needed to model entire cells, we believe that WC models are rapidly becoming feasible due to ongoing advances in measurement and computational technology. In particular, we now have a wide array of experimental methods for characterizing cells, numerous repositories which contain much of the data needed for WC modeling, and a variety of tools for extrapolating experimental data to other organisms and

conditions. In addition, we now have a wide range of modeling and simulation tools, including tools for designing models, rule-based model formats for describing complex models, and tools for simulating multi-algorithmic models. However, few of these resources support the scale required for WC modeling, and many these resources remain siloed.

Nevertheless, we and others are beginning to model entire cells [17][25][26][27][28]. In 2012, we and others reported the first dynamical model that represents all of the characterized genes in a cell [27]. The model represents 28 pathways of the small bacterium *Mycoplasma genitalium* and predicts the essentiality of its genes with 80% accuracy.

However, several bottlenecks remain to build more comprehensive and more accurate WC models. In particular, we do not yet have all of the data needed for WC modeling or tools for designing, describing, or simulating WC models. To accelerate WC modeling, we must develop new methods for characterizing the single-cell dynamics of each metabolite and protein; develop new methods for scalably designing, simulating, and calibrating high-dimensional dynamical models; develop new standards for describing and verifying dynamical models; and assemble an interdisciplinary WC modeling community.

In this part, we summarize the scientific, engineering, and medical problems which are motivating WC modeling; propose the phenotypes that WC models should aim to predict and the molecular mechanisms that WC models should aim to represent; outline the fundamental challenges of WC modeling; describe why WC models are feasible by reviewing the existing methods, data, and models which could be leveraged for WC modeling; review the latest WC models and their limitations; outline the most immediate bottlenecks to WC modeling; propose a plan for achieving WC models; and summarize ongoing efforts to advance WC modeling.

1.1 Motivation for WC modeling

In our opinion, WC modeling is motivated by the needs to understand biology, personalize medicine, and design microorganisms. Biological science needs comprehensive models that represent the sequence, function, and interactions of each gene to help scientists holistically understanding cell biology. Similarly, precision medicine needs comprehensive models that predict phenotype from genotype to help physicians interpret the pathophysiological impact of genetic variation which can occur in any gene, and synthetic biology requires comprehensive models to help bioengineers rationally design microbial genomes for a wide range of applications.

In addition, WC models could help researchers address specific scientific problems such as determining how transcriptional regulation, non-coding RNA, and other pathways combine to regulate protein expression. Furthermore, each WC model could be used to address multiple questions, avoiding the need to build separate models for each question. However, few scientific problems require WC models, and we believe that most scientific problems would be more easily addressed with focused modeling.

Here, we describe the main applications which are motivating WC modeling. In the following sections, we define the biology that WC models must represent to support these applications and describe how to achieve such WC models.

1.1.1 Biological science: understand how genotype influences phenotype

Historically, the main motivation for WC modeling has been to help scientists understand how genotype and the environment determine phenotype, including how each individual gene, reaction, and pathway contributes to cellular behavior. For example, WC models could help researchers integrate heterogeneous experimental data about multiple genes and pathways. WC models could also help researchers gain novel insights into how pathways interact to control behavior. By comparison to experimental data, WC models could also help researchers identify gaps in our understanding. In addition, WC models would enable researchers to conduct experiments with complete control, infinite scope, and unlimited resolution, which would allow researchers to conduct complex experiments that would be infeasible in vitro.

1.1.2 Medicine: personalize medicine for individual genomes

Recent studies have shown that each patient has a unique genome, that genetic variation can occur in any gene and pathway, and that small genetic differences can cause patients to respond differentially to the same drugs. Together, this suggests that medicine could be improved by tailoring therapy to each patient's genome. Physicians are beginning to use data-driven models to tailor medicine to a small number of well-established genetic variants that have large phenotypic effects. Tailoring medicine for all genetic variation requires WC models that represent every gene and that can predict the phenotypic effect of any combination of genetic variation. Such WC models would help physicians predict the most likely prognosis for each patient and identify the best combination of drugs for each patient (Figure 1.1). For example, WC models could help oncologists conduct personalized in silico drug trials to identify the best chemotherapy regimen for each patient. Similarly, WC models could help obstetricians identify diseases in early fetuses. In addition, WC models could help pharmacologists avoid harmful gene-drug interactions.

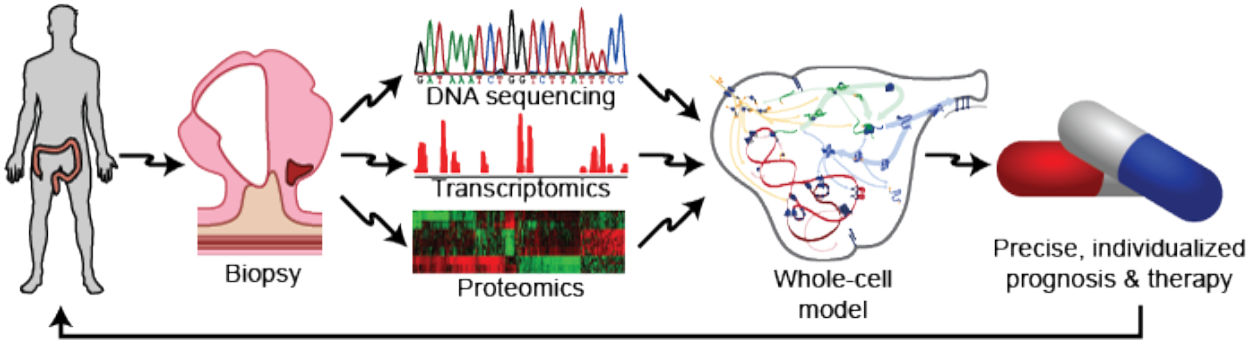


Figure 1.1: WC models could transform medicine by helping physicians use patient-specific models informed by genomic data to design personalized prognoses and therapies.

1.1.3 Synthetic biology: rationally design microbial genomes

Synthetic biology promises to create microorganisms for a wide range of industrial, medical, security applications such as cheaply producing chemicals, drugs, and fuels; quickly detecting diseased tissue; killing pathogenic bacteria; and decontaminating industrial waste. Currently, microorganisms are often engineered using directed evolution [4][5]. However, directed evolution is often time-consuming and limited to small phenotypic changes. Recently, researchers at the JCVI have begun to pioneer methods for chemically synthesizing entire genomes [29]. Realizing the full potential of this methodology requires WC models that can help bioengineers design entire genomes. For example, WC models could help bioengineers analyze the impact of synthetic circuits on host cells, design efficient chassis for synthetic circuits, and design bacterial drug delivery systems that can detect diseased tissue and synthesize drugs in situ.

1.2 The biology that WC models should aim to represent and predict

In the previous section, we argued that medicine and bioengineering need comprehensive models that can predict phenotype from genotype. Here, we outline the specific phenotypes that we believe that WC models should aim to predict and the specific physiochemical mechanisms that we believe that WC models should aim to represent to support medicine and bioengineering (Figure 1.2). In the following sections, we outline why we believe that WC models are becoming feasible and describe how to build and simulate WC models.

1.2.1 Phenotypes that WC models should aim to predict

To support medicine and bioengineering, we believe that WC models should aim to predict the phenotypes of individual cells over their entire life cycles (Figure 1.2b). Specifically, we believe that WC models should aim to predict the

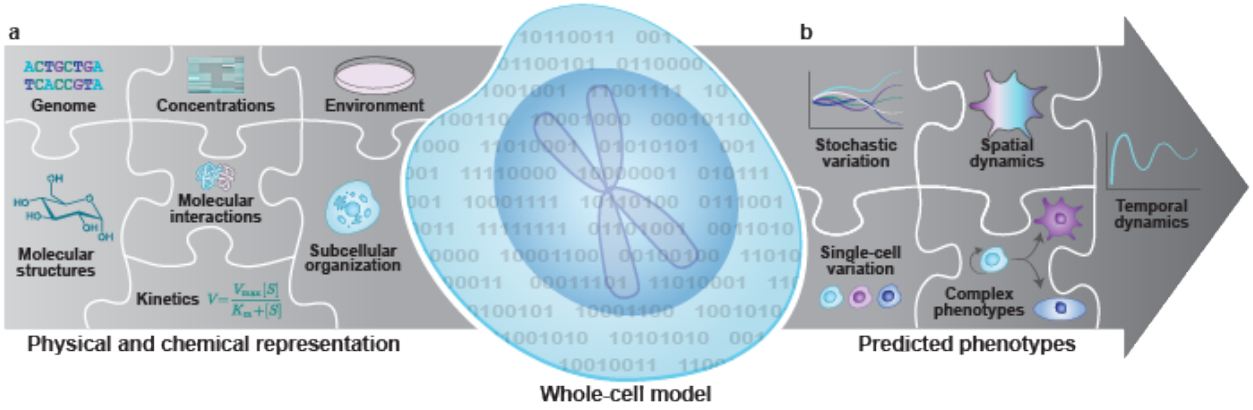


Figure 1.2: The physical and chemical mechanisms that WC models should aim to represent (**a**) and the phenotypes that WC models should aim to predict (**b**).

following five levels of phenotypes:

- **Stochastic dynamics:** To help physicians understand how genetic variation affects how cells respond to drugs, and to help bioengineers design microorganisms that are robust to stochastic variation, WC models should predict the stochastic behavior of each molecular species and molecular interaction. For example, this would help physicians design drugs that are robust to variation in RNA splicing, protein modification, and protein complexation. This would also help bioengineers design feedback loops that can control the expression of key RNA and proteins.
- **Temporal dynamics:** To help physicians understand the impact of genetic variation on cell cycle regulation, and to help bioengineers control the temporal dynamics of microorganisms, WC models should predict the temporal dynamics of the concentration of each molecular species. For example, this would help physicians identify genetic variation that can disrupt cell cycle regulation and cause cancer. This would also help bioengineers design microorganisms that can perform specific tasks at specific times.
- **Spatial dynamics:** To help physicians predict the intracellular distribution of drugs, and to help bioengineers use space to concentrate and insulate molecular interactions, WC models should predict the concentration of each molecular species in each spatial domain. For example, this would help physicians predict whether drugs interact with their intended targets and predict how quickly cells metabolize drugs. This would also help bioengineers maximize the metabolic activity of microorganisms by co-localizing enzymes and their substrates.
- **Single-cell variation:** To help physicians understand how drugs affect populations of heterogeneous cells, and to help bioengineers design robust microorganisms, WC models should predict the single-cell variation of cellular behavior. For example, this would help physicians understand how chemotherapies affect heterogeneous tumors, and help bioengineers design reliable biosensors that activate at the same threshold irrespective of stochastic variation in RNA and protein expression.
- **Complex phenotypes:** To help physicians understand the impact of variation on complex phenotypes and to help bioengineers design microorganisms that can perform complex phenotypes, WC models should predict complex phenotypes such as the cell shape, growth rate, and fate. For example, this would help physicians identify the primary variants responsible for disease and help physicians screen drugs in silico. This would also help bioengineers design sophisticated strains that can detect tumors, synthesize chemotherapeutics, and deliver drugs directly to tumors.

1.2.2 Physics and chemistry that WC models should aim to represent

To predict these phenotypes, we believe that WC models should aim to represent all of the chemical reactions inside cells and all of the physical processes that influence their rates (Figure 1.2a). Specifically, we propose that WC models

aim to represent the following seven aspects of cells:

- **Sequences:** To predict how genotype influence phenotype, including the contribution of each individual variant and gene, WC models should represent the sequence of each chromosome, RNA, and protein; the location of each feature of each chromosome such as genes, operons, promoters, and terminators; and the location of each site of each RNA and protein.
- **Structures:** To predict how molecular species interact and react, WC models should represent the structure of each molecule, including atom-level information about small molecules, the domains and sites of macromolecules, and the subunit composition of complexes. For example, this would enable WC models to predict the metabolism of novel compounds.
- **Subcellular organization:** To capture the molecular interactions that occur inside cells, WC models should represent the spatial organization of cells and the localization of each of metabolite, RNA, and protein species. For example, this would enable WC models to predict the spatial compartments in which molecular interactions occur.
- **Concentrations:** To capture the molecular interactions that can occur inside cells, WC models should also represent the concentration of each molecular species in each organelle and spatial domain.
- **Molecular interactions:** To capture how cells behave over time, WC models should represent the participants and effect of each molecular interaction, including the molecules that are consumed, produced, and transported, the molecular sites that are modified, and the bonds that are broken and formed. For example, this would enable WC models to capture the reactions responsible for cellular growth and homeostatic maintenance.
- **Kinetic parameters:** To predict the temporal dynamics of cell behavior, WC models should represent the kinetic parameters of each interaction such as the maximum rate of each reaction and the affinity of each enzyme for its substrates and inhibitors. For example, this would enable WC models to predict the impact of genetic variation on the function of each enzyme.
- **Extracellular environment:** To predict how the extracellular environment, including nutrients, hormones, and drugs, influences cell behavior, WC models should represent the concentration of each species in the extracellular environment. For example, this should enable WC models to predict the minimum media required for growth.

1.3 Fundamental challenges to WC modeling

In the previous section, we defined the biology that WC models should represent and predict. Building WC models that represent all of the biochemical activity inside cells and that can predict any cellular phenotype is challenging because this requires integrating molecular behavior to the cellular level across several spatial and temporal scales; assembling a complete molecular understanding of cell biology from incomplete, imprecise, and heterogeneous data; and simulating, calibrating, and validating computationally-expensive, high-dimensional models. Here, we describe these challenges to WC modeling. In the following sections, we describe emerging methods for overcoming these challenges to achieve WC models.

1.3.1 Integrating molecular behavior to the cell level over several spatiotemporal scales

The most fundamental challenge to WC modeling is integrating the behavior of individual species and reactions to the cellular level over several spatial and temporal scales. This is challenging because it requires accurate parameter values and scalable methods for simulating large models. Here, we summarize these challenges.

Sensitivity of phenotypic predictions to molecular parameter values

The first challenge to integrating molecular behavior to the cellular level is the sensitivity of model predictions to the values of critical parameters, which necessitates accurate parameter values. Accurately identifying these values is challenging because, as described below, it is challenging to optimize high-dimensional functions and because, as described in Section 1.3.1, our experimental data is incomplete and imprecise.

High computational cost of simulating large fine-grained models

A second challenge to integrating molecular behavior to the cellular level is the high computational cost of simulating entire cells with molecular granularity. For example, simulating one cell cycle of our first WC model of the smallest known freely living organism took a full core-day of an Intel E5520 CPU, or approximately 1×10^{15} floating-point operations [27]. Based on this data, the fact that human cells are approximately 10^6 larger, and the fact that a typical WC simulation experiment will require at least 1,000 simulation runs, a typical WC simulation experiment of a human cell will require approximately 10^6 core-years. To simulate larger and more complex organisms, we must develop faster parallel simulators.

1.3.2 Assembling a unified molecular understanding of cells from imperfect data

In our opinion, the greatest challenge to WC modeling is assembling a unified molecular understanding of cell biology. As illustrated in Figure 1.3, this requires assembling comprehensive data about every molecular species and molecular interaction. For example, to model *M. genitalium* we reconstructed (a) its subcellular organization; (b) its chromosome sequence; (c) the location, length, direction and essentiality of each gene; (d) the organization and promoter of each transcription unit; (e) the expression and degradation rate of each RNA transcript; (f) the specific folding and maturation pathway of each RNA and protein species including the localization, N-terminal cleavage, signal sequence, prosthetic groups, disulfide bonds and chaperone interactions of each protein species; (g) the subunit composition of each macromolecular complex; (h) its genetic code; (i) the binding sites and footprint of every DNA-binding protein; (j) the structure, charge and hydrophobicity of every metabolite; (k) the stoichiometry, catalysis, coenzymes, energetics and kinetics of every chemical reaction; (l) the regulatory role of each transcription factor; (m) its chemical composition and (n) the composition of its growth medium [30].

This is challenging because our data is incomplete, imprecise, heterogeneous, scattered, and poorly annotated. Here, we summarize these limitations and the challenges they present for WC modeling.

Incomplete data

The biggest limitation of our experimental data is that we do not have a complete experimental characterization of a cell. In particular, we have limited genome-scale data about individual metabolites and proteins, limited data about cell cycle dynamics, limited data about cell-to-cell variation, limited data about culture media, and limited data about cellular responses to genetic and environmental perturbations. Many genome-scale datasets are also incomplete. For example, most metabolomics and proteomics methods can only measure small numbers of metabolites and proteins.

Imprecise and noisy data

A second limitation of our experimental data is that many of our measurement methods are imprecise and noisy. For example, fluorescent microscopy cannot precisely quantitate single-cell protein abundances, single-cell RNA sequencing cannot reliably discern unexpressed RNA, and mass-spectrometry cannot reliably discern unexpressed proteins.

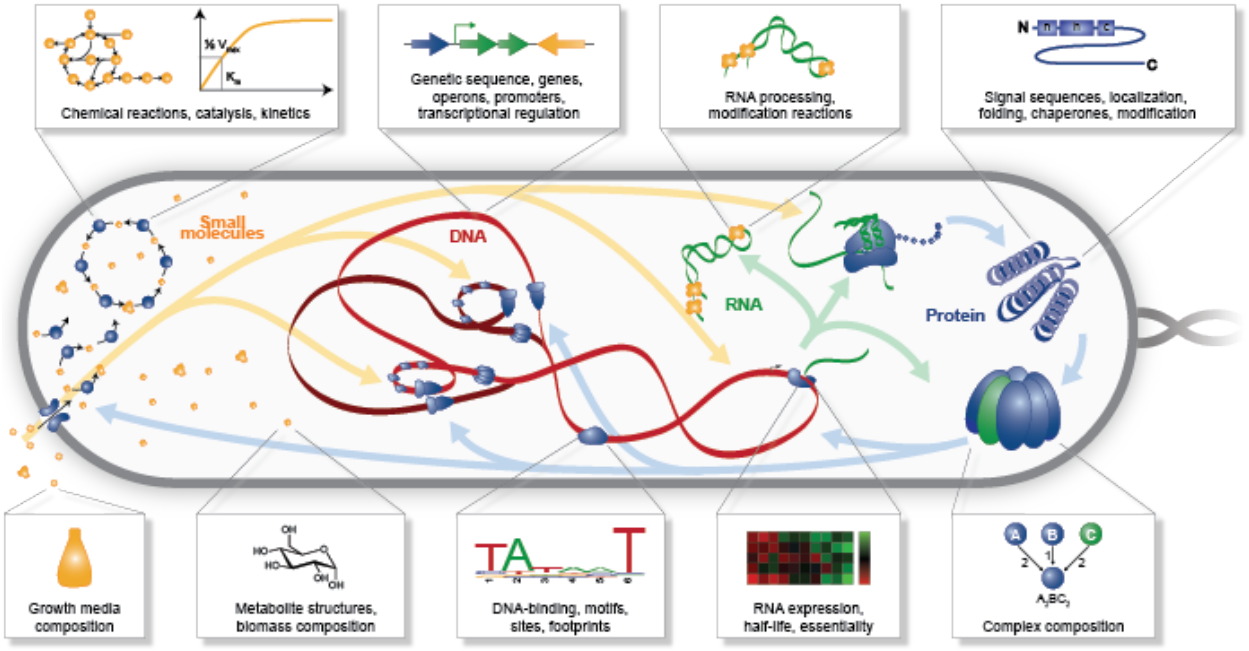


Figure 1.3: WC models require comprehensive data about every molecular species and molecular interaction.

Heterogeneous experimental methods

A third limitation of our experimental data is that our data is highly heterogeneous because we do not have a single experimental technology that is capable of completely characterizing a cell. Rather, we have a wide range of methods for characterizing different aspects of cells at different scales with different levels of resolution. For example, mass-spectrometry can quantitate the concentrations of tens of metabolites, deep sequencing can quantitate the concentrations of tens of thousands RNA, and each biochemical experiment can quantitate one or a few kinetic parameters.

Consequently, our experimental data also spans a wide range of scales and units. For example, we have extensive molecular information about the participants in each metabolic reaction and their stoichiometries, but we only have limited information about the substrates of each protein chaperone. As a second example, we have extensive single-cell information about RNA expression, but we have limited single-cell data about metabolite concentrations.

Heterogeneous organisms and environmental conditions

A fourth limitation of our data is that we only have a small amount of data about each organism and environmental condition, and only a small amount of data from each laboratory. However, collectively, we have a large amount of data.

Siloed data

Another limitation of our data is that no resource contains all of the data needed for WC modeling. Rather, our data is scattered across a wide range of databases, websites, textbooks, publications, supplementary materials, and other resources. For example, ArrayExpress [31] and the Gene Expression Omnibus [32] (GEO) only contain RNA abundance data, PaxDb only contains protein abundance data [33], and SABIO-RK only contains kinetic data [34]. Furthermore, many of these data sources use different identifiers and different units.

Insufficient annotation

Furthermore, much of our data is insufficiently annotated to understand its biological semantic meaning and provenance. For example, few RNA-seq datasets in ArrayExpress [31] have sufficient metadata to understand the environmental condition that was measured, including the concentration of each metabolite in the growth media and the temperature and pH of the growth media. Similarly, few kinetic measurements in SABIO-RK [34] have sufficient metadata to understand the strain that was measured.

1.3.3 Selecting, calibrating and validating high-dimensional models

A third fundamental challenge to WC modeling is the high-dimensionality of WC models which makes WC models susceptible to the "curse of dimensionality", the need for more data to constrain high-dimensional models [1]. In particular, the curse of dimensionality makes it challenging to select, calibrate, and validate WC models because we do not yet have sufficient data to data to select among multiple possible WC models, avoid overfitting WC models, precisely determine the value of each parameter, or test the accuracy of every possible prediction. Furthermore, it is computationally expensive to select, calibrate, and validate high-dimensional models.

1.4 Feasibility of WC models

Despite the numerous challenges to WC modeling described in the previous section, we believe that WC modeling is rapidly becoming feasible due to ongoing technological advances throughout computational systems biology, bioinformatics, genomics, molecular cell biology, applied mathematics, computer science, and software engineering including methods for experimentally characterizing cells, repositories for sharing data, tools for building and simulating dynamical models, models of individual pathways, and model repositories. While substantial work remains to adapt and integrate these technologies into a unified framework for WC modeling, these technologies are already forming a strong intellectual foundation for WC modeling. Here, we review the technologies that we believe are making WC modeling feasible, and describe their present limitations for WC modeling. In the following section, we describe how we are beginning to leveraging these technologies to build and simulate WC models.

1.4.1 Experimental methods, data, and repositories

Here, we review advances in measurement methods, data repositories, and bioinformatics tools that are generating the data needed for WC modeling, aggregating this data into repositories, and producing tools for extrapolating data to other genotypes and environments.

Measurement methods

Advances in biochemical, genomic, and single-cell measurement are rapidly generating the data needed for WC modeling [35][36][37] (Table 1.1). For example, Meth-Seq can assess epigenetic modifications [38], Hi-C can determine the average structure of chromosomes [39], ChIP-seq can determine protein-DNA interactions [40], fluorescence microscopy can determine protein localizations, mass-spectrometry can quantitate average metabolite concentrations, scRNA-seq [41][42] can quantitate the single-cell variation of each RNA [41], FISH [43] can quantitate the spatiotemporal dynamics and single-cell variation of the abundances of a few RNA, mass spectrometry can quantitate the average abundances of hundreds of proteins [44][45], mass cytometry can quantitate the single-cell variation of the abundances of tens of proteins [46], and fluorescence microscopy can quantitate the spatiotemporal dynamics and single-cell variation of the abundances of a few proteins. However, improved methods are still needed to measure the dynamics of the entire metabolome and proteome.

Table 1.1: Types of experimental data that can be used to build, calibrate, and validate WC models.

Data type	URL	Reference
Metabolites		
Structure		
Mass spectrometry	http://doi.org/10.1002/mas.20108	Dettmer et al., 2007
Concentration		
Fluorescence microscopy	http://doi.org/10.1126/science.1243259	Zenobi, 2013
Mass spectrometry	http://doi.org/10.1002/mas.20108	Dettmer et al., 2007
Spectrophotometry	http://doi.org/10.1016/B978-0-12-416618-9.00005-4	TeSlaa and Teitell, 2014
DNA		
Structure		
DNA sequencing	http://doi.org/10.1038/nbt1486	Shendure and Ji, 2008
Methylation sequencing	http://doi.org/10.1038/nrg2732	Laird, 2010
Chromosome conformation capture	http://doi.org/10.1038/nrg3454	Dekker et al., 2013
Concentration		
Flow cytometry	http://doi.org/10.1016/j.it.2012.02.010	Bendall et al., 2012
RNA		
Structure		
RNA sequencing	http://doi.org/10.1038/nrg2934	Ozsolak and Milos, 201
Modification sequencing (ICE, MERIP-Seq)	http://doi.org/10.1016/j.trsl.2014.04.003	Liu and Pan, 2015
X-ray crystallography	http://doi.org/10.1016/S0076-6879(09)69006-6	Reyes et al., 2009
Localization		
Fluorescence in situ hybridization	http://doi.org/10.1126/science.1250212	Lee et al., 2014
Transcription rate		
ChIP-seq	http://doi.org/10.1038/nrg2641	Park, 2009
GRO-seq	http://doi.org/10.1126/science.1162228	Core et al., 2008
Half-life		
Microarray timecourse	http://doi.org/10.1101/gr.912603	Selinger et al., 2003
RNA sequencing timecourse	http://doi.org/10.1038/nature10098	Schwanhäusser et al., 20
Concentration		
Microarray	http://doi.org/10.1038/35087138	Schulze and Downward,
RNA sequencing	http://doi.org/10.1038/nrg2934	Ozsolak and Milos, 201
Fluorescence in situ hybridization	http://doi.org/10.1126/science.1188308	Taniguchi et al., 2010
Proteins		
Structure		
Mass spectrometry	http://doi.org/10.1126/science.1124619	Domon and Aebersold,
Nuclear magnetic resonance spectroscopy	http://doi.org/10.1146/annurev.biochem.73.011303.074004	Tugarinov et al., 2004
RNA sequencing	http://doi.org/10.1038/nrg2934	Ozsolak and Milos, 201
X-ray crystallography	http://doi.org/10.1007/978-1-60327-159-2_3	Ilari and Savino, 2008
Localization		
Fluorescence microscopy	http://doi.org/10.1126/science.1124618	Giepmans et al., 2006
Translation rate		
Ribosomal profiling	http://doi.org/10.1038/nrg3645	Ignolia, 2014
Half-life		
Fluorescence timecourse	http://doi.org/10.1098/rsob.140002	Knop and Edgar, 2014
Mass spectrometry timecourse	http://doi.org/10.1038/nature10098	Schwanhäusser et al., 20
Concentration		
Flow cytometry	http://doi.org/10.1016/j.it.2012.02.010	Bendall et al., 2012
Fluorescence microscopy	http://doi.org/10.1126/science.1124618	Giepmans et al., 2006
Mass cytometry	http://doi.org/10.1016/j.it.2012.02.010	Bendall et al., 2012

Continued on

Table 1.1 – continued from previous page

Data type	URL	Reference
Mass spectrometry	http://doi.org/10.1126/science.1124619	Domon and Aebersold,
Spectrophotometery	http://doi.org/10.1016/S0076-6879(09)63008-1	Noble and Bailey, 2009
Interactions		
RNA-DNA		
CHIRP-Seq	http://doi.org/10.1016/j.molcel.2011.08.027	Chu et al., 2011
Protein-metabolite		
Mass spectrometry	http://doi.org/10.1126/science.1124619	Domon and Aebersold,
Protein-DNA		
ChIP-seq	http://doi.org/10.1038/nrg2641	Park, 2009
DNase-seq	http://doi.org/10.1101/pdb.prot5384	Song and Crawford, 201
Protein-RNA		
CLIP-seq	http://doi.org/10.1002/wrna.31	Darnell, 2010
RIP-seq	http://doi.org/10.1016/j.molcel.2010.12.011	Zhao et al., 2010
Protein-protein		
Co-immunoprecipitation	http://doi.org/10.1101/pdb.prot3898	Sambrook and Russell,
Tandem affinity purification	http://doi.org/10.1016/j.pep.2010.04.009	Xu et al., 2010
Two-hybrid screen	http://doi.org/10.3390/ijms10062763	Brückner et al., 2009
Reaction fluxes		
Isotopic labeling	http://doi.org/10.1002/wsbm.1167	Klein and Heinzle, 2012
Phenotypic data		
Cell size		
Fluorescence microscopy	http://doi.org/10.1146/annurev.cellbio.042308.113408	Muzzey and van Oudena
Growth rates		
Spectrophotometery	http://doi.org/10.1177/2211068214555414	Jensen et al., 2015
Division times		
Fluorescence microscopy	http://doi.org/10.1002/cyto.a.20812	Wang et al., 2010
Motility, chemotaxis		
Fluorescence microscopy	http://doi.org/10.1038/sj.emboj.7601227	Dormann and Weijer, 20

Data repositories

Researchers are rapidly aggregating the experimental data needed for WC modeling into repositories (Table 1.2). This includes specialized repositories for individual types of data such as ECMDB [47] and YMDB [48] for metabolite concentrations; ArrayExpress [31] and the Gene Expression Omnibus [32] (GEO) for RNA abundances; PaxDb [33] for protein abundances; BiGG [49] for metabolic reactions, and SABIO-RK for kinetic parameters [34], as well as general purpose repositories such as FigShare [50], SimTk [51], and Zenodo [52].

Some researchers are making the data in these repositories more accessible by providing common interfaces to multiple repositories such as BioMart [53], BioServices [54], and Intermine [55].

Other researchers are making the data in these repositories more accessible by integrating the data into meta-databases. For example, KEGG contains a variety of information about metabolites, proteins, reactions, and pathways [56]; Pathway Commons contains extensive information about protein-protein interactions and pathways [57]; and UniProt contains a multitude of information about proteins [58].

In addition, some researchers are integrating information about individual organisms into PGDBs such as the BioCyc family of databases [59][60]. These databases contain a wide range of information including the stoichiometries of individual reactions, the compositions of individual protein complexes, and the genes regulated by individual transcription factors. Because PGDBs already contain integrated data about a single organism, PGDBs could readily be leveraged to build WC models. In fact, Latendresse developed MetaFlux to build constraint-based models of metabolism from EcoCyc [61].

Furthermore, meta-databases such as *Nucleic Acid Research*'s Database Summary [62] and re3data.org [63] contain lists of data repositories.

Most of these repositories have been developed by encouraging individual researchers to deposit their data or by employing curators to manually extract data from publications, supplementary files, and websites. In addition, researchers are beginning to use natural language processing to develop tools for automatically extracting data from publications [64].

Table 1.2: Repositorie
to build, calibrate, and

Database	Content
Species structures	
Metabolites	
ChEBI	Compound structures
KEGG Compound	Compound structures
KEGG Glycan	Glycan structures
Metabolomics Workbench Metabolite Database	Compound structures
LIPID MAPS	Lipid structures
PubChem	Compound structures
DNA	
ArrayExpress	Functional genomics data including Hi-C data
GenBank	DNA sequences
GEO	Functional genomics data including Hi-C data
MethDB	Methylation sequencing data
RNA	
ArrayExpress	Functional genomics data including RNA-seq data that encompasses initiation and
GEO	Functional genomics data including RNA-seq data that encompasses initiation and
MODOMICS	Post-transcriptional modifications
RNA Modification Database	Post-transcriptional modifications
Protein	
3d-footprint	3-dimensional footprints
dbPTM	Post-translational modifications
PDB	3-dimensional structures
RESID	Post-translational modifications
UniMod	Post-translational modifications
UniProt	Functional protein annotations including post-translational modifications
Localization and signal sequences	
RNA	
Fly-FISH	RNA localizations
RNALocate	RNA localizations
Protein	
COMPARTMENTS	Protein localizations for Arabidopsis thaliana, Caenorhabditis elegans, Drosophila
Human Protein Reference Database	Protein localizations for Homo sapiens
LOCATE	Protein localizations for Homo sapiens and Mus musculus
LocDB	Protein localizations for Arabidopsis thaliana and Homo sapiens
LocSigDB	Protein localizations for eukaryotes
OrganelleDB	Protein localizations
PSORTdb	Protein localizations for bacteria and archaea
UniProt	Functional protein annotations including protein localizations
Concentrations	
Metabolites	
BioNumbers	Quantitative measurements of physical, chemical, and biological properties includi

Table 1.

Database	Content
ECMBD	Metabolite concentrations in Escherichia coli
HMDB	Metabolite concentrations in Homo sapiens
MetaboLights	
YMDB	Metabolite concentrations in Saccharomyces cerevisiae
RNA	
ArrayExpress	Functional genomics data including RNA abundances from microarray and RNA-se
Expression Atlas	RNA abundances across organisms and environmental conditions
GEO	Functional genomics data including RNA abundances from microarray and RNA-se
Proteins	
Review	
Human Protein Atlas	Protein abundances for Homo sapiens
PaxDb	Protein abundances
Plasma Proteome Database	Protein abundances for Homo sapiens plasma
PRIDE	Mass-spectrometry proteomics data
Interactions	
Protein-Metabolite, See also: Cofactors	
Review	
DrugBank	Drugs and their targets
STITCH	Drugs and their targets
SuperTarget	Drugs and their targets
Therapeutic Targets Database	Drugs and their targets
Protein-DNA	
ArrayExpress	Functional genomics data including ChIP-seq data of protein-DNA interations
GEO	Functional genomics data including ChIP-seq data of protein-DNA interations
DBD	Predicted transcription factors
DBTBS	Bacillus subtilis transcription factors and the operons they regulate
ORegAnno	Transcription factor binding sites
TRANSFAC	Transcription factor binding motifs
UniProbe	Transcription factor binding motifs
Protein-Protein	
Review	
ConsensusPathDB	Homo sapiens molecular interactions including protein-protein interactions
BioGRID	Protein-protein interactions
CORUM	Protein complex composition
DIP	Protein-protein interactions
IntAct	Molecular interactions including protein-protein interactions
STRING	Protein-protein interactions
UniProt	Function protein annotations including protein complex compositions
Reactions	
Stoichiometries, catalysis	
BioCyc	Reaction stoichiometries and catalysts
KEGG	Reaction stoichiometries and catalysts
MACiE	Detailed reaction mechanisms
Rhea	Reaction stoichiometries
UniProt	Reaction stoichiometries and catalysts
Cofactors	
CoFactor	Organic enzyme cofactors
PDB	3-dimensional protein structures including cofactors
UniProt	Functional protein annotations including cofactors

Table 1.

Database	Content
Rate laws and rate constants	
BioNumbers	Quantitative measurements of physical, chemical, and biological properties includi
BRENDA	Kinetic parameters and rate laws
SABIO-RK	Kinetic parameters and rate laws
Pathways	
Metabolic	
Review	
BioCyc	Species-specific pathways
KEGG PATHWAY	Species-specific pathways
Signaling	
Review	
hiPathDB	Metadatabase of Homo sapiens signaling pathways
KEGG PATHWAY	Pathways including signaling pathways
NetPath	Immune signaling pathways
PANTHER Pathway	Pathways including signaling pathways
Pathway Commons	Metadatabase of signaling pathways
Reactome	Pathways including signaling pathways
WikiPathways	Community curated pathways including signaling pathways
Meta-databases and meta-database tools	
Review	
BioCatalogue	List of web services
BioMart	Tools for integrating data from multiple repositories
BioMoby	Ontology-based messaging system for discovering data
BIOSERVICES	Python APIs to several popular repositories
BioSWR	List of web services
ELIXIR	Effort to develop a common data infrastructure for Europe
NAR Database Summary	List of database papers published in Nucleic Acids Research database issues
re3data.org Registry	List of data repositories

Prediction tools

Accurate prediction tools can be a useful alternative to constraining models with direct experimental evidence. Currently, many tools can predict molecular properties such as the organization of genes into operons, RNA folds, and protein localizations (Table 1.3). For example, PSORTb can predict the localization of bacterial proteins [65] and TargetScan can predict the mRNA targets of small non-coding RNAs [66]. In particular, these tools can be used to impute missing data and extrapolate observations to other organisms, genetic conditions, and environmental conditions. However, many current prediction tools are not sufficiently accurate for WC modeling.

Tool	Prediction(s)
Metabolites	
Physical properties	
Review	Survey of several chemoinformatic packages
Chemistry Development Kit (CDK)	Java libraries for processing chemical information
Cinfony	A common API to several cheminformatics toolkits
Indigo	A toolkit for molecular fingerprinting, substructure searching, and visualization
JChem	Tools for draw and visualizing molecules and searching chemical databases

Tool	Prediction(s)
Open Babel	Tools for searching, converting, analyzing, and storing chemical structures
RDKit	Cheminformatics toolkit
Thermodynamics	
UManSysProp	Estimates the standard Gibbs free energy of formation of organic molecules using the Joback g
Web GCM	Estimates the standard Gibbs free energy of formation of organic molecules using the Mavrovo
DNA	
Promoters	
Review	Review of promoter prediction methods for Homo sapiens
PePPER	Predicts prokaryote promoters
Promoter	Predicts vertebrate PolII promoters
PromoterHunter	Predicts prokaryote promoters
Genes	
Review	Review of several gene prediction software tools
GeneMark	Family of tools for predicting viral, prokaryotic, archaeal, and eukaryotic genes
GENESCAN	Predicts plant and vertebrate genes
GLIMMER	Predicts viral, prokaryotic, and archaeal genes
Operons	
Review	Survey of several operon prediction methods
DOOR	Predicts prokaryotic operons
OperonDB	Estimates the likelihood that pairs of genes are in the same operon
ProOpDB	Predicts prokaryotic operons
VIMSS	Predicts prokaryotic and archaeal operons
Variant interpretation	
PolyPhen-2	Predicts the functional effects of amino acid substitutions
PROVEAN	Predicts the functional effects of amino acid substitutions and indels
SIFT	Predicts the functional effects of amino acid indels
RNA	
Splice sites	
Review	Review of methods for predicting splice sites
GeneSplicer	Predicts eukaryotic splice sites
Human Splicing Finder	Identify and predict mutations' effect on human splicing motifs
NetGene2	Predicts splice sites in Arabidopsis thaliana, Caenorhabditis elegans, and Homo sapiens
NNSplice	Predicts splice sites Drosophila melanogaster and Homo sapiens
Secondary structure	
Review	Review of methods for predicting RNA secondary structures
Mfold	Predicts RNA secondary structures
RNAstructure	Predicts RNA and DNA secondary structures
ViennaRNA	Predicts RNA secondary structures
Open reading frame	
ORF Finder	Predicts open reading frames
ORF Investigator	Predicts open reading frames
ORFPredictor	Predicts open reading frames from EST and cDNA sequences
Terminators	
Review	Review of prokaryotic transcription termination that cites several methods for predicting termin
ARNold	Predicts prokaryotic rho-independent terminators
FindTerm	Predicts prokaryotic rho-independent terminators
GeSTer	Predicts prokaryotic rho-independent terminators
TransTermHP	Predicts prokaryotic rho-independent terminators
Proteins	

Tool	Prediction(s)
Localization	
Review	Review of methods for predicting the subcellular localization of prokaryotic and eukaryotic pro
Review	Review of methods for predicting the subcellular localization of prokaryotic proteins
Cell-PLoc	Predicts the subcellular localization of proteins for multiple species
MultiLoc	Predicts the subcellular localization of proteins for multiple species
PSORTb	Predicts the subcellular localization of prokaryotic and archaeal proteins
SecretomeP	Predicts signal peptide-independent protein secretion
WoLF PSORT	Predicts the subcellular localization of eukaryotic proteins
Signal sequence	
Review	Architecture, function and prediction of long signal peptides
Phobius	Predict protein transmembrane topology and signal peptides from AA sequences
PRED-LIPO	Predict lipoprotein and secretory signal peptides in gram-positive bacteria
PRED-SIGNAL	Predict signal peptides in archaea
SignalP	Predict signal peptide cleavage sites in prokaryotic and eukaryotic proteins
Disulfide bonds	
Review	Review of methods predicting disulfide bonds
Review	Review of methods predicting disulfide bonds
Cyscon	A consensus model for predicting disulfide bonds
DIANNA	Predicts disulfide bonds
Dinsolve	Predicts disulfide bonds
DIPro	Predicts disulfide bonds
DISULFIND	Predicts disulfide bonds
Complex abundance	
SiComPre	Predicts the abundances of Homo sapiens and Saccharomyces cerevisiae protein complexes
Half-lives	
N-End rule	Predicts the half-lives of Escherichia coli, Saccharomyces cerevisiae and mammalian (rabit) pr
Interactions	
miRNA targets	
Review	Review of methods for predicting miRNA targets
Review	Review of methods for predicting miRNA targets
DIANA-microT-CDS	Predicts miRNA targets in Caenorhabditis elegans, Drosophila melanogaster, Homo sapiens, ar
miRSearch	Predicts miRNA targets in Homo sapiens, Mus musculus, and Rattus norvegicus
MirTarget	Predicts miRNAs targets in several animals
PITA	Predicts miRNA targets in Caenorhabditis elegans, Drosophila melanogaster, Homo sapiens, ar
STarMir	Predicts miRNA targets in Caenorhabditis elegans, Homo sapiens, and Mus musculus
TargetScan	Predicts miRNA targets in several animals
Protein-DNA binding sites	
Review	Review of tools for predicting transcription factor binding sites
Review	Review of tools for predicting transcription factor binding sites
DBD	Predicts DNA-binding domains of transcription factors
JASPAR	Predicts transcription factor binding motifs
Weeder	Predicts likely transcription factor binding motifs
Chaperones	
BiPPred	Predicts the interactions of mammalian proteins with chaperone BiP
cleverSuite	Predicts the interactions of Escherichia coli proteins with chaperone DnaK/GroEL
LIMBO	Predicts the interactions of Escherichia coli proteins with chaperone DnaK
Reaction center and atom mapping	
Review	Review of methods for reaction mapping and reaction center detection
CAM	Predicts the mapping of reactant to product atoms

Tool	Prediction(s)
CLCA	Predicts the mapping of reactant to product atoms
MWED	Predicts the mapping of reactant to product atoms
ReactionDecoder	Predicts the mapping of reactant to product atoms
ReactionMap	Predicts the mapping of reactant to product atoms

1.4.2 Modeling and simulation tools

Here, we review several advances in modeling and simulation technology that we believe are beginning to enable researchers to aggregate and organize the data needed for WC modeling and design, describe, simulate, calibrate, verify, and analyze WC models.

Data aggregation and organization tools

To make the large amount of publicly available data usable for modeling, researchers are developing tools such as BioServices [54] for programmatically accessing repositories and using PGDBs to organize the data needed for modeling. PGDBs are well-suited to organizing the data needed for WC models because they support structured representations of metabolites, DNA, RNA, proteins, and their interactions. However, traditional PGDBs provided limited support for non-metabolic pathways and quantitative data. Consequently, we are developing `WholeCellKB`, a PGDB specifically designed for WC modeling [30].

Model design tools

Several software tools have been developed for designing models of individual cellular pathways including BioUML [67], CellDesigner [68], COPASI [69], JDesigner [70], and Virtual Cell [71] which support dynamical modeling; RuleBender which supports rule-based modeling [72]; and COBRApy [73], FAME [74], and RAVEN [75] which support constraint-based metabolic modeling; and (Table 1.4).

Recently, researchers have developed several tools that support some of the features needed for WC modeling. This includes SEEK which helps researchers design models from data tables [76], Virtual Cell which helps researchers design models from KEGG pathways [71][56], MetaFlux which helps researchers design metabolic models from PGDBs [61], the Cell Collective [77] and JWS Online [78] which help researchers build models collaboratively, PySB which helps researchers design models programmatically [79], and semanticSBML [80] and SemGen [81] which help researchers merge models.

Table 1.4: Software tools that can be used to help build, calibrate, validate, simulate, visualize, and analyze WC models.

Tool	URL	Reference
Data aggregation tools		
BioCatalogue	https://www.biocatalogue.org	Bhagat et al., 2010
BIOSERVICES	https://pythonhosted.org/bioservices	Cokelaer et al., 2013
Data organization tools		
GMOD	http://gmod.org	Papanicolaou and Heckel, 2010
Pathway Tools	http://brg.ai.sri.com/ptools	Karp et al., 2016
WholeCellKB	http://www.wholecellkb.org	Karr et al., 2013
Model design tools		
CellDesigner	http://www.celldesigner.org	Matsuoka et al., 2014
COPASI	http://copasi.org	Mendes et al., 2009
JWS Online	http://jjj.biochem.sun.ac.za	Olivier and Snoep, 2004

Continued on next page

Table 1.4 – continued from previous page

Tool	URL	Reference
MetaFlux	http://brg.ai.sri.com/ptools	Latendresse et al., 2012
PhysioDesigner	http://www.physiodesigner.org	Asai et al., 2012
RAVEN	http://biomet-toolbox.org/index.php?page=downtools-raven	Agren et al., 2013
RuleBender	http://bionetgen.org/index.php/Quick_Start	Smith et al., 2012
VirtualCell	http://vcell.org	Schaff et al., 2016
Model testing and verification tools		
biolab	http://www.lehman.edu/academics/cmacs/bio-lab.php	Clarke at al., 2008
MEMOTE	https://memote.readthedocs.io	
SBML-to-PRISM	http://www.prismmodelchecker.org/sbml	
Model description languages		
BioNetGen	http://bionetgen.org	Harris et al., 2016
BioPAX	http://www.biopax.org	Demir et al., 2010
CellML	https://www.cellml.org	Cuellar et al., 2015
kappa	http://dev.executableknowledge.org	Wilson-Kanamori et al., 2015
ML-Rules	http://jamesii.informatik.uni-rostock.de/jamesii.org/	Maus et al., 2011
PySB	http://pysb.org/	Lopez et al., 2013
SBML	http://sbml.org	Hucka et al., 2015
Simulation description languages		
SED-ML	http://sed-ml.org	Waltemath et al., 2011
SESSL	http://sessl.org	Ewald and Uhrmacher, 2014
Simulators		
cobrapy	http://opencobra.github.io/cobrapy	Ebrahim et al., 2013
COPASI	http://copasi.org	Mendes et al., 2009
ECell	http://www.e-cell.org	Takahashi et al., 2003
Lattice Microbes	http://www.scs.illinois.edu/schulten/lm	Hallock et al., 2014
libRoadRunner	http://libroadrunner.org	Somogyi et al., 2015
NFSim	http://michaelsneddon.net/nfsim	Sneddon et al., 2011
VirtualCell	http://vcell.org	Schaff et al., 2016
Simulation result formats		
HDF5	https://support.hdfgroup.org/HDF5	Folk et al., 2011
NuML	https://github.com/numl/numl	Dada et al., 2017
SBRML	http://www.comp-sys-bio.org/SBRML.html	Dada et al., 2010
Simulation result databases		
Bookshelf	http://sbcb.bioch.ox.ac.uk/bookshelf	Vohra et al., 2010
Dynameomics	http://www.dynameomics.org	van der Kamp et al., 2010
SEEK	https://fair-dom.org/platform/seek	Wolstencroft et al., 2011
WholeCellSimDB	http://www.wholecellsimdb.org	Karr et al., 2014
Visualization tools		
Vega	https://vega.github.io	Satyanarayan et al., 2017
The Visualization Toolkit (VTK)	http://www.vtk.org	Hanwell et al., 2015
WholeCellViz	http://www.wholecellviz.org	Lee et al., 2013
Workflow management tools		
Galaxy	https://usegalaxy.org	Walker et al., 2016
Taverna	http://www.taverna.org.uk	Wolstencroft et al., 2013
VizTrails	https://www.vistrails.org	Freire and Silva, 2012

However, none of these tools are well-suited to WC modeling because none of these tools support all of the features needed for WC modeling including programmatically designing models from large data sources such as PGDBs; collaboratively designing models over a web-based interface; designing composite, multi-algorithmic models; representing models in terms of rule patterns; and recording the data sources and assumptions used to build models.

Model selection tools

Several methods have also been developed to help researchers select among multiple potential models, including likelihood-based, Bayesian, and heuristic methods [82]. ABC-SysBio [83][84], ModelMage [85], and SYSBIONS [86] are some of the most advanced model selection tools. However, these tools only support deterministic dynamical models.

Model refinement tools

Several tools have been developed for refining models, including using physiological data to identify molecular gaps in metabolic models and using databases of molecular mechanisms to fill molecular gaps in metabolic models [87][88]. GapFind uses mixed integer linear programming to identify all of the metabolites that cannot be both produced and consumed in metabolic models, one type of molecular gap in metabolic models [89]. GapFill [89], OMNI [90], and SMILEY [91] use linear programming to identify the most parsimonious set of reactions from reaction databases such as KEGG [56] to fill molecular gaps in metabolic models. FastGapFill is one of the most efficient of these gap filling tools [92]. GrowMatch extends gap filling to find the most parsimonious set of reactions that not only fill molecular gaps in metabolic models, but also correct erroneous gene essentiality predictions [93]. ADOMETA [94], GAUGE [95], likelihood-based gap filling [96], MIRAGE [97], PathoLogic [98] and SEED [99] extend gap filling further by using sequence homology and other genomic data to identify the genes which most likely catalyze missing reactions in metabolic networks. However, these tools are only applicable to metabolic models.

Model formats

Several formats have been developed to represent cell models including formats such as CellML [100] that represent models as collections of variables and equations, formats such as SBML [101] that represent models as collections of species and reactions, and more abstract formats such as BioNetGen [102], Kappa [103], and ML-Rules [104] that represent models as collections of species and rule patterns.

The Systems Biology Markup Language (SBML) was developed in 2002 to represent dynamical models that can be simulated by integrating ordinary differential equations or using the stochastic simulation algorithm, as well as the semantic biological meaning of models. Recently, SBML has been extended to support a wide range of models through the development of several new packages. The flux balance constraints package supports constraint-based models, the qualitative models package supports logical models, the spatial processes package support spatial models that can be simulated by integrating PDEs, the multistate multicomponent species package supports rule-based model descriptions, and the hierarchical model composition package supports composite models. SBML is by far the most widely supported and commonly used format for representing cell models. For example, SBML is supported by COPASI [69], the most commonly used cell modeling software program and BioModels, the most commonly used cell model repository [105]. However, SBML creates verbose model descriptions, the multistate multicomponent species package only supports a few types of combinatorial complexity, SBML does not directly support multi-algorithmic models, and SBML cannot represent model provenance including the data sources and assumptions used to build models [106].

More recently, Faeder and others have developed BioNetGen [102] and other rule-based formats to efficiently describe the combinatorial complexity of protein-protein interactions. These formats enable researchers to describe models in terms of species and reaction patterns which can be evaluated to generate all of the individual species and reactions in a model. This abstraction helps researchers describe reactions directly in terms of their chemistry, describe large models concisely, and avoid errors in enumerating species and reactions. Models that are described in rule-based formats such as BioNetGen can be simulated either by enumerating all of the possible species and reactions and then simulating the expanded model via conventional deterministic or stochastic dynamical simulation methods, or via network-free simulation which iteratively discovers individual species and reactions during simulation [107]. BioNetGen is the most commonly used rule-based modeling format and NFsim is the most commonly used network-free simulator. However, BioNetGen only supports few types of combinatorial complexity, BioNetGen does not support composite or multi-algorithmic models, BioNetGen cannot represent the semantic biological meaning of models, and BioNetGen cannot represent model provenance.

Simulation algorithms

Several algorithms have been developed to simulate cells with a wide range of granularity including algorithms for integrating systems of ODEs and PDEs, stochastic simulation algorithms, algorithms for simulating logical networks and Petri nets, and hybrid algorithms for co-simulating models that are composed of mathematically-dissimilar submodels.

The most commonly used algorithms to simulate cell models include algorithms for integrating systems of ODEs. These algorithms are best suited to simulating well-characterized and well-mixed systems that involve large concentrations that are robust to stochastic fluctuations. These algorithms are poorly suited to simulating stochastic processes that involve small concentrations, as well as poorly characterized pathways with little kinetic data. Consequently, ODE integration algorithms are poorly suited for WC modeling.

Stochastic simulation algorithms such as the Stochastic Simulation Algorithm (SSA) or Gillespie's Algorithm [108], newer, more efficient implementations of SSA such as the Gibson-Bruck method and RSSA-CR [109], and approximations of SSA such as tau leaping, are commonly used to simulate pathways that involve small concentrations that are susceptible stochastic variation. However, these algorithms are only suitable for dynamical models which require substantial kinetic data, they are computationally expensive, especially for models that include reactions that have high fluxes, and they are limited to models with small state spaces. Consequently, stochastic simulation algorithms are poorly suited for simulating WC models.

Network-free simulation algorithms are stochastic simulation algorithms for efficiently simulating rule-based models without enumerating every possible species and reaction prior to simulation and instead discovering the active species and reactions during simulation. Unlike traditional stochastic simulation algorithms, network-free simulation algorithms can represent large models that have combinatorially large or even infinite state spaces. Otherwise, network-free stochastic simulation algorithms have the same limitations as other stochastic simulation algorithms.

FBA is the second-most commonly used algorithm for simulating cell models. FBA predicts the steady-state flux of each metabolic reaction using detailed information about the stoichiometry and catalysis of each reaction, a small amount of quantitative data about the chemical composition of cells, a small amount of data about the exchange rate of each extracellular nutrient, and the assumption that metabolism has evolved to maximize the rate of cellular growth. However, FBA has limited ability to predict metabolite concentrations and temporal dynamics, and its assumptions are largely only applicable to microbial metabolism. Consequently, FBA is not well-suited to simulating entire cells.

Logical simulation algorithms are frequently used for coarse-grained simulations of transcriptional regulation and other pathways for which we have limited kinetic data. Logical simulations are computationally efficient because they are coarse-grained. However, logical simulation algorithms are poorly suited to WC modeling because they cannot generate detailed quantitative predictions, and therefore have limited utility for medicine and bioengineering.

Multi-algorithmic simulations are ideal for WC modeling because they can simulate models that include fine-grained representations of well-characterized pathways, as well as coarse-grained representations of poorly-characterized pathways. Takahashi et al. developed one of the first algorithms for co-simulating multiple mathematically-dissimilar submodels [3]. However, their algorithm is not well-suited to WC modeling because it does not support FBA or network-free simulation. Recently, we and others developed a multi-algorithm simulation meta-algorithm which supports ODE integration, conventional stochastic simulation, network-free stochastic simulation, FBA, and logical simulation [27]. However, our algorithm violates the arrow of time and is not scalable to large models.

Simulation experiment formats

The Minimum Information About a Simulation Experiment (MIASE) guidelines have been developed to establish the minimum metadata that should be provided about a simulation experiment to enable other researchers to reproduce and understand the simulation [2]. The Simulation Experiment Description Markup Language (SED-ML) [110] and the Simulation Experiment Specification via a Scala Layer (SESSL) [111] formats have been developed to represent simulation experiments. Both formats are capable of representing all of the model parameters and simulator arguments needed to simulate a model. However, both formats are limited to a small range of model formats and simulators. SED-ML is limited to models that are represented using XML-based formats such as SBML, and SESSL is currently limited to Java-based simulators. Consequently, neither is currently well-suited to WC modeling.

Simulation tools

Numerous tools have been developed to simulate cell models including the BioUML [67], Cell Collective [77], COBRApy [73], COPASI [69], E-Cell [112], FAME [74], iBioSim [113], libRoadRunner [114], JWS Online [78], NFsim [107], RAVEN [75], and Virtual Cell [71].

COPASI is the most commonly used simulation tool. COPASI supports several deterministic, stochastic, and hybrid deterministic/stochastic simulation algorithms. However, COPASI does not support network-free stochastic simulation, FBA, logical, or multi-algorithmic simulation and COPASI does not support high-performance parallel simulation of large models.

Virtual Cell supports several deterministic, stochastic, hybrid deterministic/stochastic, network-free, and spatial simulation algorithms. However, Virtual Cell does not support FBA or multi-algorithmic simulations and Virtual Cell does not support high-performance parallel simulation of large models.

COBRApy, FAME, and RAVEN support FBA of metabolic models. However, these packages provide no support for other types of models.

E-Cell is one of the only simulation programs that supports multi-algorithmic simulation. However, E-Cell does not support FBA or rule-based simulation, and E-Cell does not scale well to large models.

Several tools including cupSODA [115], cuTauLeaping [116], and Rensselaer's Optimistic Simulation System (ROSS) [117] have been developed to simulate models in parallel. However, cupSODA only supports deterministic simulation, cuTauLeaping only supports network-based stochastic simulation, cupSODA and cuTauLeaping only support GPUs, and ROSS is a low-level, general-purpose framework for distributed CPU simulation.

Calibration tools

Accurate parameter values are essential for reliable predictions. Many methods have been developed to calibrate models by numerically optimizing the values of their parameters, including derivative-based initial value methods and stochastic multiple shooting methods [118].

Several complementary methods have also been developed to optimize computationally-expensive, high-dimensional functions, including surrogate modeling, distributed optimization, and automatic differentiation. Surrogate modeling, which is also referred to as function approximation, metamodeling, response surface modeling, and model emulation, promises to reduce the computational cost of numerical optimization by optimizing a computationally cheaper model which approximates the original model [119][120][121][122]. Surrogate modeling has been used in several fields including aerospace engineering [123], hydrology [124], and petroleum engineering [125]. However, further work is needed to develop methods for efficiently generating reduced surrogate WC models.

Distributed optimization is also a promising approach for optimizing computationally expensive functions. Distributed optimization uses multiple agents, each simultaneously employing the same algorithm on different regions, to quickly identify optima [126][127]. Furthermore, agents can cooperate by exchanging information. Distributed optimization has been used in several fields including aerospace and electrical engineering [128][129] and molecular dynamics [130].

Another promising approach for optimizing computationally expensive functions is automatic differentiation. Automatic differentiation is an efficient technique for analytically computing the derivative of a function [131]. Automatic differentiation can be used to make derivative-based optimization methods tractable in cases where finite difference calculations are prohibitively expensive. Automatic differentiation has been used to identify parameters in chemical engineering [132], biomechanics [133], and physiology [134].

Several software tools have also been developed for calibrating cell models [135][136][137][138][139]. Some of the most advanced model calibration tools include DAISY which can evaluate the identifiability of a model [140], ABC-SysBio which uses approximate Bayesian computation [83], saCeSS which supports distributed, collaborative optimization [141], and SBSI which supports several distributed optimization methods [142]. Some of the most popular modeling tools, including COPASI [69] and Virtual Cell [71], also provide model calibration tools. However,

none of these tools support multi-algorithmic models. To efficiently calibrate WC models, we should combine numerical optimization methods with additional techniques such as reduced surrogate modeling, distributed computing, and automatic differentiation.

Verification tools

Several tools have been developed to verify cell models, including formal verification tools that seek to prove or refute mathematical properties of models and informal verification tools that help modelers organize and evaluate computational tests of models. BioLab [143] and PRISM [144] are formal tools for verifying BioNetGen-encoded and SBML-encoded models, respectively. Memote [145] and SciUnit [146] are unit testing frameworks for organizing computational tests of models. Continuous integration tools such as CircleCI [147] and Jenkins [148] can be used to regularly verify models each time they are modified and pushed to a version control system (VCS) such as Git [149].

Simulation results formats

HDF5 is an ideal format for storing simulation results [150]. In particular, HDF5 supports hierarchical data structures, HDF5 supports compression, HDF5 supports chunking to facilitate fast retrieval of small slices of large datasets, HDF5 can store both simulation results and their metadata, and there are HDF5 libraries available for several languages including C++, Java, MATLAB, Python, and R.

Simulation results databases

Several database systems have been developed to organize simulation results for visual and mathematical analysis and disseminate simulation results to the community [151][152][153][154][155][156][157]. We developed `WholeCellSimDB`, a hybrid relational/HDF5 database, to organize, search, and share WC simulation results [158]. WholeCellSimDB uses HDF5 to store simulation results and a relational database to store their metadata. This enables WholeCellSimDB to efficiently store simulation results, quickly search simulations by their metadata, and quickly retrieve slices of simulation results. WholeCellSimDB providers uses two interfaces to deposit simulation results; a web-based interface to search, browse, and visualize simulation results; and a JSON web service to retrieve simulation results. However, further work is needed to scale WholeCellSimDB to larger models and to develop tools for quickly searching WholeCellSimDB.

Simulation results analysis

Several tools have been developed to analyze and visualize simulation results. The most popular simulation software programs, including COPASI [69], E-Cell [112], and Virtual Cell [71], provide basic tools for visualizing simulation results. Tools such as Escher [159] and Pathway Tools Omics Viewer [160] can also be used to visualize simulation results.

We developed `WholeCellViz` to visualize WC simulation results in their biological context [161]. WholeCellViz provides users time series plots and interactive animations to visualize model predictions, and enables users to arrange grids of plots and animations to help users compare predictions across multiple simulation runs and simulated conditions. However, further work is needed to scale WholeCellViz to larger models and to make it easier to incorporate new visualizations into WholeCellViz.

1.4.3 Models of individual pathways and model repositories

Since the 1950's, researchers have been using the tools described above to model cells. This has led to numerous models that represent individual pathways. Here, we review our progress in modeling individual pathways, building repositories of cell models, and their utility for WC modeling.

Models of individual pathways

Over the past 30 years, researchers have developed a wide range of models of individual cellular pathways [105] (Figure 1.4, Table 1.5). In particular, researchers have developed models of cell cycle regulation [162]; circadian rhythms [163]; electrical signaling [164]; metabolism [165][166][167]; signaling pathways such as the JAK/STAT, NF-κB, p53, and TGFβ pathways [168]; transcriptional regulation [169], and multicellular processes such as developmental patterning [170] and infection. However, many pathways have not been modeled at the scale of entire cells, including several well-studied pathways. For example, although we have extensive knowledge of the mutations responsible for cancer, we have few models of DNA repair; although we have extensive structural and catalytic information about RNA modification, we have few kinetic models of RNA modification; and although we have detailed atomistic models of protein folding, we have few cell-scale models of chaperone-mediated folding.

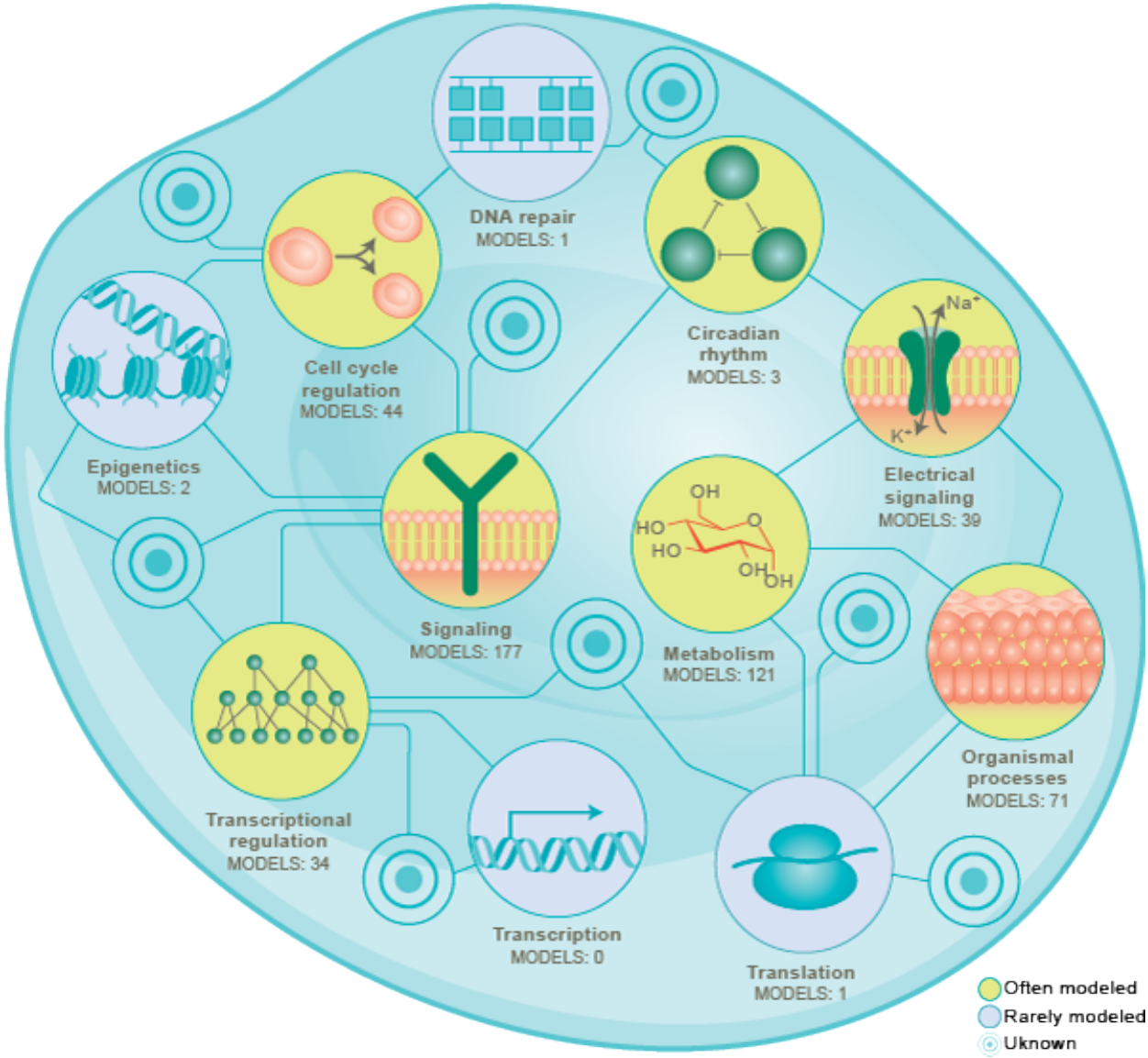


Figure 1.4: WC models can be built by leveraging existing models of well-studied processes (colors) and developing new models of other processes (gray).

Table 1.5: Pathway distribution, computational representations, and taxonomic distribution of the models contained in the BioModels model repository (Chelliah et al., 2015).

		Number of models in BioModels, by kingdom				Mean model size		
Pathway	Formalisms	Viruses	Eukaryotes	Bacteria	Unannotated	Species	Reactions	Parameters
Cell cycle	ODEs, SSA		44			14.0	19.4	33.6
Cell death	ODEs		11		2	24.5	33.6	42.2
Circadian regulation	ODEs		38		1	17.3	31.2	65.5
DNA repair	ODEs		1			23.0	25.0	26.0
Electrical signaling	ODEs		34		5	12.7	26.4	37.5
Gene expression regulation	Boolean network		9	10	5	11.9	14.0	15.5
Host-pathogen interaction	ODEs	1	2	1		24.3	44.5	58.0
Intracellular transport	ODEs		2		2	7.8	12.8	16.3
Macromolecule modification	ODEs		1		2	10.7	26.0	19.7
Metabolism	FBA, ODEs		100	16	5	57.0	39.7	195.6
Motility	ODEs, PDEs		2	2		40.8	48.3	79.5
Organismal process	ODEs	1	66	2	2	17.2	20.1	48.8
Regulation, other	ODEs		5		14	12.0	17.8	22.2
Signal transduction	ODEs, SSA		144	3	30	35.3	54.1	67.8
Stress response	ODEs		9			16.6	19.4	46.2

Collectively, these models span a broad range of scales. For example, although most of these models represent the chemical transformations responsible for each pathway, some of these models, such as most transcriptional regulation models, use coarser representations. As a second example, although most of these models represent temporal dynamics, most metabolic models only represent the steady-state behavior of metabolism [13]. Similarly, although most of these models represent cells as well-mixed bags, some of these models represent the spatial distribution of individual compounds including nutrients and hormones [171][172][173]. In addition, although most of these models represent the mean behavior of cells, averaged over multiple cells and cell cycle phases, a few of these models represent the temporal dynamics of the cell cycle and the variation among single cells.

Collectively, these models also use a wide range of computational representations and simulation algorithms. Many of these models are represented as reaction networks. However, some of the largest of these models must be represented using rules [102] or Boolean networks. Many of these models can be simulated by integrating ODEs. However, some of the largest models must be simulated using network-free methods [107], the steady-state metabolism models must be simulated with FBA [13], some of the spatiotemporal models must be simulated by integrating PDEs, and some of the network models must be simulated by iteratively evaluating Boolean regulatory functions [174].

These pathway models could be used to help build WC models. However, substantial work would be required to integrate these models into a single model because these models describe different scales, make different assumptions, are represented using different mathematical formalisms, are calibrated to different organisms and conditions, and are represented using different identifiers and formats. To avoid needing to substantially revise pathway models for

incorporation into WC models, modelers should build pathway models explicitly for integration into WC models. This requires the modeling community to embrace a common format, common identifiers, common units, and common standards for model calibration and validation.

1.4.4 Models of multiple pathways

Since 1999 when Tomita *et al.* reported one of the first models of multiple pathways of *M. genitalium* [17], researchers have been trying to build increasingly comprehensive models of multiple pathways. In particular, this has led to models of *Escherichia coli* and *Saccharomyces cerevisiae* which describe their metabolism and transcriptional regulation [18][19]; their metabolism, signaling, and transcriptional regulation [20][21][22]; and their metabolism and RNA and protein synthesis and degradation [23]. Table 1.6 summarizes several recently published and proposed models of multiple pathways. Despite this progress, these models only represent a small number of pathways and a small number of organisms.

Table 1.6: Models of multiple cellular pathways and their computational representations.

Pathways		Computational representation	Species	Status	References
28	Chromosome Condensation, Chromosome Segregation, Cytokinesis, DNA damage, DNA repair, DNA supercoiling, FtsZ Polymerization, Host interaction, Macromolecular complexation, Metabolism, Protein activation, Protein decay, Protein folding, Protein modification, Protein processing I, Protein processing II, Protein translocation, Replication, Replication Initiation, Ribosome assembly, RNA decay, RNA modification, RNA processing, Terminal organelle assembly, Transcription, Transcriptional regulation, Translation, tRNA aminoacylation	Hybrid: Boolean, flux balance analysis, ordinary differential equations, stochastic simulation	Mycoplasma genitalium	Published	Karr et al., 2012
6	Metabolism, protein complexation, RNA maturation, RNA modification, transcription, translation	Flux balance analysis	Escherichia coli	Published	Thiele et al., 2009
5	Metabolism, protein degradation, RNA degradation, transcription, translation	Ordinary differential equations	Mycoplasma genitalium	Published	Tomita et al., 1999
3	Circadian rhythms, metabolism, transcriptional regulation	Hybrid: flux balance analysis, ordinary differential equations	Synechocystis sp. PCC 6803	Proposed	Steuer et al., 2012
3	Contraction, electrical signling, metabolism	Ordinary differential equations	Homo sapiens	Proposed	Bassingthwaighte et al., 2005
3	Metabolism, signal transduction, transcriptional regulation	Hybrid: Boolean, flux balance analysis, ordinary differential equations	Escherichia coli	Published	Covert et al., 2008
3	Metabolism, signal transduction, transcriptional regulation	Hybrid: constraint-based modeling, ordinary differential equations, phenomenological modeling	Escherichia coli	Published	Carrera et al., 2014
5	Metabolism, signal transduction, transcriptional regulation	Ordinary differential equations	Saccharomyces cere-	Published	Klipp et al., 2005

To represent multiple pathways, most of these models have been developed by combining separate submodels of each pathway, using the most appropriate mathematical representation for each pathway. This has led to multi-algorithmic models which must be simulated by co-simulating the individual submodels. Because there are few multi-algorithmic simulation tools and most of these models only combine two or three submodels, the developers of most of these models have developed ad hoc methods to simulate their models. For example, Covert et al. developed an ad hoc method to simulate their hybrid dynamic FBA / Boolean model of the metabolism and transcriptional regulation of *E. coli* [18] and Chandrasekaran and Price developed a different ad hoc method to simulate their hybrid FBA / Bayesian model of the metabolism and transcriptional regulation of *E. coli* [19]. Because there are few tools for working with such integrative models, these models have also been described with different ad hoc formats and identifiers, simulated with different ad hoc simulation software programs, and calibrated and validated with different ad hoc methods.

Model repositories

Several model repositories, including BioModels [105] and the Physiome Model Repository [175], have been developed to make it easy to find models (Table 1.7). However, only a few of these repositories support integrated models; most of these repositories only support a limited number of model formats; many reported models are never deposited to any model repository; many of the models that are deposited are not sufficiently annotated for other researchers to understand, reuse, and extend the models; and only a few of the repositories also support the information needed to simulate models such as parameter values.

Table 1.7: Repositories that contain published models that can be modified, extended, and combined to create WC models.

Repository	Content	URL	Reference
BiGG	Repository for constraint-based models of metabolism	http://bigg.ucsd.edu	King et al., 2016
BioModels	Repository for SBML-encoded models that contains many cell cycle, circadian, electrical signaling, metabolism, and signal transduction models	http://www.ebi.ac.uk/biomodels-main	Chelliah et al., 2015
FigShare	Repository for supplemental materials that contains some models	https://figshare.com	
GitHub	Repository for code that contains some models	https://github.com	
JWS Online	Online environment for systems biology modeling that includes a model repository	http://jjj.biochem.sun.ac.za	Peters et al., 2017
Open Source Brain	Repository for NeuroML-encoded models of neurophysiology	http://www.opensourcebrain.org	Gleeson et al., 2012
Physisome Repository	Repository for CellML-encoded models that contains physiological models	https://models.physiomeproject.org	Yu et al., 2011
SimTK	Repository for data and code that contains several biomechanics models	https://simtk.org	

1.5 Emerging principles and methods for WC modeling

In the previous section, we outlined the ongoing technological advances that are making WC modeling feasible. Here, we propose several principles for WC modeling and describe how we and others are adapting and integrating these

technologies into a methodology for WC modeling. In the following sections, we outline the major remaining bottlenecks to WC modeling, highlight ongoing efforts to overcome these bottlenecks, and describe how we are beginning to use this methodology to build WC models.

1.5.1 Principles of WC modeling

Based on our experience, we propose several guiding principles for WC modeling (Figure 1.5).

- **Modular modeling.** Similar to other large engineered systems such as software, WC models should be built by partitioning cells into pathways, outlining the interfaces among these pathways, building submodels of each pathway, and combining these submodels into a single model. This approach reduces the dimensionality of model construction, calibration, and validation and facilitates collaborative modeling.
- **Multi-algorithmic simulation.** Furthermore, to capture both well- and poorly-characterized pathways, each pathway should be represented using the most appropriate mathematical representation given our knowledge and data about each pathway. In particular, multi-algorithmic simulation should be used to create identifiable models which can be calibrated from our experimental data.
- **Experimental calibration and validation.** WC models should be rigorously calibrated and extensively validated via comparison to detailed experimental data across a wide range of molecular mechanisms, phenotypes, and scales.
- **Systemization and standards.** To scale modeling to entire cells and facilitate collaboration, we should systemize every aspect of dynamical modeling, develop standards for describing WC models and standard protocols for validating and merging model components, and encourage researchers to embrace these standard protocols and formats.
- **Technology development.** To enable WC modeling, we must develop technologies for systematically and scalably building, calibrating, simulating, and validating WC models. These technologies should be modular to facilitate collaborative technology development and integrated into a unified framework to provide modelers user-friendly modeling and simulation tools.
- **Leverage existing methods and data.** Where possible, WC modeling should take advantage of existing computational methods and experimental data. For example, WC modeling should take advantage of parallel simulation methods developed by computer science and WC models should be built, in large part, from data aggregated from public repositories.
- **Focus on critical problems and clear, achievable goals.** To maximize our efforts, we should periodically identify the key bottlenecks to WC modeling and periodically refocus our efforts on overcoming these bottlenecks. Based on lessons learned from other "big science" projects [176][177], we should also delineate clear goals and clearly define the responsibilities of each researcher.
- **Focus on model organisms.** To facilitate collaboration, early WC modeling efforts should focus on a small number of organisms and cell lines that are easy to culture, well-characterized, karyotypically and phenotypically "normal", genomically stable and relevant to a wide range of basic science, medicine, and bioengineering. This includes well-characterized bacteria such as *Escherichia coli* and well-characterized human cell lines such as the H1 human embryonic stem cell (hESC) line.
- **Reproducibility, transparency, extensibility, and openness.** To facilitate collaboration and maximize impact, WC models and simulations should be reproducible, comprehensible, and extensible. For example, to enable other modelers to understand a model, the biological semantic meaning of each species and reaction should be annotated, the data sources and assumptions used to design the model should be annotated, and the parameter values used to produce each simulation result should be recorded. Furthermore, each WC model and WC modeling technology should be free and open-source.
- **Constant innovation.** Because we do not yet know exactly what WC models should represent, what WC models should predict, or how to build WC models, we should periodically evaluate the quality of our models and methods and iteratively improve our models and methods as we learn more about cell biology and WC

modeling. This should include how we partition cells into pathways, the interfaces that we define among the pathways, and how we simulate multi-algorithmic models.

- **Interdisciplinary collaboration.** WC modeling should be an interdisciplinary collaboration among modelers, experimentalists, computer scientists, and engineers, and research sponsors. Furthermore, there should be open and frequent communication among the WC modeling community.

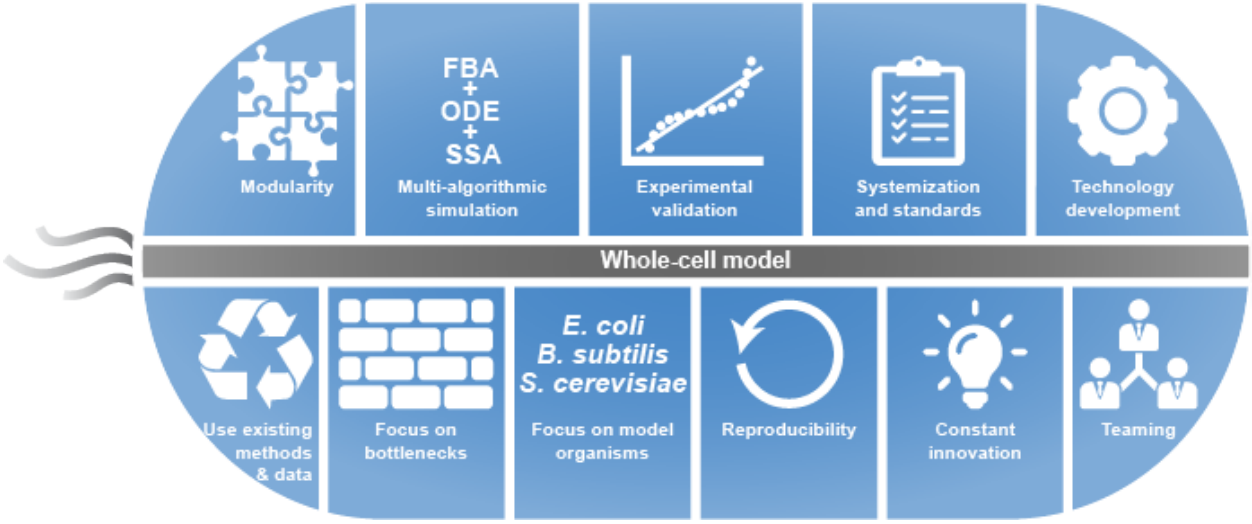


Figure 1.5: Principles of WC modeling.

1.5.2 Methods for WC modeling

To enable WC models, we and others are adapting and integrating the technologies described in Section 1.4.2 into a workflow for scalably building, simulating, and validating WC models (Figure 1.6). (1) Modelers will use `Datanator` to aggregate, standardize, and integrate the experimental data that they will need to build, calibrate, and validate their model into a single dataset. (2) Modelers will use this data to design submodels of each individual pathway using the most appropriate mathematical representation for each pathway, and encode their model in `wc_rules`, a rule-based format for describing WC models. (3) Modelers will construct reduced models, and use them to calibrate each submodel and their entire model. (4) Modelers will use formal verification and/or unit testing to verify that their model functions as intended and recapitulates the data used to build the model. (5) Modelers will use `wc_sim`, a scalable, network-free, multi-algorithmic simulator, to simulate their model. (6) Modelers will use `WholeCellSimDB` to organize their simulation results and use `WholeCellViz` to visually analyze these results. Importantly, every tool in this workflow will facilitate collaboration to help researchers work together, and these tools will be modular to enable us and others to continuously improve this methodology. We plan to implement this workflow by leveraging recent advances in computational and experimental technology (Section 1.4). Here, we describe the six steps of this emerging workflow.

Data aggregation, standardization, and integration

The first step of WC modeling is to aggregate, standardize, integrate, and select the experimental data needed for WC modeling into a single dataset for model building, calibration, and validation (Figure 1.6a).

First, we must aggregate a wide range of experimental data from a wide range of databases such as such as biochemical data about metabolite concentrations from ECMDB [47], RNA-seq data about RNA concentrations from ArrayExpress [31], and mass-spectrometry data about metabolite concentrations from PaxDb [33]. Where possible, data should be aggregated using database downloads and web services. Otherwise, data should be aggregated by

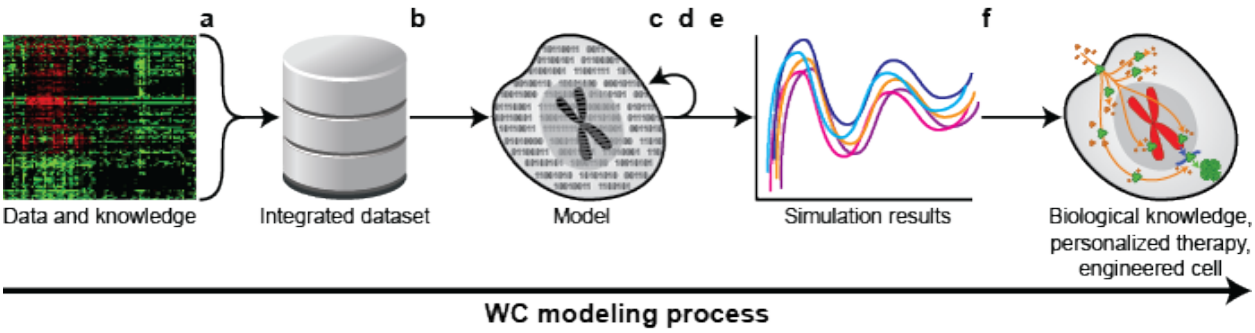


Figure 1.6: Emerging workflow for scalably building, simulating, and validating WC models. (**a**) Modelers will aggregate the data for WC modeling into a single dataset. (**b**) Modelers will use this data to design multi-algorithmic WC models. (**c,d**) Modelers will use reduced models to calibrate, verify, and validate models. (**e**) Modelers will simulate multi-algorithmic WC models by co-simulating their submodels. (**f**) Modelers will visualize and analyze their results to discover new biology, personalize medicine, and design microorganisms.

scraping webpages. In addition to aggregating data from databases, we should also aggregate data from collaborators, individual publications, and bioinformatics prediction tools such as PSORTb [65] and TargetScan [66].

To the extent possible, we should record the provenance of this data including the biosample (e.g., species, strain, genetic variants) and environmental conditions (e.g., temperature, pH, growth media) that were measured, the experimental method used to generate the data, the computational method used to analyze the data, and the citation for original data to help us select the most relevant data for modeling and trace models back to their data sources.

Second, we must standardize the identifiers and units used to describe this data. For example, metabolites should be identified using the IUPAC International Chemical Identifier (InChI) format [178] and RNA should be identified by their genomic coordinates. Similarly, all units should be standardized to SI units or combinations of SI units.

Third, we must integrate this data by linking the data together through common metabolites, chromosomes, RNA, proteins, and interactions. To enable this data to be quickly searched and explored, this data should be organized into a relational database.

Fourth, we must identify the most relevant data within our database for the species and environmental condition that we want to model. For each experimental measurement that we need to constrain a model, we must search our database for data observed for similar biology (e.g., metabolites, RNA, proteins, and interactions), genotypes (e.g., species, strain, and genetic variants), and environmental conditions (e.g., temperature, pH, growth media); calculate the relevance of each experimental observation; and calculate the consensus of the relevant observations, weighted by their relevance.

Fifth, we should organize these consensus experimental values and their provenance (experimental evidence and the method used to calculate the consensus value) into a single dataset. Pathway/genome databases (PGDB) can be used to organize this information because PGDBs are well-suited to representing relationships among experimental data about a single species. We have developed the `WholeCellKB` PGDB to organize the data needed for WC modeling. WholeCellKB provides users three interfaces to deposit experimental data for WC models, extensive functionality for validating this data, a web-based user interface to search and browse this data, and a JSON web service to programmatically retrieve data for model construction.

Model design

The second step of WC modeling is to use the data aggregated in the first step to design models, including each species and interaction (Figure 1.6b). To represent the details of well-characterized pathways, as well as coarsely represent poorly-characterized pathways, WC models should be built by partitioning cells into pathways, modeling each pathway using the most appropriate mathematical representation, and combining pathway submodels into composite, multi-algorithmic models.

To capture the large number of possible cellular phenotypes, WC models should also capture the combinatorial complexity of cellular biochemistry. For example, WC models should represent the combinatorial number of RNA transcripts that can be produced from the interactions of transcription, RNA editing, RNA folding, and RNA degradation; WC models should represent the combinatorial number of possible interactions among the subunits of protein complexes; and the combinatorial number of phosphorylation states of each protein complex.

To generate accurate predictions, WC models should also aim to represent the aggregate physiology of poorly understood biology such as uncharacterized genes, uncharacterized small peptides, and uncharacterized non-coding RNA. This can be accomplished by including lumped reactions that represent the aggregate physiology of all unknown biology. For example, to accurately predict metabolic reaction fluxes, like FBA models, WC models can include reactions that capture the aggregate energy usage of all uncharacterized interactions.

To scalably and reproducibly build WC models, WC models should be programmatically built from PGDBs using scripting tools such as PySB [79].

Because WC models will never be complete, WC models should be built by designing an initial model and then iteratively improving the model until the model accurately predicts new experimental measurements. In particular, WC models can be systematically refined by identifying gaps between their bottom-up descriptions of cellular biochemistry and our physiological knowledge, searching for reactions and gene products that might fill those gaps, and parsimoniously adding species and reactions to models so they recapitulate experimental observations. Model selection methods can also be used to select among multiple potential model designs. Furthermore, version control systems such as Git [149] should be used to track model changes and enable collaborators to refine models in parallel and merge their refined models.

To enable other researchers to reproduce, understand, reuse, and extend WC models, WC models should be encoded in rule-based formats such as BioNetGen and extensively annotated. In particular, rule-based formats enable researchers to concisely describe the combinatorial complexity of cell biology. Model annotations should include semantic annotations about the biological meaning of each species and interaction such as the chemical structure of each metabolite in InChI format [178] and provenance annotations about the data sources, assumptions, and design decisions behind each species, interaction, and pathway.

Model calibration

The third step in WC modeling is to calibrate model parameters (Figure 1.6c). This should be done by using numerical optimization methods to minimize the distance between the model's predictions and related experimental observations. One promising method for calibrating composite WC models is to (a) use multi-algorithmic modeling to only create parameters whose values can be constrained by one or a small number of experimental measurements, (b) estimate the value of each individual parameter using one or a small number of experimental observations, (c) construct a set of reduced models, one for each submodel, to estimate the joint values of the parameters, and (d) use distributed global optimization tools such as saCeSS [141] to refine the joint values of the parameters [179]. This method avoids the need to calibrate large numbers of parameters of physiological data; performs the majority of model calibration using low dimensional models of individual species, reactions, and pathways; and generates successively better starting points for more refined calibration.

Model verification and validation

The fourth step in WC modeling is to verify that models behave as intended and validate that models recapitulate the true biology (Figure 1.6d). First, WC should be verified models using a series of increasingly comprehensive unit tests that test each individual species, reaction, and pathway, as well as groups of pathways and entire models. Importantly, these tests should cover all of the logic of the model. For example, these tests should test the edge cases of every rate law. Reduced models should be used to efficiently test individual species, reactions, and pathway submodels. Furthermore, to quickly identify errors, continuous integration systems such as Jenkins [148] should be used to automatically execute tests each time models are revised. Alternatively, models can be verified using formal verification systems such as PRISM [144]. However, substantial work remains to adapt formal verification to multi-algorithmic dynamical modeling.

Second, WC models should be validated by comparing their simulation results to independent experimental data that was not used for model construction or calibration. To be effective, models should be tested using a broad range of data that spans different types of predictions, genetic perturbations, and environmental conditions.

Third, because it is infeasible to validate possible model prediction, modelers should annotate how models were validated to help other modelers know which model predictions can be trusted, know which predictions still need to be validated, and reuse the validation data to validate improved and/or extended models. These annotations should include which data were used for validation, which predictions were validated, and how well the model recapitulated each experimental observation. We believe that this metadata will be critical for medicine where therapy should only be driven by validated model predictions.

Network-free multi-algorithmic simulation

The fifth step of WC modeling is to numerically simulate WC models (Figure 1.6e). Because WC models should be described using rules and composed of multiple mathematically-dissimilar submodels, WC models simulated by co-simulating their submodels. This can be achieved in three steps. First, all of the submodels should be converted to explicit time-driven submodels. For example, Boolean submodels should be converted to SSA submodels by assuming typical concentrations and kinetic rates. Second, all of the mathematically-similar submodels should be analytically merged into a single mathematically-equivalent submodel. Third, for WC models that are composed only of FBA, ODE, and ODE submodels, (a) the SSA submodel should be used as the master clock for the integration and synchronization of the submodels, (b) each time the SSA submodel advances to the next iteration, the FBA and ODE submodels should be synchronized with the SSA submodel and integrated for the same timestep as the SSA submodel, (c) and the SSA submodel should be synchronized with the FBA and ODE models. If the FBA or ODE models generate unphysical states such as negative concentrations, they must be rolled back and reintegrated for multiple smaller timesteps. To efficiently simulate WC models, the FBA and ODE models should only be evaluated periodically.

To efficiently simulate the combinatorial complexity represented by WC models, most submodels should be simulated using SSA and SSA should be implemented using network-free graph-based methods. Specifically, SSA should be implemented by representing each molecule as a graph, representing each reaction rule as a graph, searching for matching pairs of species-reaction graphs to determine the rate of each reaction, randomly selecting a reaction to fire, updating the species involved in the selected reaction, and using a species-reaction dependency graph to update the rates of all affected reactions. This methodology will enable WC simulations to scale to large numbers of possible species and reactions by only representing the configuration of each active molecule rather than representing the copy number of each possible species.

To simulate WC models quickly, WC models should be simulated using a distributed simulation framework such as parallel discrete event simulation (PDES) and partitioning WC models into cliques of tightly connected species and reactions.

To make WC simulations comprehensible and reproducible, WC simulations should be represented using a common format such as SED-ML or SESSL.

Visualization and analysis of simulation results

The sixth step of WC modeling is to visualize and analyze WC simulation results to discover new biology, personalize medicine, or design microbial genomes (Figure 1.6f). First, all of the metadata needed to understand and reproduce simulation results should be recorded, including the model, the version of the model, the parameter values, and the random number generator seed that was simulated. Second, simulation results should be logged and stored in HDF5 format [150]. Third, WC simulation results and their metadata should be organized using a tool such as `WholeCellSimDB` that helps researchers search, slice, reduce, and share simulation results. Fourth, researchers should use tools such as `WholeCellViz` to visually analyze WC simulation results and use visualization grammars such as Vega [180] to develop custom diagrams.

1.6 Latest WC models and their limitations

Because it is not yet possible to completely model a cell, researchers are pursuing several complementary approaches to modeling entire cells. Historically, researchers such as Michael Shuler focused on building coarse-grained models of the major functions of cells [25][181]. Over the last ten years, researchers have begun to leverage the growing wealth of experimental data and our increasing computational power to build fine-grained models of the molecular biology of entire cells. This includes bottom-up efforts to represent the contribution of each gene to cellular behavior starting from genome sequences and annotations [27], top-down efforts to represent the integrated behavior of each cellular process, and bottom-up efforts to model diffusion at the cell scale [182][183][26]. More recently, researchers have begun to merge these fine-grained approaches. For example, Schulten recently demonstrated a hybrid FBA-diffusion model of *E. coli* [184]. Here, we describe recent progress in each of these major approaches to WC modeling.

1.6.1 Coarse-grained models

In adddition fine-grained models, researchers have also developed several coarse-grained models of multiple cellular processes [25][181]. These models could be used to help inform the global structure and mathematical behavior of WC models. However, they generally cannot be directly incorporated into WC models because they use coarse-grained representations that are incompatible with that of fine-grained WC models.

1.6.2 Genomically-centric bottom-up fine-grained models

Toward WC models, recently, we and others demonstrated the first model which represents every characterized gene function of a cell [27] (Figure 1.7a). The model represents 28 pathways of *M. genitalium*. The model was developed by annotating the *M. genitalium* genome, reconstructing the species encoded by each gene and the reactions catalyzed by each gene using data from over 900 databases and publications, partitioning the species and reactions into 28 pathways, developing separate submodels of each pathway, and integrating the submodels into a single model. To help us organize the data used to build the model, we developed WholeCellKB, a pathway/genome database (PGDB) software system tailored for WC modeling [30], and developed scripts to generate the model from the PGDB.

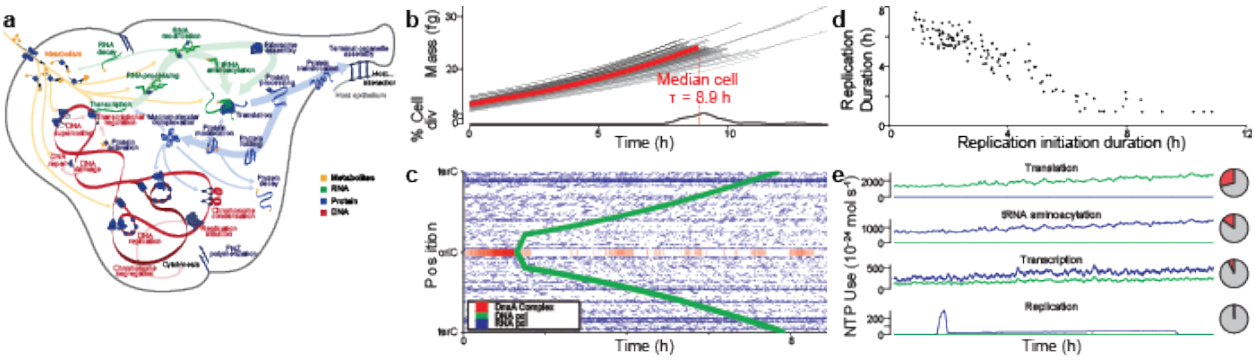


Figure 1.7: A WC model of *M. genitalium* predicts high-level cellular behaviors from the molecular level. (a) The model combines multiple submodels of individual cellular subsystems. We validated the model by comparing its outputs to experimental data which describes its rate of growth (b) and RNA polymerase occupancy (c). We have used the model to understand how cells regulate their cell cycle (d) and allocate energy (e).

To capture our varying level of knowledge about each pathway, we described each pathway using the most appropriate mathematical representation. For example, we represented transcription and translation as stochastic models, represented metabolism using FBA, and represented cell division with ODEs. We combined the submodels into a single model by mapping their inputs and outputs onto a common set of global variables that we formed by taking the union of the state variables of the individual submodels.

We developed a novel algorithm to simulate the combined model by co-simulating the submodels. The algorithm co-simulated the submodels by partitioning the copy number variables into separate pools for each submodel proportional to their anticipated consumption, iteratively integrating the submodels, updating the global variables by merging the pools associated with the submodels, and updating all other state variables. To help us analyze the model's simulation results, we also developed `WholeCellSimDB`, a database for organizing, storing, and sharing WC simulation results [158] and `WholecellViz`, a web-based software tool for visualizing high-dimensional WC simulation results in their biological context [161].

We calibrated the model by constructing a set of reduced models that focused on each pathway submodel, calibrating the individual submodels, and using the parameter values learn from calibrating the individual submodels as a starting point for calibrating the entire model [179].

We validated the model by constructing numerous reduced models that focused on individual submodels and groups of submodels, checking that the submodels and groups of submodels are consistent with our knowledge such as the Central Dogma, and checking that the submodels and groups of submodels are consistent with the experimental data that we used to build the model and additional independent experimental data (Figure 1.7b,c). In particular, we demonstrated that the model recapitulates the observed *M. genitalium* growth rate and predicts the essentiality of each gene with 80% accuracy.

In addition, we have used the model to demonstrate how WC models could be used to help design synthetic circuits [185] and we have used the model to demonstrate how WC models could help reposition antibiotics among distance bacteria [186].

Despite this progress, the model does not represent several important cell functions such as the maintenance of electrochemical gradients across the cell membrane, and the model mispredicts several important phenotypes such as the growth rates of many single-gene deletion strains. Furthermore, the model took over 10 person-years to construct because it was largely built by hand; the model is difficult to understand, reuse, and extend because it was described directly in terms of its numerical simulation rather than using a high-level format such as SBML; the model's simulation software is not reusable because it was built to simulate a single model; the model's simulation algorithm violates the arrow of time and is unscalable because it only partitions a portions of the state variables among the submodels.

1.6.3 Physiologically-centric top-down fine-grained models

In parallel, researchers such as Edda Klipp are taking a complementary top-down physiologically-centric approach to WC modeling to our genomically-centric bottom-up approach to WC modeling. In contrast to our approach which starts from annotated genomes, Edda Klipp and her colleagues are modeling entire cells by enumerating the major processes present in cells, developing submodels of each process, and combining the submodels into a single model.

1.6.4 Spatially-centric bottom-up fine-grained models

In parallel, researchers such as Elijah Roberts and Zaida Luthy-Schulten are taking another complementary spatially-centric approach to WC modeling [182][183][26]. This approach focuses on representing the spatial distribution and diffusion of each molecular species, and uses molecular dynamics simulation methods to predict their spatiotemporal dynamics. However, because it is computationally expensive to simulate diffusion on the scale of entire cells, this approach is currently limited to second-scale simulations.

1.6.5 Hybrid models

As introduced above, Zaida Luthey-Schulten and her collegues have begun to merge these fine-grained approaches to WC modeling by combining a diffusion model with an FBA model [184].

1.7 Bottlenecks to more comprehensive and predictive WC models

In the previous sections, we described how we and others are beginning to build WC models. Despite this progress, it is still challenging to build and simulate WC models. To help focus the community's efforts to accelerate WC modeling, here, we summarize the major remaining bottlenecks to WC modeling (Figure 1.8). These bottlenecks are based on our own experience and a community survey of the bottlenecks to biomodeling that we conducted in 2017 [16]. In the following sections, we suggest ways to overcome these bottlenecks.

Figure 1.8: Major bottlenecks to WC modeling and the major methods, tools, and resources needed to advance WC modeling.

1.7.1 Inadequate experimental methods and data repositories

In our opinion, one of the biggest bottlenecks to WC modeling is collecting and aggregating enough high-quality experimental data to build WC models. This is a significant bottleneck because WC models require extensive data, and because, as described in Section 1.3.1, we do not yet have sufficient methods for characterizing cells, sufficient tools for annotating the semantic meaning of experimental data, sufficient repositories for aggregating and integrating experimental data, and sufficient incentives for researchers to share their data.

New measurement methods, data repositories, and data aggregation tools are needed to overcome this bottleneck: (a) improved proteome-wide methods for measuring protein abundances would facilitate more accurate models of many pathways; (b) improved metabolome-wide methods for measuring metabolite concentrations would enable more accurate models of metabolism; (c) new single-cell measurement methods would facilitate more accurate models of the phenotypic variation of single cells; (d) a new central data repository that uses consistent representations,

identifiers, and units would accelerate data aggregation [187]; and (e) new tools for searching this repository would help researchers identify relevant data for WC modeling, including data from related organisms and environments.

1.7.2 Incomplete, inconsistent, scattered, and poorly annotated pathway models

As discussed in Section 1.5, the most promising strategy for building WC models is to combine multiple separate models. However, the lack of a complete set of compatible, well-annotated, and high-quality pathway models is a major bottleneck to WC modeling [80][81][188][24]. Here, we summarize the limitations of our pathway models.

Incomplete models

Despite decades of modeling research and detailed models of several pathways, we still do not have models of most pathways. For example, we do not have models of the numerous DNA repair mechanisms, the mechanisms responsible for RNA editing, or the role of chaperones in protein folding.

Poorly validated and unreliable models

Many of our existing pathway models are insufficiently validated and reliable to be effective components of WC models. Furthermore, few models are published with sufficient information about what data was used to validate the model, which simulation predictions were validated, and which simulation predictions are reliable for other researchers to know the limitations of a model and how to properly reuse it.

Inconsistent models

Furthermore, many of our existing pathway models are inconsistent. In particular, many of existing models are described with different assumptions, granularities, mathematical representations, identifiers, units, and formats.

Unpublished and scattered models

Unfortunately, our published models are scattered across a large number of resources, including model repositories such as BioModels, Simtk, supplementary materials, GitHub, and individual lab web pages, and many reported models are never published.

Incompletely annotated models

Many reported models are also not sufficiently well-annotated to combine them into WC models. For example, the biological semantic meaning of a model is often not annotated. This makes it difficult for other researchers to understand the meaning of each variable and equation which, in turn, makes it difficult for other researchers to merge models. The provenance of a model is also rarely annotated. This makes it difficult for other researchers to understand how a model was calibrated, recalibrate the model to represent a different organism and/or condition, and merge a model with models of other organisms and/or conditions. In addition, the assumptions of a model are also rarely annotated. Similarly, this makes it difficult for other researchers to understand how a model was developed, revise a model to represent other organisms and conditions, and merge models from different organisms and conditions.

1.7.3 Inadequate software tools for WC modeling

As described in Section 1.4, a wide range of tools have been developed for modeling individual pathways. However, few of these tools support all of the features needed for WC modeling. In particular, few of these tools support the scale required for WC modeling, few of these tools support composite, multi-algorithmic modeling, few of these

tools support collaboration, and these tools do not support all of the metadata needed to understand models and their provenance.

1.7.4 Inadequate model formats

As described in Section 1.4.2, several formats have been developed to describe cell models. However, the lack of a format that supports all of the features needed for WC modeling is a major bottleneck. In particular, no existing format can represent (a) the combinatorial complexity of pathways such as transcription elongation which involve billions of sequence-based reactions; (b) the multiple scales that must be represented by WC models such as the sequence of each protein, the subunit composition of each complex, and the DNA binding of each complex; and (c) multi-algorithmic models that are composed of multiple mathematically-distinct submodels [106].

1.7.5 Lack of coordination among the cell modeling community

Another major bottleneck to WC modeling is the lack of coordination among the cell modeling community. Currently, the lack of coordination leads modelers to build competing models of the same pathways and describe models with inconsistent identifiers and formats.

1.8 Technologies needed to advance WC modeling

In the previous section, we outlined the major remaining bottlenecks to WC modeling. To overcome these bottlenecks, we must develop a wide range of computational and experimental technologies. Here, we describe the most critically needed technologies to advance WC modeling. In the following sections, we highlight our and others' ongoing efforts to develop these technologies.

1.8.1 Experimental methods for characterizing cells

While substantial data about cellular populations already exists, additional data would enable better WC models. In particular, we should develop new experimental methods for quantitating the dynamics and single-cell variation of each metabolite and protein. Additionally, we should develop methods for measuring kinetic parameters at the interactome scale, as well as methods for measuring cellular phenotypes across multiple genetic and environmental conditions.

1.8.2 Tools for aggregating, standardizing, and integrating heterogeneous data

As described in Section 1.4.1-1.4.1, extensive data is now available for WC modeling. However, this data spans a wide range of data types, organisms, and environments; the data is often not annotated and normalized; it is scattered across many repositories and publications and it is described using inconsistent identifiers and units. To make this data more usable for modeling, we must develop tools for aggregating data from multiple sources; merging data from multiple specimens, environmental conditions, and experimental procedures; standardizing data to common identifiers and units; identifying the most relevant data for a model; and averaging across multiple imprecise and noisy observations.

1.8.3 Tools for scalably designing models from large datasets

To scalably build WC models, we must develop tools for defining the interfaces among pathway submodels, collaboratively designing composite, multi-algorithmic models directly from large datasets, automatically identifying inconsistencies and gaps in dynamical models, recording how data and assumptions are used to build models, and encoding models in a rule-based format. As described in Section 1.4.2-1.4.2, several tools support each of these features.

To accelerate WC modeling, we should develop a single tool that supports all of these functions at the scale required for WC modeling.

1.8.4 Rule-based format for representing models

Several formats can represent individual biological processes. However, no existing format is well-suited to representing the scale or mathematical diversity required for WC modeling [106][189]. To succinctly represent WC models, we should develop a rule-based format that can (a) represent models in terms of high-level biological constructs such as DNA, RNA, and proteins; (b) represent each molecular species at multiple levels of granularity (for example, as a single species, as a set of sites, and as a sequence); (c) represent all of the combinatorial complexity of molecular biology including the complexity of interactions among protein sites, as well as the complexity of protein-metabolite, protein-DNA, and protein-RNA interactions and the complexity of template-based polymerization reactions such as the combinatorial number of RNA than arise from the interaction of RNA splicing, editing, and mutations; (d) represent composite, multi-algorithmic models; (e) represent the biological semantic meaning of each species and interaction using database-independent formats such as InChI [178] and DNA, RNA, and protein sequences; and (f) represent model provenance including the data and assumptions used to build models.

1.8.5 Scalable network-free, multi-algorithmic simulator

To simultaneously represent well-characterized pathways with fine detail and coarsely represent poorly-characterized pathways, WC modeling requires a multi-algorithmic simulator that can scalably co-simulate mathematically-dissimilar submodels that are described using rule patterns. However, no existing simulator supports network-free, multi-algorithmic, and parallel simulation. To scalably simulate WC models, we should develop a parallel, network-free, multi-algorithmic simulator [190]. At a minimum, the simulator should support FBA, ODE integration, and stochastic simulation.

1.8.6 Scalable tools for calibrating models

As discussed in Section 1.4.2, several tools are available for calibrating small single-algorithm models. However, these tools are not well-suited to calibrating large multi-algorithmic models. To calibrate WC models, we must develop new methods and software tools for scalably calibrating rule-based multi-algorithmic models. We and others have begun to explore using reduced models to efficiently calibrate WC models [179]. However, further work is needed to formalize these methods, including developing automated methods for reducing WC models.

1.8.7 Scalable tools for verifying models

To fulfill our vision of using WC models to drive medicine and bioengineering, it will be critical for modelers to rigorously verify that WC models function as intended. As discussed in Section 1.4.2, researchers are beginning to adapt tools from computer science and software engineering to verify cell models. However, none of the existing or planned tools support rule-based, multi-algorithmic models. To help modelers verify WC models, we must adapt formal verification and/or unit testing for WC modeling. Furthermore, to help researchers quickly verify models, these tools should help researchers verify entire WC models, as well as help researchers verify reduced models and individual submodels.

1.8.8 Additional tools that would help accelerate WC modeling

In addition to these essential tools, we believe that WC modeling would also be accelerated by additional tools for annotating and imputing data, additional tools for sharing WC models and simulation results, additional tools for visualizing simulation results, and community standards for designing, annotating, and verifying WC models.

- **Tools and standards for annotating data.** To make our experimental more useful for modeling, we should develop software tools that help researchers annotate their data and encourage experimentalists to use these tools to annotate their data.
- **Bioinformatics prediction tools.** While existing bioinformatics tools can predict many properties of metabolites, DNA, RNA, and proteins, additional tools are needed to accurately predict the molecular effects of insertions, deletions, and structural variants. Such tools would help WC models design microbial genomes and predict the phenotypes of individual patients.
- **Repositories for WC models.** To help researchers share whole-cell models, BioModels and other model repositories should be extended to support WC models. In addition, these repositories should be extended to support provenance metadata, validation metadata, simulation experiments, and simulation results.
- **Version control system for WC models.** To help researchers collaboratively develop WC models, we should develop a version control system for tracking the changes to WC models contributed by individual collaborators and merging WC model components developed by collaborators. This system could be developed by combining Git [149] with a custom program for differencing WC models.
- **Simulation format.** SED-ML and SESSL can represent simulations of models that are encoded in XML-based formats such as SBML and Java-based formats such as ML-Rules. However, neither is well-suited to representing simulations of models that are encoded in other formats such as BioNetGen. To accelerate WC modeling, we should extend SED-ML to support non-XML-based models or extend SESSL to support other programming languages such as Python and C++.
- **Database for organizing simulation results.** We and others have begun to develop tools for organizing simulation results. However, these tools have limited functionality. To help researchers analyze WC simulation results, we must develop an improved database for simulation results that helps researchers quickly search simulation results for specific features and quickly retrieve specific slices of large simulation results datasets. This database should be implemented using a distributed database and/or data processing technologies such as Apache Spark.
- **Tools for visualizing simulation results.** We and others have also begun to develop tools for visualizing high-dimensional simulation results. However, these tools have limited functionality, they are not easily extensible, and they struggle to handle large datasets. To help researchers analyze WC models to gain new biological insights, we must develop a new tool for visually exploring and analyzing WC simulation results. To enable researchers to incorporate new visual layouts, this tool should support a standard visualization grammar such as Vega [180]. Furthermore, to handle terabyte-scale simulation result datasets, this tools should be implemented using a high-performance visualization toolkit such as VTK [191].
- **Community standards.** To facilitate collaboration, we should develop guidelines for designing WC models, standards for annotating and verifying WC models, and a protocol for merging WC model components. The model design guidelines should describe the preferred granularity of WC model components and the preferred interfaces among WC model components. The standards for annotating and verifying WC models should describe the minimum acceptable semantic and provenance metadata for WC models. The protocol for merging WC model components should describe how to incorporate a new component into a WC model, how to test the new component and the merged model, and how to either accept the new component or reject the candidate component if it cannot be verified or is not properly annotated.

1.9 A plan for achieving comprehensive WC models as a community

In the previous sections, we described the potential of WC models to advance medicine and bioengineering, summarized the major bottlenecks to WC modeling, and outlined several technological solutions to these bottlenecks. To maximize our efforts to achieve WC models, we believe that we should begin to develop a plan for achieving WC models. Here, we propose a three-phase plan to achieve the first comprehensive WC model (Figure 1.9). The plan focuses on developing a WC model of H1-hESCs because we believe that the community should initially focus on a single cell line and because H1-hESCs are relatively easy to culture, well-characterized, karyotypically and phenotypically "normal", genomically stable and relevant to a wide range of basic science, medicine, and bioengineering.

Although the plan focuses on a single cell line, the methods and tools developed under the plan would be applicable to any organism, and the H1-hESC model could be contextualized to represent other cell lines, cell types, and individuals.

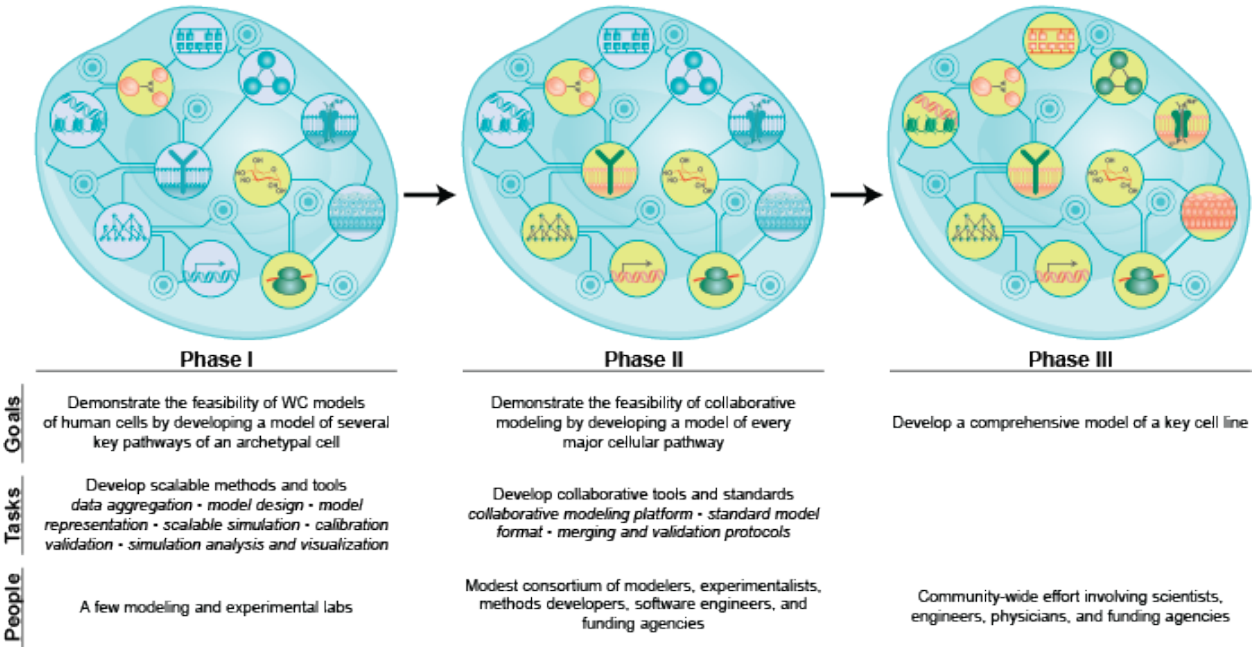


Figure 1.9: The first WC models can be achieved in three phases: (1) demonstrating the feasibility of WC models by developing scalable modeling tools and using them to model several core processes, (2) demonstrating the feasibility of collaborative modeling by developing a collaborative modeling platform and using it to model additional processes, and (3) developing a comprehensive model as a community.

1.9.1 Phase I: Piloting the core technologies and concepts of WC modeling

Phase I should demonstrate the feasibility of WC models by developing the core technologies needed for WC modeling, and using these tools to build a model of a few critical pathways of H1-hESCs. First, we should develop tools for aggregating the data needed for WC modeling, tools for designing models directly from data, a rule-based format for describing models, tools for quickly simulating multi-algorithmic models, tools for efficiently calibrating and validating high-dimensional models, and tools for visualizing and analyzing high-dimensional simulation results. Second, a small group of researchers should use these tools and public data to build a model of the core pathways of H1-hESCs including several key signal transduction pathways, metabolism, DNA replication, transcription, translation, and RNA and protein degradation. Phase I should also begin to form a WC modeling community by organizing meetings and courses, developing WC modeling training materials, and discussing potential WC modeling standards.

1.9.2 Phase II: Piloting collaborative WC modeling

Phase II should focus on demonstrating the feasibility of collaborative WC modeling by developing collaborative modeling tools, and using them to expand the H1-hESc model begun in Phase I. First, we should combine the technologies developed in Phase I into a collaborative web-based WC modeling platform to enable multiple experts to build models together. Second, the community should develop standards for describing, validating, and merging submodels. Third, a modest consortium of modelers and experimentalists should expand the H1-hESc model developed in Phase I by partitioning H1-hESCs into distinct pathways, outlining the interfaces among these pathways, and tasking individual researchers with modeling additional pathways such as cell cycle regulation, DNA repair, and cell division. Fourth, we should extensively validate the combined model. Phase II should also continue to develop the fundamental tech-

nologies needed for WC modeling and continue to build a WC community by organizing meetings, courses, and other community events.

1.9.3 Phase III: Community modeling and model validation

Phase III should produce the first comprehensive WC model. First, we should assemble a large community of modelers and experimentalists and train them to use the platform developed in Phases I and II. Second, individual researchers should volunteer to model individual pathways and merge them into the global H1-hESc model. Third, we should continue to validate the combined model. Fourth, researchers should use the model to generate testable hypotheses to discover new biology, new disease mechanisms, and new drug targets. Fifth, we should also begin to develop methods for contextualizing the H1-hESC model to represent other cell lines, cell types, and individuals. In addition, the community should continue to develop the core technologies and standards needed for WC modeling, continue to refine the partitioning of cells into pathways, continue to refine the interfaces among the pathways, continue to organize meetings and course, and continue to develop WC modeling tutorials.

1.10 Ongoing efforts to advance WC modeling

In the previous section, we proposed a plan for achieving the first comprehensive WC model as a community. Although we do not yet have an organized WC modeling community, we and others are beginning to pilot WC models and the technology needed to achieve them. Here, we summarize the ongoing efforts to pioneer WC modeling.

1.10.1 Genomically-centric models

Currently, there are three genomically-centric WC models in development of *Mycoplasma pneumoniae*, *E. coli*, and H1-hESCs.

Mycoplasma pneumoniae

To explore how to build more comprehensive and more accurate models, we are working with Drs. Maria Lluch-Senar and Luis Serrano to develop a comprehensive model that represents all of the characterized genes of the bacterium *M. pneumoniae*.

M. pneumoniae is a small gram-positive bacterium that has one of the smallest genomes among all known freely-living organisms and that is one of the most common causes of walking pneumonia. *M. pneumoniae* is tractable to WC modeling because it has a small genome and a small mass; because Dr. Lluch-Senar, Dr. Serrano, and others have extensively characterized *M. pneumoniae*; and because most of its genome is functionally annotated. However, *M. pneumoniae* can be difficult to characterize because it grows slowly and because there are few experimental methods for manipulating *M. pneumoniae*, some aspects of *M. pneumoniae* are challenging to model because there is no known defined growth media for *M. pneumoniae*, and the *M. pneumoniae* research community is small. Because *M. pneumoniae* has such a small genome, *M. pneumoniae* is frequently used to study the minimal requirements of cellular life, explore the origins of cellular life, and pilot genome-scale synthetic biology methods such as whole-genome synthesis and genome transplantation. *M. pneumoniae* is also frequently studied to gain insights into the pathophysiology of walking pneumonia.

The model will be based both on genomic, transcriptomic, and proteomic data about *M. pneumoniae* collected by Drs. Lluch-Senar and Serrano, as well as a broad range of biochemical and single-cell data about related species aggregated from public databases and publications. In addition to using the model to demonstrate the feasibility of more comprehensive models and drive the development of WC modeling methods, we hope to use this model to engineer a fast-growing, efficient chassis for future bioengineering projects.

Escherichia coli

To explore how to model more complex bacteria, Prof. Markus Covert and his group at Stanford University are modeling the model gram-negative bacterium *E. coli*. The project focuses on *E. coli* because *E. coli* is the best-characterized bacterium and because there are a wide variety of experimental methods for manipulating and characterizing *E. coli*. Because *E. coli* is substantially more complex than reduced bacteria such as *M. genitalium* and *M. pneumoniae*, initially, this project will focus on modeling core pathways such as metabolism, RNA and protein synthesis and degradation, DNA replication, and cell division. The model will be based primarily on data observed for *E. coli* aggregated from a wide range of sources. Prof. Covert and his group are using this model to demonstrate the feasibility of more comprehensive WC models, as well as gain novel insights into the pathogenesis of *E. coli*.

H1 human embryonic stem cells (hESCs)

To explore how to model eukaryotic cells, we are also beginning to model H1-hESCs. ESCs are pluripotent cells derived from the inner cell mass of a blastocyst at 4-5 days post-fertilization that can generate all three primary germ layers. We have chosen to pilot human WC models with hESCs because they are karyotypically and phenotypically "normal"; they are genomically stable; they can self-renew; and they are relevant to a wide range of basic science, medicine, and tissue engineering.

Furthermore, we have chosen to focus on H1-hESCs because they can be cultured with feeder-free media and because they have been extensively characterized. For example, H1 was one of the three cell lines most deeply characterized by the ENCODE project [192]. In addition, H1 was one of the first five hESC lines [193], H1 was the first cell line to approved under NIH's Guidelines for Stem Cell Research, and, as of 2010, H1 was studied in 30% of all hESC studies [194].

Because human cells are vastly more complex than bacteria, we are beginning by modeling the core pathways responsible for stem cell growth, maintenance, and self-renewal, including metabolism, transcription, translation, RNA and protein degradation, signal transduction, and cell cycle regulation. This model will also be based both on genomic, transcriptomic, and proteomic data about H1-hESCs aggregated from publications, as well as biochemical and single-cell data about related cell lines aggregated from several databases. In addition to using the model to demonstrate the feasibility of human WC models and driving the development of WC modeling methods, we hope to use the model to gain new insights into the biochemical mechanisms responsible for regulating the rate of stem cell growth.

1.10.2 Physiologically-centric, spatially-centric, and hybrid models

As described in Section 1.6.3-1.6.4, Klipp, Roberts, and others are also developing physiogically-centric models of *S. cerevisiae*, spatially-centric models of *E. coli*, and hybrid spatially-centric/FBA models of *E. coli*.

1.10.3 Technology development

Currently, we are developing three technologies for aggregating the data needed for WC modeling; concisely representing multi-algorithmic WC models using rules; and simulating rule-based, multi-algorithmic models.

Data aggregation

WC modeling requires a wide range of data. Unfortunately, as described in Section 1.7.1, aggregating this data is a major bottleneck to WC modeling because this data is scattered across a wide range of databases and publications. To help modelers obtain the data needed for WC modeling, we are developing a methodology for systematically and scalably identifying, aggregating, standardizing, and integrating the data needed for WC modeling, and we are developing a software program called `Datanator` which implements this methodology. The methodology consists of eight steps:

1. **Aggregation.** Modelers should retrieve a wide range of data from a wide range of sources such as metabolite concentrations from ECMDB, RNA concentrations from ArrayExpress, protein concentrations from PaxDb, reaction stoichiometries from KEGG, and kinetic parameters from SABIO-RK. Where possible, this should be implemented using downloads and web services. Where this is not possible, this should be implemented by scraping web pages and manually curating individual publications. Importantly, modelers should also record the provenance of each downloaded dataset.
2. **Parsing.** Modelers should parse each data source into an easily manipulatable data structure.
3. **Standardization.** Modelers should standardize the identifiers, metadata, and units of their data. The metadata should include the species and environmental conditions that were observed, the method used to measure the data, the investigators who collected the data, and the citation of the original data. We recommend using absolute identifiers such as InChI to describe all possible measurements, using ontologies such as the Measurement Method Ontology (MMO) to describe metadata consistently, and using SI units.
4. **Integration.** Modelers should merge the aggregated data into a single dataset. We recommend that modelers use relational databases such as SQLite to organize their data and make their data searchable.
5. **Filtering.** For each model parameter that modelers would like to constrain with experimental data, modelers should identify the most relevant observations within their dataset by scoring the similarity between the physical properties of the parameter and each observation, the species that they want to model and the observed species, and the environmental condition that they want to model and the observed conditions.
6. **Reduction.** For each model parameter, modelers should reduce the relevant data to constraints on the value of the parameter by calculating the mean and standard deviation of the relevant data, weighted by its similarity to the physical property, species, and environmental condition that the modeler wants to model.
7. **Review.** Because it is difficult to fully describe the context of experimental measurements and, therefore, difficult to automatically identify relevant data for a model, modelers should manually review the least relevant data to potentially select alternative observations or integrate more relevant data from other sources.
8. **Storage.** Lastly, modelers should store the reduced data and its provenance in a data structure that is conducive to building models. We recommend organizing this data using a specialized PGDB such as `WholeCellKB`.

We have already developed a common platform which implements this methodology, and data aggregation modules for the most critical data types for WC modeling. Going forward, we plan to develop additional modules for aggregating data from a wider range of sources and we plan to develop a user-friendly web-based interface for using Datanator. In addition, we hope to explore additional data aggregation methods such as natural language processing and crowdsourcing.

Model representation

As described in Section 1.7.4, no existing format is well-suited to representing composite, multi-algorithmic WC models. In particular, there is no format which is well-suited to describing all of the combinatorial complexity of cellular biochemistry, representing composite, multi-algorithmic models, and representing the semantic biological meaning and provenance of models.

To accelerate WC modeling, we are developing, `wc_rules`, a more abstract rule-based format for describing WC models. The format will be able to represent each molecular species at multiple levels of granularity (for example, as a single species, as a set of sites, and as a sequence); represent all of the combinatorial complexity of each molecular species and interaction; represent composite, multi-algorithmic models; represent the data, assumptions, and design decisions used to build models; and represent the semantic biological meaning of models. We are developing tools to export models described with `wc_rules` to BioNetGen and SBML, as well as a simulator for simulating models described with `wc_rules`.

Simulation of genomically-centric models

As described in Section 1.7.3, no existing simulator is well-suited to simulating computationally-expensive, high-dimensional, rule-based, multi-algorithmic WC models. In particular, there are only a few parallel simulators, only a few rule-based simulators, only a couple of multi-algorithmic simulators, and no simulator which supports all of these technologies.

To accelerate WC modeling, we are beginning to use the Viatra [195] graph transformation engine and the ROSS [117] PDES engine to develop `wc_sim`, a parallel, network-free, multi-algorithmic simulator that can simulate models described in `wc_rules` [190]. Simulations will consist of six steps:

1. **Compile models to a low-level format.** We will compile models described with `wc_rules` to a low-level format which can be interpreted by the simulation engine.
2. **Merge mathematically compatible submodels.** We will analytically merge all mathematically-compatible submodels, producing a model which is composed of at most one FBA, one ODE, and one SSA submodel.
3. **Partition submodels into cliques.** To use multiple machines to simulate models, we will partition models into cliques that can be simulated on separate machines with minimal communication to synchronize the cliques.
4. **Assign cliques to core.** We will use ROSS to assign each clique to a separate machine and use event messages and rollback to synchronize their states.
5. **Co-simulate mathematically-distinct submodels.** We will co-simulate the FBA, ODE, and SSA submodels by periodically calculating the fluxes predicted by FBA and ODE models and interpolating them with each SSA event.
6. **Rule-based simulation of SSA cliques.** We will use Viatra to represent each species and reaction pattern as a graph and iteratively select reactions, fire reactions, and update the species graphs. To efficiently simulate both sparsely and densely concentrated species, we will use a hybrid population/particle representation in which each species graph will represent a species and its copy number, and we will periodically merge identical graphs that represent the same species.

1.11 Resources for learning about WC modeling

To learn more about WC modeling, we recommend attending a WC modeling summer school or participating in the WC modeling forum. Below are brief descriptions of these resources.

1.11.1 Summer schools

We and others organize annual WC modeling summer schools [106][196][197] for graduate students and postdoctoral scholars. The schools teach the fundamental principles of WC modeling through brief lectures and hands-on exercises. The schools also provide opportunities to network with other WC researchers. Please see http://wholecell.org for information about upcoming schools.

1.11.2 Online forum

The WC modeling forum is an online platform which enables researchers to initiate and participate in discussions about WC modeling.

1.12 Outlook

Despite several challenges, we believe that WC models are rapidly becoming feasible thanks to ongoing advances in experimental and computational technology. In particular, in Section 1.9, we have proposed a three-stage plan to achieve comprehensive WC models as a community. The cornerstones of this plan include developing practical solutions to the key bottlenecks; forming a collaborative interdisciplinary community; and adhering to common interfaces, formats, identifiers, and protocols. We have already developed tools for organizing the data needed for WC modeling, organizing WC simulation results, and visualizing WC simulation results, and we have begun to organize a WC modeling community. Currently, we are developing tools for aggregating the data needed for WC modeling, concisely describing WC models, and scalably simulating WC models, and we are continuing to organize WC modeling meetings. We are eager to advance WC modeling, and hope you will join us!

CHAPTER 2

Foundational concepts and skills for computational biology

2.1 Typing

We believe that it is important for computational scientists to be able to type efficiently. Below are several excellent online resources for learning and practicing how to type efficiently.

- typing.com
- TypingCAT
- TypingClub

2.2 Software engineering

2.2.1 An introduction to Python

The goal of this tutorial is to introduce the fundamentals of Python. This tutorial assumes familiarity with at least one other object-oriented programming language such as Java, MATLAB, or Perl.

For a more elementary introduction to Python for individuals with little or no programming experience, we recommend the following five websites. The first four of the websites provide interactive coding exercises.

- Codecademy: Learn Python
- DataCamp: Intro to Python for Data Science
- Dataquest: Python Programming
- Rosalind: Python problems
- Learn Python the hard way

Key concepts

This tutorial will teach you how to use the following Python elements

- Core language features
  - Core builtin data types
  - Variables
  - Operators
  - Boolean statements
  - Loops
  - Context managers
  - Functions
  - Classes
  - Modules
  - Decorators
- Copying variables
- String formatting
- Printing to standard out
- Numerical computing with `NumPy`
- Plotting graphs with `matplotlib`
- Reading and writing to/from text, csv, and Excel files
- Exceptions and warnings

Installing Python and Python development tools

To get started using Python, we recommend installing Python 3, the pip package manager, the Sublime text editor, and the ipython interactive shell. See Section 2.2.14 for more information.

Data types

Scalars

Python provides several classes to represent Booleans (`bool`), integers (`int`), floats (`float`), and strings (`str`). The following code instantiates Python Booleans, integers, floats, and strings:

```
True        # a instance of `bool`
False       # another instance of `bool`
1           # an instance of `int`
1.0         # an instance of `float`
'a string' # an instance of `str`
```

Lists, tuples, sets, and dictionaries

Python also provides several classes to represent lists (`list` and `tuple`), sets (`set`), and hashes (`dict`). The following code creates Python lists, sets, and hashes:

```
[1, 2, 3]         # a `list`
(1, 2, 3)         # a `tuple`
set([1, 2, 3])    # a `set`
{'a': 1, 'b': 2} # a `dict`
```

Lists and tuples can both represent ordered series of values. The major difference between lists and tuples is that lists are mutable (elements can be added and removed), whereas tuples are immutable (elements cannot be added or removed). In addition, tuples can be unpacked as illustrated below.

The individual elements of a list, tuple, or dictionary can be accessed as illustrated below:

```
list_var = [1, 2, 3]
list_var[0] # get the first element
list_var[-1] # get the last element
list_var[1:3] # get a subset of the list containing second and third elements

tuple_var = (1, 2, 3)
tuple_var[0] # get the first element
tuple_var[-1] # get the last element
tuple_var[1:3] # get a subset of the list containing second and third elements
a, b, c = tuple_var # unpack the tuple into the values of a, b, and c

dict_var = {'a': 1, 'b': 2}
dict_var['a'] # get the value of the 'a' key
```

Variables

The = operator can be used to create or set the value of a variable as illustrated below:

```
my_var = [1, 2, 3]
my_var = 'a' # reassign my_var to a string
```

Note, Python variables do not have to be declared and are not typed.

Boolean statements

As illustrated below, Boolean statements can be created using a variety of comparison operators (==, >=, <=, etc.) and binary operators (`and`, `or`, `not`):

```
x and y
x or y
x >= 1 and x <= 2
x == 1.0
```

If statements

If/else statements can be implemented as illustrated below:

```
if {statement}:
    ...
else:
    ...
```

The `elif` directive can be used to achieve a similar behavior to the switch directives of other languages:

```
if {statement_1}:
    ...
elif {statement_2}:
    ...
else:
    ...
```

Loops

Python provides a for loop which can be used to iterate over ranges of values, lists, tuples, sets, dictionaries, and matrices as illustrated below. Note, the code that should be executed with the for loop must be nested underneath the loop definition and indented.:

```
# iterate from 0 .. iter_max
for iter in range(iter_max):
    ...

# iterate over the values of a list, tuple, set, or matrix
list_var = [...]
for value in list_var:
    ...

# iterate over the keys in a dictionary
dict_var = {...}

for key in dict_var:
    ...

for key in dict_var.keys():
    ...

# iterate over the values in a dictionary
for value in dict_var.values():
    ...

# use tuple unpacking to iterate over the keys and values in a dictionary
for key, value in dict_var.items():
    ...
```

While loops can be implemented as illustrated below:

```
while {statement}:
    ...
```

The `continue` directive can be used to advance to the next iteration of a loop and the `break` directive can be used to exit a loop.

Functions

Python functions can be defined and evaluated as illustrated below:

```
# define a function with one required and one optional argument
def my_func(required_arg_1, optional_arg_2=default_value):
    ...
    return return_val # return the value return_val

return_val_1 = my_func(value_1)
return_val_2 = my_func(value_1, arg_2=value_2)
```

Inline *lambda* functions can also be defined as illustrated below:

```
my_func = lambda required_arg_1: ...
```

Classes

Python classes can be defined and objects can be instantiated as illustrated below. Note, `self` is the name typically used to refer to the class instance.:

```
# create a class with one attribute
class MyClass(object):

    # the method called when an instance of the class is constructed
    def __init__(self, required_arg_1, optional_arg_2=default_value):
        self.attribute_1 = ... # define the attributes of the class
        ...

    def my_method(self, required_arg_1, optional_arg_2=default_value):
        return self.attribute_1 # access the attribute of the class

my_instance = MyClass(value_1) # create an instance of the class
my_instance.attribute_1 # get the value of attribute_1
my_instance.attribute_1 = value_2 # set the value of attribute_1
value_4 = my_instance.my_method(value_3) # evaluate the method of the class
```

Note, all Python class attributes are public. The _ prefix is often used to indicate attributes that should be treated as protected and the __ prefix is often used to indicate attributes that should be treated as private.

Subclasses can be created as illustrated below:

```
class MySecondClass(MyClass):

    def __init__(self, required_arg_1):
        super(MySecondClass, self).__init__(required_arg_1) # call the constructor␣
↪for the parent class
        ...
```

Modules

Python programs can be organized into multiple *modules* by splitting the code into multiple directories and/or files. In order for Python to recognize a directory as a module, the directory must contain a file with the name `__init__.py`. This file can be blank. For example, the following file structure will create two modules, each with three sub-modules:

```
/path/to/project/
    module_1/
        __init__.py
        sub_module_1a.py
        sub_module_1b.py
        sub_module_1c.py
    module_2/
        __init__.py
        sub_module_2a.py
        sub_module_2b.py
        sub_module_2c.py
```

The `import` directive can be used to access code from other modules. For example, the following code fragment could be used within `sub_module_2a.py` to access code from the other modules

import module_1.sub_module_1a module_1.sub_module_1a.my_func(...) module_1.sub_module_1a.MyClass(...)

from module_1 import sub_module_1b sub_module_1b.my_func(...) sub_module_1b.MyClass(...)

from module_1 import sub_module_1b as s1c s1c.my_func(...) s1c.MyClass(...)

from . import sub_module_2b sub_module_2b.my_func(...) sub_module_2b.MyClass(...)

String formatting

Strings can be formatted using the `str.format` method as illustrated below. This method can be used to substitute variables into strings using the `{}` placeholder:

```
'{} {} {}'.format('first value', 2, 3.0)
```

Printing to the command line

The `print` method can be used to write to standard output:

```
print('Message')
```

Reading and writing to/from files with `csv` and `pyexcel`

The follow example illustrates how to read and write text files:

```
# write content to a file
file_handle = open('filename.txt', 'w')
file_handle.write(content)
file_handle.close()

# write content to a file using a context manager
with open('filename.txt', 'w') as file_handle:
    file_handle.write(content)

# read content from a file using a context manager
with open('filename.txt', 'r') as file_handle:
    content = file_handle.read()
```

The follow example illustrates how to read and write csv files:

```
import csv

# write a list of lists to a csv file
with open('eggs.csv', 'w') as csvfile:
    csv_writer = csv.writer(csvfile)
    for row in rows:
        csv_writer.writerow(row)

# write a list of dictionaries to a csv file with row headings
with open('eggs.csv', 'r') as csvfile:
    csv_writer = csv.DictReader(csvfile, fieldnames)
    for row in rows:
        csv_writer.writerow(row)

# read a csv file into a list of lists
with open('eggs.csv', 'r') as csvfile:
    rows = csv.reader(csvfile)

# read a csv file with row headings into a list of dictionaries
with open('eggs.csv', 'r') as csvfile:
    rows = csv.DictReader(csvfile)
```

The following example illustrates how to reading and write Excel files using the `pxexcel` package:

```
import pxexcel

book = pxexcel.get_book(file_name="example.xlsx")
book.save_as("another_file.xlsx")
```

Warnings and exceptions

Warnings can be issued and suppressed as illustrated below:

```
import warnings
warnings.warn('Warning message')

warnings.simplefilter("ignore", warnings.UserWarning) # ignore a class of warnings
```

Custom warning categories can be created and used as illustrated below:

```
class MyWarning(warnings.UserWarning):
    ...
warnings.warn('Message', MyWarning)
```

Exceptions can be issued as illustrated below:

```
raise Exception('Message')
```

Exceptions can be handled as illustrated below:

```
try:
    ... # code which raises an exception
except:
    ... # code to execute if the try block raises an exception

try:
```

(continues on next page)

(continued from previous page)

```
    ... # code which raises an exception
except Exception as exception:
    ... # code to execute if the try block raises an exception and the exception is␣
↪an instance of Exception
```

Custom exception classes can be defined and raised as illustrated below:

```
class MyException(Exception):
    ...

raise MyException(...)
```

Other Python languages features

Python provides a variety of additional powerful language features

- Context managers: context managers can be used to automatically run code at the beginning and end of a nested below
- Copying: the `copy.copy` and `copy.deepcopy` methods can be used to make copies of variables
- Customizable operators: the methods executed by operators such as ==, >=, and <= can be customized by overriding the `__eq__`, `__geq__`, and `__leq__` methods
- Decorators: decorators can be used to wrap the execution of a method. Examples of decorators include `@classmethod`
- Getters and setters: Getters and setters can be implemented by defining methods and decorating them with the `@property` and `@property.setter` decorators

Exercises

- Write a function which computes the volume of a spherical cell
- Write a function which uses if statements to return the type of a codon (start, stop, other)
- Write a class which represents RNA, with an attribute that stores the sequence of each transcript and a method which uses a dictionary to compute the amino acid sequence of the protein coded by the transcript
- Import the `csv` package and use it to read a comma-separated file with a header row into a list of dictionaries
- Use the `print` and `format` methods to write *Hello {your name}!* to standard out

See intro_to_wc_modeling/concepts_skills/software_engineering/python_introduction.py for solutions to these exercises.

2.2.2 Numerical computing with `NumPy`

`NumPy` is the most popular numerical computing package for Python. The following examples illustrate the basic functionality of `NumPy`.

Array construction

```
import numpy
arr = numpy.array([[1, 2, 3], [4, 5, 6]]) # a 2x3 matrix with elements equal to 1..6
arr = numpy.empty((2, 2)) # a 2x2 matrix with each element equal to a randomly␣
↪selected number
arr = numpy.zeros((2, 2)) # a 2x2 matrix with each element equal to 0.0
arr = numpy.ones((3, 4)) # a 2x3 matrix with each element equal to 1.0
arr = numpy.full((2, 3), 2.0) # a 2x3 matrix with each element equal to 2.0
```

Concatenation

```
arr1 = numpy.array([...])
arr2 = numpy.array([...])
numpy.concatenate(arr1, arr2)
```

Query the shape of an array

```
arr = numpy.array([...])
arr.shape()
```

Reshaping

```
arr = numpy.array([...])
arr.reshape((n_row, n_col, ...))
```

Selection and slicing

```
arr = numpy.array([[1, 2, 3], [4, 5, 6]])
arr[1, 2] # get the value at second row and third column
arr[0, :] # get the first row
arr[:, 0] # get the first column
```

Transposition

```
arr = numpy.array([...])
arr_trans = arr.transpose()
```

Algebra

```
arr1 = numpy.array([...])
arr2 = numpy.array([...])

arr1 + 2 # element-wise addition
arr1 - 2 # element-wise subtraction
arr1 + arr2 # matrix addition
arr1 - arr2 # matrix subtraction
```

(continues on next page)

(continued from previous page)

```
2. * arr # scalar multiplication
arr1 * arr2 # element-wise multiplication
arr1.dot(arr2) # matrix multiplication

arr ** 2. # element-wise exponentiation
```

Trigonometry

```
arr = numpy.array([...])

numpy.sin(arr) # element-wise sine
numpy.cos(arr) # element-wise cosine
numpy.tan(arr) # element-wise tangent
numpy.asin(arr) # element-wise inverse sin
numpy.acos(arr) # element-wise inverse cosign
numpy.atan(arr) # element-wise inverse tangent
```

Other mathematical functions

```
numpy.sqrt(arr) # element-wise square root
numpy.ceil() # element-wise ceiling
numpy.floor() # element-wise floor
numpy.round() # element-wise round
```

Data reduction

```
arr = numpy.array([...])

arr.all() # determine if all of the values are logically equivalent to `True`
arr.any() # determine if any of the values are logically equivalent to `True`
arr.sum() # sum
arr.mean() # mean
arr.std() # standard deviation
arr.var() # variance
arr.min() # minimum
arr.max() # maximum
```

Random number generation

```
# set the state of the random number generator to reproducibly generate random values
numpy.random.seed(1)

# select a float, randomly between 0 and 1
numpy.random.rand()

# select a random integer between 0 and 10
numpy.random.randint(10)

# select a random integer according to a Poisson distribution with :math:`\lambda = 2`
numpy.random.poisson(2.)
```

NaN and infinity

```
arr = numpy.array([...])

numpy.nan
numpy.isnan(arr)

numpy.inf
numpy.isinf(arr)
numpy.isfinite(arr)
```

Exercises

- Concatenate two 3x1 arrays of zeros and ones, and get its shape
- Select the first column of a random 2x3 array
- Transpose a random 2x3 array into a 3x2 array
- Reshape a random 2x3 array into a 3x2 array
- Create a random 2x3 array and round it
- Create a random 100x1 array of Poisson-distributed values with lambda = 10 and calculate its min, max, mean, and standard deviation
- Calculate the element-wise multiplication of two random arrays of size 3x3
- Calculate the matrix multiplication of two random arrays of size 2x3 and 3x4
- Check that infinity is greater than 10^{10}

See intro_to_wc_modeling/concepts_skills/software_engineering/numpy_exercises.py for solutions to these exercises.

NumPy introduction for MATLAB users

The NumPy documentation contains a concise summary of the NumPy analog for each MATLAB function.

2.2.3 Plotting data with `matplotlib`

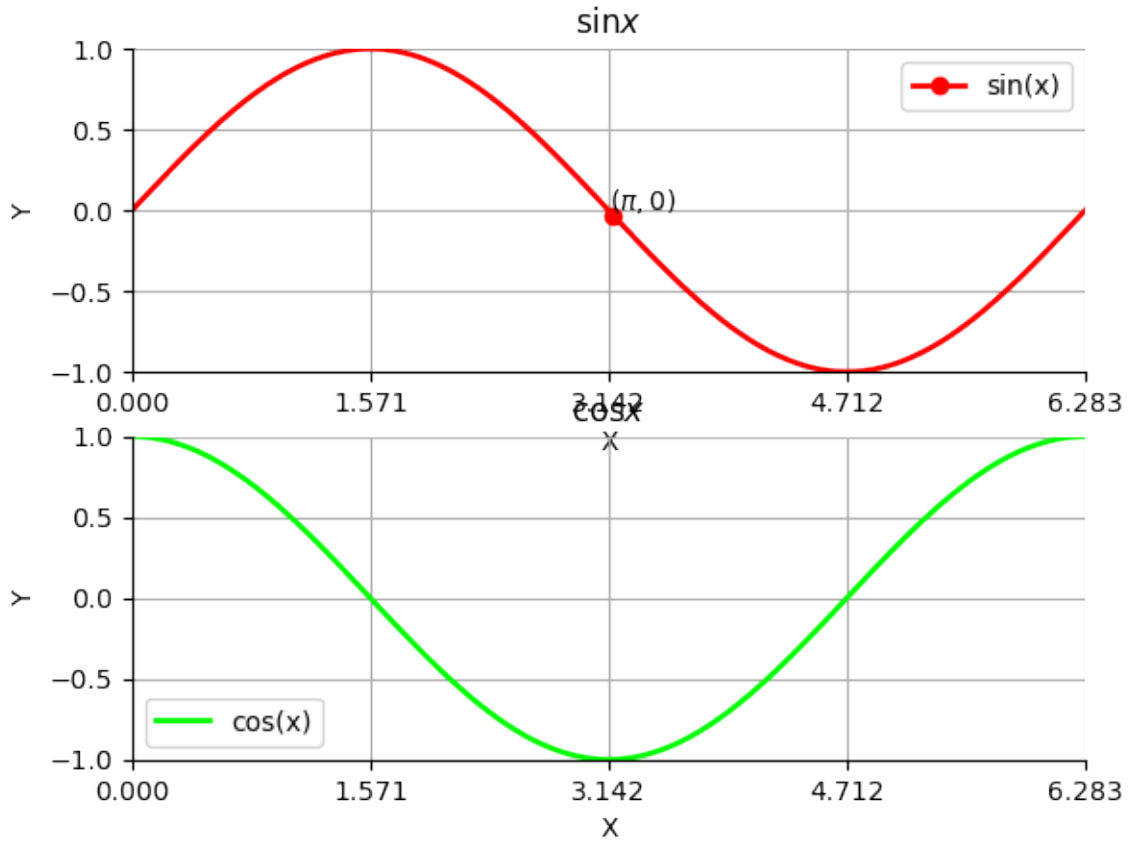
`matplotlib` is one of the simplest and popular plotting libraries for Python. The following example which produces the line plot shown below illustrates the basic functionality of `matplotlib`:

```
y0 = numpy.sin(x)
y1 = numpy.cos(x)
line0, = axes[0].plot(x, y0, label='sin(x)')
line1, = axes[1].plot(x, y1, label='cos(x)')

# set line color, style
line0.set_color((1, 0, 0))  # set color to red
line0.set_linewidth(2.)

line1.set_color((0, 1, 0))  # set color to green
line1.set_linewidth(2.)

# set axes limits
axes[0].set_xlim([0, numpy.pi * 2])
```

(continues on next page)

(continued from previous page)

```
axes[0].set_ylim([-1, 1])

axes[1].set_xlim([0, numpy.pi * 2])
axes[1].set_ylim([-1, 1])

# set axes ticks
axes[0].set_xticks([0, numpy.pi / 2, numpy.pi, numpy.pi * 3 / 2, numpy.pi * 2])
axes[0].set_yticks([-1, -0.5, 0, 0.5, 1])

axes[1].set_xticks([0, numpy.pi / 2, numpy.pi, numpy.pi * 3 / 2, numpy.pi * 2])
axes[1].set_yticks([-1, -0.5, 0, 0.5, 1])

# add title and axis labels
axes[0].set_title(r'$\sin{x}$')
axes[0].set_xlabel('X')
axes[0].set_ylabel('Y')

axes[1].set_title(r'$\cos{x}$')
axes[1].set_xlabel('X')
axes[1].set_ylabel('Y')

# add annotations
line0.set_markevery([50])
line0.set_marker('o')
axes[0].text(numpy.pi, 0, r'$(\pi, 0)$')

# turn off axis border
axes[0].spines['top'].set_visible(False)
axes[0].spines['right'].set_visible(False)

axes[1].spines['top'].set_visible(False)
axes[1].spines['right'].set_visible(False)

# turn on grid
axes[0].grid(True)
axes[1].grid(True)

# add legend
axes[0].legend()
axes[1].legend()

# display figure
# fig.show()

# save figure
png_filename = os.path.join(os.path.dirname(__file__), '../../../docs/concepts_skills/
↪software_engineering/matplotlib-example.png')
fig.savefig(png_filename, transparent=True, bbox_inches='tight')  # save in png format

pdf_filename = os.path.join(os.path.dirname(__file__), '../../../docs/concepts_skills/
↪software_engineering/matplotlib-example.pdf')
fig.savefig(pdf_filename, transparent=True, bbox_inches='tight')  # save in pdf format

os.remove(pdf_filename)
```



Plot types

In addition to line plots, `matplotlib` provides functions to create a wide range of plots

- `bar`: vertical bar plot
- `barh`: horizontal bar plot
- `errorbar`: plots lines with error bars
- `fill`: filled polygons
- `hist`: 1-D histogram
- `hist2d`: 2-D histogram
- `scatter`: scatter plot

See the matplotlib documentation for a complete list of the available plot types.

2.2.4 Developing database, command line, and web-based programs with Python

Using databases with SQLAlchemy and SQLite

Databases are useful tools for organizing and quickly searching large datasets. Relational databases are the most common type of databases. Relational databases are based around schemas or structured definitions of the types of your data and the relationships among your data. These structured definitions facilitate fast searching of large datasets. However, these schemas can also make it cumbersome to represent multiple types of data with different structures. To

overcome this limitation, several Python package provide support for editing or migrating schemas. Recently, there has been significant progress in the development of No-SQL databases which do not have fixed schemas.

Database engines are the software programs which implement SQL and No-SQL databases. Several popular SQL database engines include MySQL, Oracle, and SQLite. Several popular No-SQL database engines include CouchBase, CouchDB, and MongoDB.

SQL (Structured Query Language) is the language used to describe relational database schemas and how to insert and retrieve data to/from them. Each relational database engine uses it own variant of SQL, but the SQL languages used by MySQL, Oracle, SQLite and most other popular relational database engines are very similar.

Most of the popular database engines have their own Python interfaces. In addition, there are several Python packages such as SQLAlchemy which abstract away many of the details the individual database engines and enable Python developers to use database with little direct interaction with SQL. These packages make it easy to map between Python objects and rows in relational database tables and between attributes of those objects and columns of those tables. Thus, the packages often referred to as object-relational mappers.

This tutorial will teach you how to use SQLAlchemy to build a SQLite database to represent of single-cell organisms and their components (compartments, species, and reactions). First, we will build a *schema* that can describe cells. Second, we will build the database. Third, we will populate the database with data. Finally, we will query the database.

Define the Python object model

In order to represent organisms, compartments, species, and reactions, we must create four Python classes. These classes should inherit from `sqlalchemy.ext.declarative.declarative_base()` so that SQLAlchemy can map them onto SQLite tables and each class should define a `__tablename__` class attribute to tell SQLAchemy which table each class should be mapped onto. First, these classes should define their instance attributes and how each instance attribute should be mapped onto a column in the relational database (i.e. their type in the database). In the example below, we have created `name` and `ncbi_id` instance attributes. Furthermore, to ensure we only create once Organism instance per organism, we have declared that the ncbi_id attributes must be unique (i.e. no two objects can have the same ncbi_id).:

```
import sqlalchemy
import sqlalchemy.ext.declarative
import sqlalchemy.orm

Base = sqlalchemy.ext.declarative.declarative_base()
# :obj:`Base`: base model for local sqlite database

class Organism(Base):
    __tablename__ = 'organism'

    ncbi_id = sqlalchemy.Column(sqlalchemy.Integer(), unique=True)
    name = sqlalchemy.Column(sqlalchemy.String(), unique=True)


class Compartment(Base):
    __tablename__ = 'compartment'

    name = sqlalchemy.Column(sqlalchemy.String())


class Specie(Base):
    __tablename__ = 'specie'

    name = sqlalchemy.Column(sqlalchemy.String())
```

(continues on next page)

(continued from previous page)

```
class Reaction(Base):
    __tablename__ = 'reaction'

    name = sqlalchemy.Column(sqlalchemy.String())
```

Define the relationships between the classes

Once we have defined all of the classes that we need to represent organisms, compartments, species, and reactions, we can define how these classes are related to each other by defining relationship attributes and foreign keys. Foreign keys are columns that represent pointers to rows in other tables. Relationship attributes tell SQLAchemy how to map foreign keys between rows in tables onto references between Python objects. In order to define foreign keys, we must also define primary keys that for each table that the foreign keys can be related to. This can be done by adding `_id` attributes to each Python class.

There are four possible types of relationships between Python objects/relational table rows

- One-to-one relationships
- One-to-many relationships
- Many-to-one relationships
- Many-to-many relationships

The first three types of relationships can be defined by adding additional foreign key columns to tables. To define a many-to-many relationship, we must create an additional association table which contains foreign keys to both tables that we would like to relate.

The `cascade` argument to `sqlalchemy.orm.relationship` tells SQLAlchemy whether or not related objects should be deleted when their parents are deleted.:

```
specie_reaction = sqlalchemy.Table(
    'specie_reaction', Base.metadata,
    sqlalchemy.Column('specie__id', sqlalchemy.Integer, sqlalchemy.ForeignKey('specie.
↪_id')),
    sqlalchemy.Column('reaction__id', sqlalchemy.Integer, sqlalchemy.ForeignKey(
↪'reaction._id')),
)
# :obj:`sqlalchemy.Table`: Specie:Reaction \many-to-many association table


class Organism(Base):
    __tablename__ = 'organism'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    ncbi_id = sqlalchemy.Column(sqlalchemy.Integer(), unique=True)
    name = sqlalchemy.Column(sqlalchemy.String(), unique=True)


class Compartment(Base):
    __tablename__ = 'compartment'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String())
    organism_id = sqlalchemy.Column(sqlalchemy.Integer(), sqlalchemy.ForeignKey(
↪'organism._id'))
```

(continues on next page)

(continued from previous page)

```
    organism = sqlalchemy.orm.relationship('Organism',
                                           foreign_keys=[organism_id],
                                           backref=sqlalchemy.orm.backref(
↪'compartments', cascade="all, delete-orphan"))


class Specie(Base):
    __tablename__ = 'specie'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String())
    compartment_id = sqlalchemy.Column(sqlalchemy.Integer(), sqlalchemy.ForeignKey(
↪'compartment._id'))
    compartment = sqlalchemy.orm.relationship('Compartment',
                                              foreign_keys=[compartment_id],
                                              backref=sqlalchemy.orm.backref('species
↪', cascade="all, delete-orphan"))


class Reaction(Base):
    __tablename__ = 'reaction'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String())
    species = sqlalchemy.orm.relationship('Specie',
                                          secondary=specie_reaction,
                                          backref=sqlalchemy.orm.backref('reactions',␣
↪cascade="all, delete-orphan", single_parent=True))
```

Optimizing the schema

To speed up the querying of the database, we can instruct SQLlite to build index tables, or pre-sorted copies of the primary tables that can be used to quickly find rows using a binary search rather than having to iterate over every row in a table. This can be done by setting the `index` argument to each column constructor to `True`:

```
specie_reaction = sqlalchemy.Table(
    'specie_reaction', Base.metadata,
    sqlalchemy.Column('specie__id', sqlalchemy.Integer, sqlalchemy.ForeignKey('specie.
↪_id'), index=True),
    sqlalchemy.Column('reaction__id', sqlalchemy.Integer, sqlalchemy.ForeignKey(
↪'reaction._id'), index=True),
)
# :obj:`sqlalchemy.Table`: Specie:Reaction \many-to-many association table


class Organism(Base):
    __tablename__ = 'organism'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    ncbi_id = sqlalchemy.Column(sqlalchemy.Integer(), index=True, unique=True)
    name = sqlalchemy.Column(sqlalchemy.String(), unique=True)


class Compartment(Base):
    __tablename__ = 'compartment'
```

(continues on next page)

(continued from previous page)

```
    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String())
    organism_id = sqlalchemy.Column(sqlalchemy.Integer(), sqlalchemy.ForeignKey(
↪'organism._id'), index=True)
    organism = sqlalchemy.orm.relationship('Organism',
                                           foreign_keys=[organism_id],
                                           backref=sqlalchemy.orm.backref(
↪'compartments', cascade="all, delete-orphan"))


class Specie(Base):
    __tablename__ = 'specie'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String())
    compartment_id = sqlalchemy.Column(sqlalchemy.Integer(), sqlalchemy.ForeignKey(
↪'compartment._id'), index=True)
    compartment = sqlalchemy.orm.relationship('Compartment',
                                              foreign_keys=[compartment_id],
                                              backref=sqlalchemy.orm.backref('species
↪', cascade="all, delete-orphan"))


class Reaction(Base):
    __tablename__ = 'reaction'

    _id = sqlalchemy.Column(sqlalchemy.Integer(), primary_key=True)
    name = sqlalchemy.Column(sqlalchemy.String())
    species = sqlalchemy.orm.relationship('Specie',
                                          secondary=specie_reaction,
                                          backref=sqlalchemy.orm.backref('reactions',␣
↪cascade="all, delete-orphan", single_parent=True))
```

Create the database

Once we have defined the Python data model, we can use SQLAlchemy to generate the database:

```
DATABASE_FILENAME = 'test.sqlite'
# :obj:`str`: path to store the database

engine = sqlalchemy.create_engine('sqlite:///' + DATABASE_FILENAME)
# :obj:`sqlalchemy.engine.Engine`: database engine

# create the database
Base.metadata.create_all(engine)
```

We can use the sqlite3 lite command lite utility to inspect the schema of the database that SQLAchemy generated.:

```
sqlite3 test.sqlite .schema
```

Insert records into the database

We can insert records into the database by (1) creating a "session" on the database, (2) instantiating instances of the Organism, Compartment, Specie, and Reaction classes, (3) adding those instances to the session, and (4) "committing" the session. A session is an in-memory copy of the database which can be used to query and make changes to the database. However, the changes are not saved to the database (and therefore not accessible to other sessions) until they are committed.

Note, SQLAlchemy automatically creates constructors for each class which have keyword arguments for each instance attribute.

Note, SQLAlchemy automatically adds add objects to sessions that are linked to other objects that have been explicitly added to the session.:

```
session = sqlalchemy.orm.sessionmaker(bind=engine)()
# :obj:`sqlalchemy.orm.session.Session`: sqlalchemy session

# create homo sapiens organism with one reaction
organism = Organism(ncbi_id=9606, name='Homo sapiens')
session.add(organism)

compartment = Compartment(name='cytosol')
organism.compartments.append(compartment)

atp = Specie(name='atp')
compartment.species.append(atp)

adp = Specie(name='adp')
compartment.species.append(adp)

pi = Specie(name='pi')
compartment.species.append(pi)

h2o = Specie(name='h2o')
compartment.species.append(h2o)

h = Specie(name='h')
compartment.species.append(h)

reaction = Reaction(name='atp_hydrolysis')
reaction.species = [atp, adp, pi, h2o, h]


# create E. colii organism with one reaction
organism = Organism(ncbi_id=562, name='Escherichia coli')
session.add(organism)

compartment = Compartment(name='cytosol')
organism.compartments.append(compartment)

gtp = Specie(name='gtp')
compartment.species.append(gtp)

gdp = Specie(name='gdp')
compartment.species.append(gdp)

pi = Specie(name='pi')
compartment.species.append(pi)
```

(continues on next page)

(continued from previous page)

```
h2o = Specie(name='h2o')
compartment.species.append(h2o)

h = Specie(name='h')
compartment.species.append(h)

reaction = Reaction(name='gtp_hydrolysis')
reaction.species = [gtp, gdp, pi, h2o, h]


# save the new objects to the database
session.commit()
```

Querying the database

The following examples illustrate how to query the database:

```
# get the number organisms
organisms = session.query(Organism) \
    .count()

# select all of the organisms
organisms = session.query(Organism) \
    .all()

# order the organisms by their names
organisms = session.query(Organism) \
    .order_by(Organism.name) \
    .all()

# order the organisms by their names in descending order
organisms = session.query(Organism) \
    .order_by(Organism.name.desc()) \
    .all()

# select only organism names
organisms = session.query(Organism.name) \
    .all()

# select a subset of the database
homo_sapiens = session.query(Organism) \
    .filter(Organism.ncbi_id=9606) \
    .first()

# using joining to select a subset based on reaction names
homo_sapiens = session.query(Organism) \
    .join(Compartment, Organism.compartments) \
    .join(Specie, Compartment.species) \
    .join(Reaction, Specie.reactions) \
    .filter(Reaction.name='atp_hydrolysis') \
    .first()

# get the number of species per organism
homo_sapiens = session.query(Organism, sqlalchemy.func.count(Organism._id)) \
```

(continues on next page)

(continued from previous page)

```
    .join(Compartment, Organism.compartments) \
    .join(Specie, Compartment.species) \
    .group_by(Organism._id) \
    .all()
```

Editing and removing records

The following examples illustrate how to edit and remove records:

```
# edit the name of Homo sapiens to "H. sapiens"
homo_sapiens = session.query(Organism) \
    .filter(Organism.ncbi_id=9606) \
    .first()
homo_sapiens.name = 'H. sapiens'
session.commit()

# delete H. sapiens
session.query(Organism) \
    .filter(Organism.ncbi_id=9606) \
    .delete()
session.commit()

# delete E. coli
e_coli = session.query(Organism) \
    .filter(Organism.ncbi_id=562) \
    .first()
session.delete(e_coli)
session.commit()
```

SQLAlchemy documentation

See the SQLAlchemy documentation for additional information about building and querying databases with SQLAlchemy.

Additional tutorials

There are several good tutorial on how to use SQLAlchemy and SQLite

- Introductory Tutorial of Python's SQLAlchemy
- SQLAlchemy ORM for Beginners
- SQLAlchemy Object Relational Tutorial

Advanced concepts

- Migrations

Building command line programs with Cement

Cement is a useful package for easily building command line programs.

See the Cement quickstart guide for an introduction to Cement.

Building web-based programs with Flask

Flask is a useful package for building web-based programs.

There are several good Flask tutorials

- Flask Tutorial
- The Flask Mega-Tutorial
- Flask by Example

2.2.5 Writing code for Python 2 and 3

Because Python 2.7 and 3 are similar, and because many people still use Python 2, we recommend writing code that is compatible with both Python 2.7 and 3. Furthermore, we recommend using the `future` and `six` to write code that is compatible with both Python 2.7 and 3. http://python-future.org provides helpful summaries of the differences between Python 2.7 and 3, and how to write code that is compatible with both Python 2.7 and 3.

2.2.6 Organizing Python code into functions, classes, and modules

There are several good resources on how to effectively structure Python code, including common

- Python 3 Patterns, Recipes and Idioms
- Python Design Patterns: Video, Slides
- How To Design A Good API and Why it Matters: Video, Slides

2.2.7 Structuring Python projects

We recommend using the following principles to organize Python projects:

- Use a separate repository for each project
- Store only one package in each repository
- Structure each package as outlined below, following the recommendations provided by The Hitchhiker's Guide To Python:

```
repository_name/                # source code directory (1)
    __init__.py                 # each source code directory must contain␣
↪an __init__.py file
    __main__.py                 # optional, for command line programs
    _version.py                 # file with version number
    data/                       # directory for data files needed by the␣
↪code
tests/                          # directory for test code
    fixtures/                   # fixtures for tests
        secret/                 # git-ignored fixtures containing␣
↪usernames, passwords, and tokens
```

(continues on next page)

(continued from previous page)

```
    requirements.txt             # list of packages required to run the␣
↪tests, but not
                                 # required by the project; used by␣
↪CircleCI (2, 3)
docs/                            # directory for documentation
    conf.py                      # documentation configuration
    index.rst                    # main documentation file
    requirements.txt             # packages required to compile the␣
↪documentation (2, 3)
    requirements.rtd.txt         # list of packages required to compile␣
↪the documentation;
                                 # used by Read the Docs (2)
    _build/html/                 # directory where compiled documentation␣
↪is saved
    _static                      # optional for static files such as .css␣
↪and .js files
                                 # needed for the documentation
examples/                        # (optional) directory for examples of␣
↪how to use the code
LICENSE                          # license file
MANIFEST.in                      # list of files that should be␣
↪distributed with the package
README.md                        # Read me file; displayed by GitHub
requirements.txt                 # list of required packages (2, 3)
requirements.optional.txt        # list of optional requirements (2, 3)
setup.cfg                        # options for the installation script
setup.py                         # installation script
.circleci/                       # directory for CircleCI configuration
    config.yml                   # CircleCI configuration
    requirements.txt             # list of locations of requirements not␣
↪in PyPI (2, 3)
    downstream_dependencies.yml  # List of downstream dependencies in YAML␣
↪format (3)
.gitignore                       # list of file paths and extensions that␣
↪Git should ignore
.readthedocs.yml                 # Read the Docs configuration
```

(1) The name of the source code directory should be the same as that of the repository.

(2) For details about *requirements.txt* files see the section about *Changing package dependencies for a CircleCI build*.

(3) These dependencies can be determined automatically by karr_lab_build_utils.

- Separate code that is useful on its own—distinct from the project, into independent packages and repositories.

2.2.8 Revisioning code with Git, GitHub, and Meld

This tutorial will teach you how to use Git, GitHub, and Meld to manage code revisions/versions/changes.

Git is a popular software tool that can be used to manage changes to code, including merging changes from multiple developers. Git organizes code into "repositories". On your machine, each repository corresponds to a directory (and all of its files and subdirectories).

GitHub is a popular website for hosting Git repositories. In particular, GitHub facilities the merging of code changes among multiple developers. GitHub also makes it easy to distribute source code.

Meld is a program for graphically displaying the differences between two text files. Meld is very helpful to understanding how files have been edited.

Installing and configuring the required software

This tutorial requires git and meld. Execute this command to install these package:

```
apt-get install git meld libgnome-keyring-dev
```

Execute these commands to configure Git:

```
git config --global user.name "John Doe"
git config --global user.email "johndoe@example.com"

cd /usr/share/doc/git/contrib/credential/gnome-keyring
sudo make
git config --global credential.helper /usr/share/doc/git/contrib/credential/gnome-
↪keyring/git-credential-gnome-keyring
```

Add the following to `~/.gitconfig`:

```
[diff]
    tool = meld
[difftool]
    prompt = false
[difftool "meld"]
    cmd = meld "$LOCAL" "$REMOTE"
```

Instructions

Create a GitHub account

Visit https://github.com, click "Sign up", and follow the on screen instructions.

Join the Karr Lab GitHub organization (group)

Send your GitHub username to Jonathan so he can add you to our group.

Create a repository

1. Sign into GitHub
2. Add a repository at [https://github.com/new{]}(https://github.com/new). Note, our convention is to use `lower_camel_case` package names.

Clone a repository (download it to your computer)

Git repositories are download from GitHub by "cloning" them. Execute the following command to clone the repository for these tutorials:

```
git clone https://github.com/KarrLab/intro_to_wc_modeling.git
cd intro_to_wc_modeling
```

This will create a directory with the name "intro_to_wc_modeling", download all of the files for the repository from GitHub, and save them to the new directory.

Reviewing the files that have been changed

After you have changed one or more files, you can see a list of the files that you have changed, added, or deleted by running `git status`. To see the changes made to an individual file execute `git difftool path/to/file`.

Commit changes to a repository

After you have changed one or more files, you can commit (save) those changes to the repository.

1. Select the changes that you would like to commit by executing `git add path/to/file`. Repeat this for each change you would like to commit.
2. Execute `git commit -m "<brief description of the changes>"`

Marking versions with tags

You can use tags to mark the version numbers of key revisions. This can be done by running `git tag <version_number>`.

Push your changes to GitHub

Once you are ready to share one or more commits with the rest of our lab, you can push them to GitHub by executing `git push`. If you also need to push tags, then you will need to run `git push --tags`.

Pull changes from other developers from GitHub

To retrieve changes made by other developers, execute `git pull`. Note, if you have made changes which conflict with those made by other developers, Git will prompt you to manually review the conflict lines of code.

Using graphical programs to manage repositories

Several graphical programs are also available to manage Git repositories

- Git Cola
- GitHub Desktop (Windows, Mac)
- GitKraken
- SmartGit

Additional Git tutorials

There are numerous additional tutorials which cover more advanced concepts such as branching

- Try Git
- Git tutorial
- Git Tutorials and Training

2.2.9 Testing Python code with unittest, pytest, and Coverage

The goal of this tutorial to teach you how to effectively test and debug Python code.

Unit testing is a powerful methodology for testing and debugging code and ensuring that code works as intended. You can unit test your code by writing numerous tests of your code, each of which executes the code and assert that the expected result is produced.

To use unit testing effectively, it is best to begin by writing tests of individual pieces of your code, debugging each individual piece until they all work, and then proceeding to write larger tests of larger assemblies of code.

Collectively, your tests should cover all of the logic of your code. The following are the most popular metrics for evaluating the coverage of your tests. We recommend using statement coverage because this the simplest metric to evaluate and the most widely supported metric.

- **Statement/line coverage**: This metric evaluates the fraction of the lines of your code which were evaluated during your tests and which specific lines were not evaluated by your tests. Statement coverage is the most popular coverage metric. Statement coverage can be assessed using the `coverage` and `pytest-cov` packages and the results can be analyzed using `coverage`, Coveralls, and Code Climate.
- **Branch coverage**: This metric evalutes the fraction of the branches (if, elif, else, try, except, finally, for, while statements) of your code that were evaluated during your tests and which specific branches were not covered. Branch coverage can be a more reliable metric of the completeness of your coverage because it isn't biased by large blocks of simple code. Branch coverage can be analyzed using the `coverage` package.
- **Decision/multiple condition converage**: This metric evaluates the fraction of the conditions of your branches that were covered by your tests, and the specific conditions that were not covered. Decision coverage is a more thorough metric than branch coverage because it checks every condition of every branch. Decision coverage can be analyzed using the `instrumental` package.
- **Path coverage**: This metric evaluates the fraction of paths of your code that were evaluated by your tests. This is a more thorough metric than decision coverage which is evaluates every combination of conditions. However, path coverage is not a practical metric due to the combinatorial number of paths in large codes. To our knowledge, there is no package for path coverage.

Required packages

Execute the following commands to install the packages required for this tutorial:

```
apt-get install \
    python \
    python-pip
pip install \
    capturer \
    cement \
    coverage \
    numpy \
```

(continues on next page)

(continued from previous page)

```
    pytest \
    pytest-cov \
```

File naming and organization

Our convention is to store tests within separate `tests` subdirectories within each repository. Any non-Python files which are needed for testing, can be organized in a `fixtures` subdirectory. The `tests` directory should also contain a `requirements.txt` file which lists all of the packages that are needed to run the tests.

Often it is helpful create one file for all of the tests for each source code file and to name this `test_<source_filename>.py`.

Taken together, your test code should be organized as follows:

```
/path/to/repo/
    tests/                              # directory for all of the test files
        <source_modulename>             # parallel directory structure to source code
            test_<source_filename>.py   # parallel filenames to source code
        fixtures/                       # files needed by the tests
        requirements.txt                # list of packages needed to run the tests
```

Writing tests

In the remainder of the tutorial, we will write tests for the code located in `/path/to/this/repo/intro_to_wc_modeling/concepts_skills/software_engineering/unit_testing/` to run a simple stochastic simulation.

1. Create a file for the tests, *tests/concepts_skills/software_engineering/unit_testing/test_core.py*
2. Write a test file. For example:

```
from intro_to_wc_modeling.concepts_skills.software_engineering.unit_testing␣
↪import core
import unittest

class TestSimulation(unittest.TestCase):
    class NewClass():
        pass

    @classmethod
    def setUpClass(cls):
        """ Code to execute before all of the tests. For example, this can be␣
↪used to create temporary
        files.
        """
        pass

    @classmethod
    def tearDownClass(cls):
        """ Code to execute after all of the tests. For example, this can be used␣
↪to clean up temporary
        files.
        """
        pass
```

(continues on next page)

(continued from previous page)

```
    def setUp(self):
        """ Code to execute before each test. """
        pass

    def tearDown(self):
        """ Code to execute after each test. """
        pass

    def test_run(self):
        self.NewClass

        # run code
        sim = core.Simulation()
        hist = sim.run(time_max=10)

        # check the result
        self.assertEqual(hist.times[0], 0.)
```

Each test method should begin within the prefix `test_`

`unittest` provides numerous assertions such as those below that can be used to verify that code produces the expected results. See the unittest documentation and the numpy.testing documentation for additional assertions.

- `unittest.TestCase.assertEqual`
- `unittest.TestCase.assertNotEqual`
- `unittest.TestCase.assertTrue`
- `unittest.TestCase.assertFalse`
- `unittest.TestCase.assertIsInstance`
- `unittest.TestCase.assertGreater`
- `unittest.TestCase.assertLess`
- `unittest.TestCase.assertAlmostEqual`
- `unittest.TestCase.assertRaises`
- `numpy.testing.assert_array_equal`
- `numpy.testing.assert_array_almost_equal`

The `setUp` and `tearDown` methods can be used to organize the code that should be executed before and after each individual test. This is often useful for creating and removing temporary files. Similarly, the `setUpClass` and `tearDownClass` can be used to organize code that should be executed before and after the execution of all of the tests. This can be helpful to organizing computationally expensive operations that don't need to be executed multiple times.

Testing stochastic algorithms

Stochastic codes should be validated by testing the statistical distribution of their output. Typically this is done with the following process

1. Run the code many times and keep a list of the outputs
2. Run a statistical test of the distribution of the outputs. At a minimum test the average of the distribution is close to the expected value. If possible, also test the variance of the distribution and higher-order moments of the distribution.

Testing standard output

The `capturer` package is helpful for collecting and testing stdout generated by code. This can be used to test standard output as shown in the example below:

```
import capturer

def test_stdout(self):
    with capturer.CaptureOutput() as captured:
        run_method()
        stdout = captured.get_text()
        self.assertEqual(stdout, expected)
```

Testing cement command line programs

Cement command line programs can be tested as illustrated below:

from intro_to_wc_modeling import __main__ import capturer

def test(self): # array of command line arguments, just as they would be supplied at the command line except # each should be an element of an array arv = ['value']

with __main__.App(argv=argv) as app:

with capturer.CaptureOutput() as captured: app.run() self.assertEqual(captured.get_text(), expected_value)

See [*tests/test_main.py*](tests/test_main.py) for an annotated example.

Testing for multiple version of Python

You should test your code on both major versions of Python. This can be done as follows:

```
python2 -m pytest tests
python3 -m pytest tests
```

Running your tests

You can use pytest as follows to run all or a subset of your tests:

```
python -m pytest tests                                          # run all tests in a␣
↪directory
python -m pytest tests/test_core.py                             # run all tests in a␣
↪file
python -m pytest tests/test_core.py::TestSimulation             # run all tests in a␣
↪class
python -m pytest tests/test_core.py::TestSimulation::test_run   # run a specific test
python -m pytest tests -k run                                   # run tests that␣
↪contain `run` in their name
python -m pytest tests -s                                       # use the `-s` option␣
↪to display the stdout generated by the tests
```

Analyzing the coverage of your tests

Test coverage can be analyzed as follows:

python -m pytest –cov=intro_to_wc_modeling tests

This prints a summary of the coverage to the console and saves the details to `.coverage`.

The following can be used to generated a more detailed HTML coverage report. The report will be saved to `htmlcov/`:

```
coverage html
```

You can view the HTML report by opening `file:///path/to/intro_to_wc_modeling/htmlcov/index.html` in your browser. Green indicates lines that were executed by the tests. Red indicates lines that were not executed. Large amounts of red lines means that more tests are needed. Ideally, code would be tested to 100% coverage.

Additional tutorials

There are numerous additional tutorials on unit testing Python

- Understanding Unit Testing
- Testing Python Applications with Pytest

2.2.10 Debugging Python code using the PyCharm debugger

Debuggers are useful tools for debugging code that can be are far more powerful than print statements or logging. Debuggers allow users to interactively inspect Python programs during their execution. In particular, debuggers allow users to set breakpoints at which the Python interpret should halt its execution and enable the user to inspect the state of the program (value of each variable in the namespace and in all parent namespaces), run an arbitrary Python code (e.g. to print something out), step through the code (instruct the Python interpreter to execute one additional instructions of the program), and/or resume the program. Debuggers also allow users to set conditional breakpoints which can be used to halt the execution of a program when the value of a variable meets a specific condition. Together, debuggers make it easy to trace through programs and find errors.

Most IDEs include debuggers and there are debugger plugins for text editors such as Sublime. We recommend the PyCharm debugger. There are several tutorials on how to use PyCharm to debug code.

- Debugging in Python (using PyCharm)
- PyCharm Debugger Tutorial
- Getting Started with PyCharm 6/8: Debugging

2.2.11 Documenting Python code with Sphinx

The goal of this tutorial is to teach you how to document Python code to help other programmers – and yourself in the future – understand your code. We recommend that you document each attribute, argument, method and class, and also document each module and package.

reStructuredText is the most commonly used markup format for documenting Python code and Sphinx is the most commonly used tool for compiling Python documentation.

Required packages

Execute the following commands to install the packages required for this tutorial:

```
apt-get install \
    python \
    python-pip
pip install \
    configparse \                                # configuration file parser
    robpol86-sphinxcontrib-googleanalytics \ # sphinx support for Google analytics
    sphinx \                                     # documentation generator
    sphinx-rtd-theme                             # sphinx HTML theme
```

File naming and organization

Our convention is to place all top-level documentation within a separate `docs` directory within each repository and to embed all API documentation in the source code.:

```
/path/to/repo/
    docs/                       # directory for all of the documentation files
        index.rst               # main documentation file
        conf.py                 # Sphinx configuration
        requirements.txt        # packages required to compile the documentation
        requirements.rtd.txt # packages required to compile the documentation; used␣
↪by Read the Docs
    setup.cfg                   # contains a setting which specifies the packages for␣
↪which
                                # API documentation should be generated
```

Generating a Sphinx configuration file

Sphinx is highly configurable and can be configured using the `docs/conf.py` file.

You can generate a Sphinx configuration file by running the `sphinx-quickstart` utility and following the on screen instructions.

Note: we are using a heavily modified Sphinx configuration file. See `karr_lab_build_utils.templates.docs.conf.py` for our template. In particular, we are enabling the `napoleon` extension to support Google-style argument and attribute doc strings.

Writing documentation

In the remainder of the tutorial, we will write tests for the code located in `/path/to/this/repo/intro_to_wc_modeling/concepts_skills/software_engineering/unit_testing/`.

Using the *napoleon* style, you can document each class, method, and attribute. See `intro_to_wc_modeling/concepts_skills/software_engineering/unit_testing/core.py` for a complete example.

Classes

```
class Class():
    """ Description

    Attributes:
        name (:obj:`core.Simulation`): description
    """

    ...
```

Methods

```
def method():
    """ Description

    Args:
        name (:obj:`type`): description
        name (:obj:`type`, optional): description

    Returns:
        :obj:`type`: description

    Raises:
        :obj:`type`: description
    """

    ...
```

Top-level documentation

We can also add top-level documentation to `docs/index.rst` using the reStructuredText markup language. See the reStructuredText primer for a brief tutorial about the reStructuredText markup language.

README

In addition to this documentation, we also recommend providing a brief README file with each repository and we recommend embedded status badges at the top of this file. These badges can be embedded as shown in the example below:

```
<!-- [![PyPI package](https://img.shields.io/pypi/v/intro_to_wc_modeling.svg)](https:/
↪/pypi.python.org/pypi/intro_to_wc_modeling) -->
[![Documentation](https://img.shields.io/badge/docs-latest-green.svg)](http://docs.
↪karrlab.org/karrlab_intro_to_wc_modeling)
[![Test results](https://circleci.com/gh/KarrLab/intro_to_wc_modeling.svg?
↪style=shield)](https://circleci.com/gh/KarrLab/intro_to_wc_modeling)
[![Test coverage](https://coveralls.io/repos/github/KarrLab/intro_to_wc_modeling/
↪badge.svg)](https://coveralls.io/github/KarrLab/intro_to_wc_modeling)
[![Code analysis](https://codeclimate.com/github/KarrLab/intro_to_wc_modeling/badges/
↪gpa.svg)](https://codeclimate.com/github/KarrLab/intro_to_wc_modeling)
[![License](https://img.shields.io/github/license/KarrLab/intro_to_wc_modeling.
↪svg)](LICENSE)
![Analytics](https://ga-beacon.appspot.com/UA-86759801-1/intro_to_wc_modeling/README.
↪md?pixel)
```

Compiling the documentation

Run the following to compile the documentation:

```
sphinx-build docs docs/_build/html
```

Sphinx will print out any errors in the documentation. These must be fixed to properly generate the documentation.

It can be viewed by opening `docs/_build/html/index.html` in your browser.

2.2.12 Continuously testing Python code with CircleCI, Coveralls, Code Climate, and the Karr Lab's dashboards

To use testing to identify errors as quickly as possible when they are comparatively easy to fix, you should evaluate your tests each time you commit your code. However, this is tedious and it can be time-consuming, particularly as the software gets large and the number of tests grows. Luckily, software engineers have developed tools called continuous integration servers that can automatically test packages each time they are modified.

To use testing effectively on team projects, you should fix broken tests immediately. If you don't fix test failures quickly and instead allow tests to remain broken, you will forfeit your ability to use tests to quickly detect new errors.

We are using the CircleCI cloud-based continuous integration system that has integration with GitHub. Each time you push your code to GitHub, GitHub triggers a "hook" which instructs CircleCI to "build" your code, which includes running all of your tests and notifying you of any errors. While CircleCI is primarily designed to run tests, CircleCI builds are very flexible and can be used to run any arbitrary code upon each trigger. We have used this flexibility to instruct CircleCI to execute the following tasks within each build

- Boot our custom virtual machine
- Install any additional software needed to test the package
- Install the package
- Run all of tests using Python 2 and 3
- Record the coverage of the tests
- Compile the documentation for the package
- Save the results of the tests
- Upload the results of the tests to our test history server
- Upload the coverage of the tests to two online coverage analysis programs, Code Climate and Coveralls

Required packages

Execute the following commands to install the packages required for this tutorial:

```
sudo curl -o /usr/local/bin/circleci https://circle-downloads.s3.amazonaws.com/
↪releases/build_agent_wrapper/circleci
sudo chmod +x /usr/local/bin/circleci
```

Using the CircleCI cloud-based continuous integration system

Follow these instructions to use CircleCI to continuously test a GitHub repository

1. Log into CircleCI using your GitHub account

2. Click on the *Projects* tab
3. Click the *Add Project* button
4. If you see multiple organizations, click on the *KarrLab* button
5. Click the *Follow Project* button for any repository you want to compile and test on CircleCI
6. Add a CircleCI configuration file, `/path/to/repo/.circleci/config.yml`, to the repository to instruct CircleCI what to execute within each build. This includes the following instructions

See *Using the CircleCI cloud-based continuous integration system* please.

- Which container/virtual machine should be used to run the build. We are using a custom container so that little additional software needs to be installed to test our code. See *How to build a Ubuntu Linux image with Docker* for more information about how to create and use custom Linux containers.
- Which GitHub repository to checkout.
- How to install any additional packages needed to execute the tests.
- Instructions on how to run the tests and store the results.

See `.circleci/config.yml` for an example and see the CircleCI documentation for more information about configuring CircleCI builds.

In order to upload our test and coverage results to Code Climate, Coveralls, and our lab server, we must set three environment variables in the CircleCI settings for each repository. The values of these variables should be the tokens needed to authenticate with Code Climate, Coveralls, and our lab server. These tokens can be obtained from the corresponding Code Climate and Coveralls projects for each repository.

- `CODECLIMATE_REPO_TOKEN`: *obtain from the corresponding Code Climate project*
- `COVERALLS_REPO_TOKEN`: *obtain from the corresponding Coveralls project*
- `TEST_SERVER_TOKEN`: `jxdLhmaPkakbrdTs5MRgKD7p`

Optimizing the runtime of CircleCI builds by loading rather than compiling dependent packages

There are two main mechanisms to decrease the runtime of CircleCI builds by loading rather than compiling dependent packages:

- Use CircleCI's cache to avoid repeatedly compiling the dependent packages.

  This can configured in `.circleci/config.yml` as illustrated below:

```
- restore_cache:
    keys:
      - cache-vXXX.-{{ .Branch }}-{{ checksum "requirements.txt" }}
      - cache-vXXX-{{ .Branch }}-
      - cache-vXXX-

...

- save_cache:
    key: cache-vXXX-{{ .Branch }}-{{ checksum "requirements.txt" }}
```

  You can clear these caches by incrementing by the version number `vXXX` in `.circleci/config.yml` and pushing the updated file to GitHub. This helpful if you want to force the build to compile the dependent package.

- Create your own Docker image which already has the packages compiled

The Dockerfile for the Docker image that the Karr Lab uses with CircleCI is located at https://github.com/KarrLab/karr_lab_docker_images/tree/master/build.

See *How to build a Ubuntu Linux image with Docker* for more information.

The Karr Lab uses both of these mechanisms.

Changing package dependencies for a CircleCI build

Occasionally, you may need to change the dependencies of a repository. The following steps can perform this task:

1. Update the `pip requirements.txt` files which identify packages that the repository uses. To automate this process, use the commands in karr_lab_build_utils that can obtain a package's dependencies and identify dependencies that may be missing or unnecessary.
   - `./requirements.txt` describes the dependencies of the package. It lists the package's immediate dependencies, i.e., the other packages that are imported, and may constraint which versions are suitable for the package. It should not contain URLs, specify the source which should provide a package, or specify the specific version of a dependency to install. The systems administrator who configures the package's environment, not the programmer, should be responsible for these details.
   - `./requirements.optional.txt` describes the package's optional dependencies.
   - `./tests/requirements.txt` lists the dependencies of the package's tests.
   - `./docs/requirements.txt` describes the dependencies of the software that compiles the package's documentation.
   - `.circleci/requirements.txt` tells CircleCI where to obtain dependencies that are not located in PyPI. Dependencies can be identified by GitHub URLs with the format `git+https://github.com/--account_name--/--package_name--.git#egg=--package_name--`. All dependencies–including transitive dependencies–must be listed. The list must be arranged in dependency order, so that if package *y* depends on package *x* then *x* precedes *y*, as in a topological sort of the dependencies. This file works around limitations in pip and PyPI.
   - `./docs/requirements.rtd.txt` tells Read the Docs where to obtain dependencies that are not located in PyPI.
2. Commit the changes to the `requirements.txt` files to your code repository.

If there are errors in the compilation and/or installation of the new dependencies, you can try rebuilding the build without its cache. As described above, we recommend using CircleCI's cache to avoid repeatedly recompiling dependent packages. The cache avoids recompiling dependent packages by storing them after the first time they are built, and loading them on subsequent builds. You can force CircleCI to create a new cache by incrementing the cache version number `vXXX` specified in `.circleci/config.yml` and pushing the updated configuration file to your code repository:

```
- restore_cache:
    keys:
      - cache-vXXX.-{{ .Branch }}-{{ checksum "requirements.txt" }}
      - cache-vXXX-{{ .Branch }}-
      - cache-vXXX-

...

- save_cache:
    key: cache-vXXX-{{ .Branch }}-{{ checksum "requirements.txt" }}
```

All other builds that require your package should be configured to update its requirements at the beginning of every build. This can be implementing using pip's `-U` option. Note, the Karr Lab's builds are already configured to update their requirements at the beginning of every build.

Debugging CircleCI builds

There are four ways to debug CircleCI builds.

- You can iteratively edit and push your `.circleci/config.yml` file. However, this is slow because it is not interactive.
- From the CircleCI website, you can rebuild a build with SSH access using the "Rebuild" button at the top-right of the page for the build. After the new build starts, CircleCI will provide you the IP address to SSH into the machine which is running your build. However, this is limited to 2 h, the CircleCI virtual machines are somewhat slow because they are running on top of shared hardware, and any changes you make are not saved to the build image.
- You can use the CircleCI local executor (see below) to emulate CircleCI locally. This is a powerful way to debug CircleCI builds. However, this takes more effort to setup because it requires Docker.
- You can interactively run your code on the Docker build image. This is also a powerful way to debug CircleCI builds. However, this takes more effort to setup because it requires Docker.

Debugging CircleCI builds locally

The CircleCI local executor and interactively running your code on the build image are powerful ways to debug CircleCI builds. Below are instructions for utilizing these approaches.

1. Install Docker (see Section 6)
2. Install the CircleCI command line tool:

```
sudo curl -o /usr/local/bin/circleci https://circle-downloads.s3.amazonaws.com/
↪releases/build_agent_wrapper/circleci
sudo chmod +x /usr/local/bin/circleci
```

3. Use the Docker CLI to run a build locally

```
cd /path/to/repo
circleci build
```

   Note, this will ignore the Git checkout instructions and instead execute the build instructions using the code in `/path/to/repo`.

   Note also, if your builds need SSH keys to clone code from a private repository, you will need to prepare a Docker image with the SSH key(s) loaded into it. See this example Dockerfile.

   See the CircleCI documentation for more information about running builds locally.

4. Use Docker to interactively run the Docker build image:

```
docker run -it karrlab/wc_env_dependencies:latest bash
```

See https://github.com/KarrLab/karr_lab_docker_images/blob/master/build/test_packages.py for a detailed example of how to run builds locally using the CircleCI CLI and Docker.

Code Climate

Follow these instructions to use Code Climate to review the test coverage of a repository

1. Log into Code Climate using your GitHub account
2. Click one of the *Add a repository* links
3. Select the desired repository
4. To view the analysis, return to your dashboard and select the package from the dashboard
5. To push coverage data to Code Climate
    1. Open the settings for the package
    2. Navigate to the *Test Coverage* settings
    3. Copy the *Test reporter ID*
    4. Create an environment variable in the corresponding CircleCI build with the key = `CODECLIMATE_REPO_TOKEN` and the value = the value of the *Test reporter ID*

Once coverage data has been uploaded to Code Climate, you can use the Code Climate GUI to browse the coverage of each module, file, class, method, and line.

Coveralls

Follow these instructions to use Coveralls to review the test coverage of a repository

1. Log into Coveralls using your GitHub account
2. Click the *Add repos* button
3. Turn the selected the repository on
4. To push coverage data to Coveralls,
    1. Copy the *repo_token*
    2. Create an environment variable in the corresponding CircleCI build with the key = `COVERALLS_REPO_TOKEN` and the value = the value of *repo_token*

Once coverage data has been uploaded to Coveralls, you can use the Coveralls GUI to browse the coverage of each module, file, class, method, and line.

Karr Lab test results dashboard (tests.karrlab.org)

Follow these instructions to use the Karr Lab test results dashboard to review the test results from a CircleCI build

1. Create an environment variable in the CircleCI build with the name `TEST_SERVER_TOKEN` and value `jxdLhmaPkakbrdTs5MRgKD7p`
2. Open http://tests.karrlab.org in you browser. Once tests results have been uploaded to our tests history server, our test results dashboard will allow you to graphically review test results, as well as the performance of each test over time.

Karr Lab software development dashboard (code.karrlab.org)

Follow these instructions to use the Karr Lab software development dashboard to monitor the status of a repository

1. SSH into code.karrlab.org
2. Add a repository configuration file to `/home/karrlab_code/code.karrlab.org/repo/<repo-name>.json`
3. Copy the syntax from the other files in the same directory
4. Open http://code.karrlab.org in your browser. You should now be able to see the status of the repository, its CircleCI builds, the results of its results, the coverage of its tests, and severals statistics about how many times the repository has been cloned, forked, and downloaded from GitHub and PyPI.

2.2.13 Distributing Python software with GitHub, PyPI, Docker Hub, and Read The Docs

The goal of this tutorial is to teach you how to distribute software with GitHub, PyPI, and Read the Docs. GitHub is a code repository that can be used to distribute source code. PyPI is a repository for Python software that makes it easy for users to use pip to install your software. Docker Hub is a repository for Docker containers that makes it easy to distribute entire virtual machines that have your all of your software installed and fully configured. Read The Docs is a repository for documentation that can be use to easily distribute documentation to end users and external developers.

Required packages

Execute the following commands to install the packages required for this tutorial:

```
apt-get install pandoc
pip install \
    pypandoc \
    setuptools \
    twine \
    wheel
```

Prepare your package for distribution

Annotate the version number of your package in `_version.py` and `__init__.py`

Save the following to the `_version.py` file of your package, e.g. `/path/to/intro_to_wc_modeling/intro_to_wc_modeling/_version.py`:

```
__version__ = '0.0.1'y
```

Save the following to the `__init__.py` file of your package, e.g. `/path/to/package/intro_to_wc_modeling/__init__.py`:

```
import pkg_resources

from ._version.py import __version__
# :obj:`str`: version
```

Create a `README.md` file with an overview of the package

Save a brief description of the package to `/path/to/package/README.md`. GitHub will display the content of this file on the landing page for the repository. For example:

```
# intro_to_wc_modeling

The goal of this tutorial is to teach you how to test and document Python code.
```

Create a file `requirements.txt` which lists the required dependencies

The following example illustrates how to use `/path/to/package/requirements.txt` to specify requirements including how to specify package sources, how to specify version dependencies, and how to specify required options.:

```
numpy
scipy<=1.2
matplotlib==2.3[option]
git+https://github.com/KarrLab/obj_tables.git#egg=obj_tables
```

Packages that should be installed from PyPI should be listed by their names. Packages that should be installed from GitHub should be listed by their GitHub URL.

Version dependencies can be specified with '<', '>', '<=', '>=', and '='.

Required options can be specified by post-pending option names to each dependency.

Similarly, `docs/requirements.txt` and `tests/requirements.txt` can be used to specify packages required for testing and documentation.

In addition, `.circleci/requirements.txt` and `docs/requirements.rtd.txt` can be used to specify the locations of packages that are not in PyPI for CircleCI and Read the Docs.

Create a file `requirements.optional.txt` which lists the dependencies

Optional dependencies can be listed in `/path/to/package/requirements.optional.txt` according this syntax:

> [option] dependency_1 dependency_2 ...

These optional dependencies can be installed by post-pending the option name(s) during pip and setup.py commands, e.g.:

```
pip install package_name[option_name]
```

Create a license file

Save the following to */path/to/package/LICENSE*:

```
The MIT License (MIT)

Copyright (c) <Year> Karr Lab

Permission is hereby granted, free of charge, to any person obtaining a copy
```

(continues on next page)

(continued from previous page)

```
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Create a setup configuration file

Create a setup configuration file by following this example and saving it to `/path/to/package/setup.cfg`:

```
[bdist_wheel]
universal = 1

[coverage:run]
source =
    intro_to_wc_modeling

[sphinx-apidocs]
packages =
    intro_to_wc_modeling
```

Create a `MANIFEST.in` file which describes additional files that should be packaged with your Python code

For example, save the following to `/path/to/package/MANIFEST.in`:

```
# documentation
include README.rst

# license
include LICENSE

# requirements
include requirements.txt
include requirements.optional.txt
```

Create a setup script

You can use the `setuptools` package to build a install script for your package. Simply edit this template and save it to `/path/to/intro_to_wc_modeling/setup.py`:

```
import setuptools
try:
    import pkg_utils
except ImportError:
    import subprocess
    import sys
    subprocess.check_call(
      [sys.executable, "-m", "pip", "install", "pkg_utils"])
    import pkg_utils
import os

name = 'intro_to_wc_modeling'
dirname = os.path.dirname(__file__)

# get package metadata
md = pkg_utils.get_package_metadata(dirname, name)

# install package
setup(
    name=name,
    version=md.version,

    description='Python tutorial',
    long_description=md.long_description,

    # The project's main homepage.
    url='https://github.com/KarrLab/' + name,
    download_url='https://github.com/KarrLab/' + name,

    author='Jonathan Karr',
    author_email='jonrkarr@gmail.com',

    license='MIT',

    # See https://pypi.python.org/pypi?%3Aaction=list_classifiers
    classifiers=[
        'Development Status :: 3 - Alpha',
        'Intended Audience :: Developers',
        'Topic :: Software Development',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python',
    ],

    keywords='python, tutorial',

    # packages not prepared yet
    packages=setuptools.find_packages(exclude=['tests', 'tests.*']),
    entry_points={
        'console_scripts': [
            'intro-to-wc-modeling = intro_to_wc_modeling.__main__:main',
        ],
    },

    install_requires=md.install_requires,
    extras_require=md.extras_require,
    tests_require=md.tests_require,
    dependency_links=md.dependency_links.
)
```

Use the `entry_points` argument to specify the location(s) of command line programs that should be created. Use the `install_requires` argument to list any dependencies. Use the `tests_require` argument to specify any additional packages needed to run the tests.

See The Hitchhiker's Guide to Packaging for a more detailed explanation of the arguments to setup.

You can test the install script by running it locally:

```
pip install -e .
```

Distributing source code with GitHub

GitHub can be used to distribute source code simply by changing the public/private setting of a repository. The versions of key revisions should be marked using Git tags as illustrated below. See Section 2.2.8 for more information about using Git and GitHub.:

```
git add <path>
git commit -m "<message>"
git tag 0.0.8
git push --tags
```

Distributing Python packages with PyPI

Follow the steps below to distribute your code via PyPI.

1. Create an account at https://pypi.python.org
2. Save your login information to `~/.pypirc`:

   ```
   [distutils]
   index-servers =
       pypi

   [pypi]
   repository: https://upload.pypi.org/legacy/
   username: <username>
   password: <password>
   ```

3. Convert your `README.md` file to `.rst` format:

   ```
   pandoc --from=markdown --to=rst --output=README.rst README.md
   ```

4. Compile your package for source code and binary distribution:

   ```
   python2 setup.py sdist bdist_wheel
   python3 setup.py sdist bdist_wheel
   ```

5. Upload your package to PyPI:

   ```
   twine upload dist/*
   ```

There are also several online tutorials with more information about how to upload packages to PyPI

- How to submit a package to PyPI
- Python Packaging User Guide
- Uploading to PyPI

Distributing containers with Docker Hub

Docker Hub can be used to distribute virtual machines simply by changing the public/private setting of a repository. See *How to build a Ubuntu Linux image with Docker* for more information about using Docker and Docker Hub.

Distributing documentation with Read The Docs

After you have configured Sphinx, committed your code to GitHub, and made your repository public, follow these instructions to configure Read The Docs to compile the documentation for your code upon each push to GitHub. Note, your configuration must follow the Sphinx configuration template in `karr_lab_build_utils` for Read The Docs to properly compile your documentation. Note also, Read The Docs can only be used to compile and distribute documentation for public GitHub repositories.

1. Create an account at https://readthedocs.org
2. Log into Read The Docs
3. Click the "Import a repository" button
4. Select the repository that you wish to distribute
5. Create the project
6. Use the "Settings" and "Advanced Settings" panels to edit the settings for the project.
   - Set the homepage and tags
   - Set the requirements file to `docs/requirements.rtd.txt`
   - Set the Python configuration file to `docs/conf.py`
   - Set the Python interpreter to `CPython 3.x`
7. Optionally, use YAML files to configure the conda environment used to build the documentation within Read the Docs. This is helpful for documenting packages that depend on OS packages. The default Read the Docs conda environment cannot install OS packages, but some of these dependencies can be obtained from conda.:
   - Add the following to `/path/to/package/.readthedocs.yml`:

```
python:
   version: 3
   setup_py_install: true
requirements_file: docs/requirements.rtd.txt
conda:
    file: docs/conda.environment.yml
```

   - Add the following to `/path/to/package/docs/conda.environment.yml`:

```
name: <package>-docs
channels:
  - conda-forge
  - defaults
dependencies:
  - cython
  - pip
  - python
  - sphinx
  - pip:
    - configparser
    - sphinx_rtd_theme
```

(continues on next page)

(continued from previous page)

```
    - robpol86-sphinxcontrib-googleanalytics
    - sphinxcontrib-bibtex
    - sphinxcontrib-spelling
```

8. Add your email in the "Notifications panel" so that you receive notifications documentation compilation errors
9. Check for errors

- Navigate to "Builds"
- Click on the latest build
- Browse the tabs for errors and warnings

2.2.14 Recommended Python development tools

Below are several recommended programs for developing Python code:

- Text editors
  - Atom
  - Notepad++
  - Sublime
- Interactive Python shells
  - ipython
  - Jupyter notebooks
- Integrated development environments (IDEs) with debuggers
  - Canopy: tailored for science
  - PyCharm: good support for testing and general development
  - Spyder: tailored for science
- Test runners
  - nose
  - pytest
- Test coverage tools
  - coverage
  - instrumental
- Profilers
  - cProfile
  - line_profiler
  - memory_profiler
- Documentation generation
  - Sphinx
  - Napoleon sphinx extension
- Installing packages

  - pip
  - PyPI
- Packaging code
  - setuptools
  - twine

Installation

Python

Execute the following command to install Python 2 and 3:

```
apt-get install python python3
```

Pip package manager

Execute the following command to install the pip package manager:

```
apt-get install python-pip python3-pip
```

ipython interactive shell

Execute the following command to install the ipython interactive shell:

```
apt-get install ipython ipython3
```

Sublime code editor

Execute the following command to install the Sublime code editor:

```
sudo add-apt-repository ppa:webupd8team/sublime-text-3
sudo apt-get update
sudo apt-get install sublime-text-installer
```

We also recommend editing the following settings:

- Preferences >> Key Bindings:

```
[
    { "keys": ["ctrl+shift+r"], "command": "unbound"}
]
```

- Preferences >> Package control >> Install package >> AutoPEP8
- Preferences >> Package settings >> AutoPep8 >> Settings-User:

```
[{"keys": ["ctrl+shift+r"], "command": "auto_pep8", "args": {"preview": false}}]
```

PyCharm IDE

Execute the following command to install the PyCharm IDE:

```
mv ~/Downloads/pycharm-community-2017.1.tar.gz /opt/
    tar -xzf pycharm-community-2017.1.tar.gz
    cd pycharm-community-2017.1
    ./pycharm.sh
```

We also recommend editing the following settings:

- File >> Settings >> Tools >> Python Integrated Tools >> Default test runner: set to py.test
- Run >> Edit configurations >> Defaults >> Python tests >> py.test: add additional arguments "–capture=no"
- Run >> Edit configurations >> Defaults >> Python tests >> Nosetests: add additional arguments "–nocapture"

2.2.15 Comparison between Python and other languages

Python vs MATLAB

Several websites have nice summaries of the advantages of Python over MATLAB:

- Python vs MATLAB
- A Python Primer for MATLAB Users
- NumPy for MATLAB users

2.3 Linux

2.3.1 How to build a Linux Mint virtual machine with Virtual Box

The goal of this tutorial is to teach you how to build a Linux Mint virtual machine with Virtual Box.

Linux Mint is an easy to use variant of Linux that is derived from Ubuntu. The key advantage of Linux over Windows is the availability of numerous precompiled packages that can easily be installed using the Mint's package manager.

Virtual machines or VMs are virtual computers that run on top of the operating system of your physical computer. Among other things, virtual machines make it easy to run alternative operating systems or versions of operating systems on your computer. For example, virtual machines make it easy to use Linux on top of a Windows PC without having to create a dual boot, allowing you to use Linux and Windows simultaneously.

Virtual Box is a popular program developed by Oracle that can be used to run virtual machines.

Instructions

1. Download and install Virtual Box from https://www.virtualbox.org/wiki/Downloads
2. Download Mint Linux, Cinammon, 64-bit from https://linuxmint.com/download.php
3. Run Virtual Box
4. From the Virtual Box main menu, select "Machine" >> "New" and then follow the on screen instructions
    1. Enter a name, e.g. "Mint Linux", select Type: "Linux", and select Version: "Ubuntu (64-bit)"

2. Set the memory size to 4096 MB
3. Select the option to create a virtual hard disk
    1. Select the "VDI" format
    2. Select "Dynamically allocated",
    3. Set the size to 100 GB
    4. Click "create"
5. Highlight the new virtual machine in Virtual Box, right click on the machine, and select "Settings…".
6. In the window that opens
    1. In "General" >> "Advanced", set "Shared clipboard" to "Bidirectional"
    2. In "System" >> "Processor", set the number of processor to 50% of the maximum and enable "Enable PAE/NX"
    3. In "Display" >> "Screen", enable "Enable 3D video acceleration"
7. Highlight the new virtual machine in Virtual Box, right click on the machine, and select "Start" >> "Normal start".
8. In the window that opens
    1. Select the Mint Linux file that you download in step 2
    2. Click "Start"
9. After the temporary installation OS boots up, double click on the "Install Linux Mint" icon on the desktop
10. In the window that opens
    1. Select your language
    2. Enable "Install third-party software"
    3. Select "Erase disk and install Linux Mint" and select "Use LVM with the new Linux Mint installation"
    4. Select your location
    5. Select your keyboard layout
    6. Enter a user name, computer name, and password
    7. Allow Linux to complete the installation

Additional tutorials

There are many other more detailed tutorials on how to build Linux virtual machines

- How to install Linux Mint as a virtual machine using Windows
- Installing Ubuntu inside Windows using VirtualBox

2.3.2 How to build a Ubuntu Linux image with Docker

Docker images are configurable, free-standing computing environments that can be used to create and run custom software on top of an operating system. Unlike virtual machines (VM), images do not contain an operating system. Docker images are a convenient way to distribute complicated software programs that have numerous dependencies and complicated configurations. We are using Docker images because CircleCI allows users to use Docker images

to customize the environment used to execute each build. This makes it much easier to install programs into the environment used by CircleCI to run our builds.

Docker images are built by compiling Dockerfiles which are explicit instructions on how to build the image. Importantly, this makes Docker images very transparent.

Docker Hub is a cloud-based system for sharing a distributing Docker images. Docker Hub allows users to create image repositories, upload images, make image repositories public, and download images. CircleCI can build code using any image that is publicly available from Docker Hub.

Required packages

Execute the following commands to install and configure the packages required for this tutorial:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   xenial \
   stable"
sudo apt-get update
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    docker-ce \
    software-properties-common
sudo groupadd docker
    sudo usermod -aG docker $USER
    sudo systemctl enable docker
```

Next, logout and login again.

Run the following to check that you installed Docker correctly:

```
docker run hello-world
```

If you have network access errors, comment out *dns=dnsmasq* in `/etc/NetworkManager/NetworkManager.conf` and restart `network-manager` and `docker`:

```
sudo service network-manager restart
sudo service docker restart
```

Configuring a image

Docker uses Dockerfiles to configure images. These files contain several directives

- `FROM`: this describes the base (and its version) from which to build a image. For example, the value of this directive could be `ubuntu:latest` or a previous iteration of your image.
- `RUN`: these describe how to install software onto the machine. Because Docker creates layers for each RUN directive, you should use "&" to group related commands together and minimize the number of layers (thus disk space and bandwidth).
- `CMD`: this tells Docker what the final execution state of the image should be. Often this is set to `bash`.

See the Dockerfile reference and Docker explained for more information about Dockerfiles.

Building a image

Once you have configured the image, you can use `docker build` to compile the image:

```
docker build \
  --tag repository:tag \
  [/path/to/Dockerfile] \
  /path/to/context

docker build \
  --tag karrlab/wc_env_dependencies:latest \
  Dockerfile \
  .
```

If you do not specify a Docker file, then Docker will use the file located at ./Dockerfile. Optionally, you can also use `docker build` to tag the versions of images.

Uploading images to Docker Hub

Once you have built a image, you can upload it to Docker Hub

1. Create an account at Docker Hub
2. Login into https://hub.docker.com
3. Click on the "Create Repository" button
4. Follow the on-screen instructions
5. Use the Docker command line utility to log into Docker Hub:

   ```
   docker login
   ```

6. Push the image to the repository and optionally, tag the version of the uploaded image:

   ```
   docker push repository[:tag]
   ```

Listing existing images

You can list of all the images that are already available on your machine by running `docker images`.

Removing images

You can remove a image by running the `rmi` command:

```
docker rmi [repository:tag] [image_id]
```

Running an image

You can use the `run` command to run images:

```
docker run -it [repository:tag] [cmd]
```

If no command is provided, then Docker will run the final command in the image's configuration.

Running a Docker image instantiates a running environment called a container.

Any modifications made to the machine such as installed packages or saved files will not be discarded when the image terminates. When the image is booted up again, the image will start its execution from exactly the same state as the most recent execution of the image. This design forces you to use Docker files to explicitly describe image configurations.

2.3.3 An introduction to Linux Mint

Running command line programs

Linux makes it easy to use powerful command line programs. To run a command

1. From the Mint Menu, open the "Terminal" application
2. Type a command such as `uname` and then type enter

Note, while command is running you will not be able to run additional commands from that terminal. For example, if you run Firefox by executing `firefox`, you will not be able to execute additional commands until you close Firefox.

To run additional commands from the same terminal while other commands are running, you must execute commands in the background. This can be done by appending "&" to the end of the command. For example, `firefox &` can be used to run Firefox in the background.

Getting help for command line programs

Most command line program provide built in help. For example, these three commands can be used to get help information about Firefox:

```
man firefox
firefox -h
firefox --help
```

Installing, upgrading, and uninstalling software

Linux's package manager makes it very easy to install, upgrade, and uninstall packages (i.e. software). This is one of the key advantages of Linux over Windows.

Mint uses Synaptic to manage packages. To search for a package, first open the package manager by running `synaptic &` from the command line. In the window that opens, you can search and browse for packages by name and/or category. For example, to install the emacs text editor, search for "emacs", click the check box next to the emacs package, select "Mark for installation", accept the installation of emacs' dependencies, and finally click the "Apply" button to install emacs. To upgrade or uninstall emacs, search for "emacs", click the check box next to the emacs package, select "Mark for reinstall" or "Mark for uninstall", and finally click the "Apply" button.

Packages can also be installed, upgraded, and uninstalled from the command line using the "apt" command. For example, the following commands can be used to install, upgrade, and remove emacs:

```
apt-get install emacs
apt-get upgrade emacs
apt-get remove emcas
```

See also the Ubuntu documentation for more information about Synaptic and the Debian documentation for more information about Aptitude.

Additional tutorials

Numerous other Linux tutorials are available

- http://ryanstutorials.net/linuxtutorial
- http://linuxcommand.org/learning_the_shell.php
- https://www.edx.org/course/introduction-linux-linuxfoundationx-lfs101x-1
- https://www.udacity.com/course/linux-command-line-basics–ud595

2.4 Version and sharing data with Quilt

Conducting modeling reproducibly and collaboratively requires versioning and sharing data. Although Git/GitHub is well suited to versioning and sharing code and models, Git/GitHub is not well-suited to data because Git is based on line-by-line differencing of text files, because Git is designed for small files under 100 MB, and because Git requires the entire package and its history to be cloned. Quilt is a new system for versioning and sharing data with similar functionality to Git/GitHub and Docker/DockerHub.

2.4.1 Overview

Quilt is based on versioning packages of data, which are hierarhical trees of directories and files.

Quilt provides the following features:

- Capability to version data packages
- Capability to share packages with collaborators and with the world
- Programmatic access to upload, download, and update packages
- Web pages with READMEs to view and browse packages, including their histories

2.4.2 Using Quilt

See the Quilt documentation.

2.5 Scientific communication: papers, presentations, graphics

2.5.1 Writing manuscripts

Journal articles, or manuscripts, are the primary means by which scientific results are disseminated. As a result, your publication record will be one of the major components of your applications for new jobs and promotions. Consequently, it is important to publish your work in popular, well-read journals, and it is important to communicate your ideas clearly to gain acceptance to these journals. Below is an overview of the publication process and how to write, submit, and resubmit manuscripts for publication.

The publication process

Below are the typical steps in the publication process.

1. Complete a scientific project
2. Select a journal to submit a manuscript about the project to
3. Optionally, discuss the suitability of the proposed manuscript for publication in the selected journal
4. Write the manuscript, figures, tables, and supplementary materials, following the journals submission guidelines
5. Submit the manuscript to the journal via the journal's online submission system
    - Submit a cover letter to the journal's editors which introduces the manuscript
    - Submit a list of recommended reviewers
    - Submit the manuscript
6. The editor will then evaluate the suitability of the manuscript for the journal and either assign the manuscript to ~3 peer reviewers (other scientists in the field) or reject the manuscript
7. If the manuscript is assigned to reviewers, the reviewers will then review the manuscript and submit their recommendations and concerns to the editor
8. The editor will then collate the reviewers' concerns and decide whether to accept, potentially accept, or reject your manuscript
9. Manuscripts which are potentially accepted must be resubmitted along with responses to the reviewers' concerns (see below), which leads to another round of reviews by the reviewers and another decision by the editor.
10. Once a manuscript is accepted, the journal's production staff will provide you instructions on how to submit production quality files which they will use to format your manuscript for publication.

How to write a manuscript

The keys to writing good manuscripts are to describe why your work is significant (i.e. how it solves and important problems), to describe how your work is novel (i.e. how it is different from previous work), and to clearly present your ideas so they can easily be understood with minimal effort.

Often these goals are best accomplished by organizing the body of your manuscript into the following sections

- Introduction: describe the problem that you've tried to solve, why it is important, the prior work that has been done on the problem, how you've approached the problem in a novel way, and provide a brief overview of the methods and results of the project
- Methods: describe how you've solved the problem
- Results: describe the results of your work
- Discussion: describe the implications of the results of your work
- Conclusion: re-summarize the manuscript
- Supplementary materials: Supplementary materials should communicate all of the additional information needed to understand your project that cannot fit in page limit for your manuscript. This should include all of the data, code, and intermediate results needed to understand and reproduce the work described in the manuscript. Supplementary materials can either submitted directly to the journal or they can be submitted to public repositories and linked to the manuscript.

How to write an abstract

Abstracts should follow this hour-glass structure

1. Introduce the broader problem that you have tried to solve and explain why it is significant
2. Describe the specific problem you have tried to solve
3. Explain why the problem has not been solved previously
4. Describe how you innovatively solved the problem
5. Describe the results of your work
6. Describe the broader significance of your results

Abstracts are generally limited to 200-250 words.

See Nature's annotated abstract for more information about how to write an abstract for a manuscript.

How to format a manuscript for submission

Below are general guidelines for formatting manuscripts for submission to journals. Before submitting to a journal, also review the journal's submission guidelines.

- Layout
  - Title, authors, and affiliations
  - Abstract
  - Body of the manuscript
  - Acknowledgements
  - References
  - Figure and table captions
  - Figures, 1 per page
  - Tables
  - Supplementary content
- Citation format: consult the journal's guidelines
- Paper size: 8.5 x 11 in
- Margins: 1 in
- Font
  - 12pt
  - Serif, Times New Roman (Word) or Computer Modern (LaTeX)
- Line spacing: doubling
- Line numbers: display in left margin
- For resubmissions, use yellow highlighting to mark all major changes
- Filetype: .docx, .tex, or .pdf
- Figures:
  - Size: Below are the sizes for *Cell*. Consult your journal's guidelines.

        - Full width: 174 mm
        - 1.5 width: 114 mm
        - Half width: 85 mm
    - Font:
        - Subfigure labels: 8pt, bold
        - Other text: 5-7pt
        - Sans-serif, Arial
    - Color: RGB
    - Filetype: .eps, .pdf, or .ai

How to write a response to reviewer critiques

- As much as possible, try to address the manuscript to address the reviewers' concerns. Clearly explain which concerns you do not agree with and why.
- Begin the response with a summary of the major changes you have made in response to the reviewers concerns.
    - Thank the reviewers for their suggestions.
    - Discuss how you've improved the manuscript based on their suggestions or why you believe their suggestions should not be incorporated into the manuscript.
- Then provide point-by-point responses to each concern raised by the reviewers
    - Intersperse your responses with the reviewers' concerns.
    - Thank the reviewers for their suggestions.
    - Discuss how you've improved the manuscript based on their suggestions or why you believe their suggestions should not be incorporated into the manuscript.
- Format
    - Paper size: 8.5 x 11 in
    - Margin: 1 in
    - Font: 12pt, Serif
    - Color the reviewers' concerns blue so they can easily be differentiated from your responses.
    - Filetype: .docx or .pdf

Below are several examples of responses to reviewer critiques.

- http://journals.lww.com/greenjournal/Documents/SampleResponseToRevisions.pdf
- http://perso.citi.insa-lyon.fr/rstanica/reviews/answers1.pdf
- https://www.d.umn.edu/~jetterso/documents/Anexampleofresponsetoreviewers.doc

2.5.2 Reviewing manuscripts

As a reviewer, your job is to provide the editor expert advice about the significance and novelty of a manuscript, the correctness (e.g. appropriate methods, controls, statistical analyses) of the work described in the manuscript, and the suitability of the manuscript for the journal (i.e. does the topic of the manuscript fit with that of the journal). The editor will typically solicit 3-5 reviews and then make a final decision based on their own opinion and that of the reviewers.

This is accomplished by reading the manuscript, noting anything that is wrong or missing from the manuscript, and finally writing a written recommendation for the reviewer. In reviewing manuscripts, it is tempting to focus only on the negative aspects of manuscripts. However, you should also try to focus on important strengths of manuscripts and champion manuscripts that you believe would be significant contributions to the scientific literature.

Becoming a reviewer

Once you've published several papers, authors will begin to recommend you as a reviewer and editors will start to invite you to review manuscripts. You can also solicit invitations to review manuscripts, by contacting journal editors with a description of your scientific background and interest in reviewing papers for their journals. This will be most effective when you contact editors which are responsible for field. Browser journal websites to find lists of their editors.

Timeline

Editors generally give reviewers 2-4 weeks to review manuscripts. If you accept an invitation to review a manuscript, it is important to submit the review on time so that the editor can make a final decision in a timely manner.

Format

1. Summary of the manuscript, including what the authors did and why its important and novel
2. Summary of major criticisms
3. List of major criticisms
4. List of minor criticisms

See this example for more information.

More information

There are numerous articles on how to review manuscripts

- Step by step guide to reviewing a manuscript
- How to peer review a manuscript
- A Systematic Guide to Reviewing a Manuscript

2.5.3 Making posters

Abstracts

Conference abstracts often follow the same hourglass structure as paper abstracts (see Section 2.5.1). However, there are a few salient differences between paper and conference abstracts.

- Conference abstracts can often be longer than paper abstracts. As a result, a conference abstract can contain more details about your methods and results than a paper abstract.
- Unlike paper abstracts, conference abstracts can be written from the perspective of the anticipated state of your project at the time of the conference. In particular, conference abstracts can describe ongoing work which papers typically do not discuss.

- Oral and poster conference presentations rely heavily on the individual presenter. Accordingly, conference abstracts can be written from the first-person singular.
- Abstracts for oral presentations should summarize the material that the presenter will tell the audience.
- Abstracts for oral presentations can focus on the presenter's body of work rather than a specific project.

Content

Because posters provide limited space for text, figures, and tables, posters should focus on the most essential aspects of your work. It is not necessary to provide all of the details of your work. Because you will be standing next to your poster, you will be able to communicate directly with readers and verbally answer any questions they have. Furthermore, you can refer readers to papers and websites which contain additional information about your project.

Layout

Often it is helpful to divide your poster into blocks or columns. Often it is helpful to layout the content of the blocks in this order

- Title
- Authors
- Affiliations
- Abstract
- Overview of the problem you're solving and why it's significant
- Methodology
- Results
- Future work
- Conclusion
- Acknowledgements
- References

Formatting

- Size: 1 x 1 m
- Font family: San-serif, Arial
- **Font sizes:**
  - Title: 60 pt
  - Heading: 36 pt
  - Text: 24 pt

Software tools

Illustrator is one of the best tools for making scientific posters. Inkscape can also be used to make posters. PowerPoint can also be used to make posters, but PowerPoint does not provide users as much control over the appearance of graphics as Illustrator.

2.5.4 Making presentations

Conference presentations and seminars are excellent opportunities to get other scientists excited about your work, including seeing your work funded and published. You can maximize this opportunity by structuring your presentation and each individual slide to captivate your audience.

- Communicate your work through as story
- Communicate your ideas succinctly
- Use captivating graphics and clear examples
- Avoid domain-specific jargon

Presentation structure

Effective presentations often follow the same hour-glass structure as abstracts and papers (see the Section 2.5.1). However, there are few salient differences between presentations and papers

- Presentations should focus on your broader body of work rather than one specific project
- Presentations should emphasize your own contributions to the work and your own opinions
- Presentations should provide an overview of your work and not try to present every detail
- Presentations can provide more elaboration on your ongoing and future work
- Presentations should be more personal than manuscripts to connect with the audience. Often, it can be effective to structure a presentation like a story that you are telling the audience.

Slide design

Below are several guidelines for designing effective slides

- Each slide should focus on 1 message
- Use simple diagrams
- Use high-contrast colors so your slides are viewable even with dim projectors
- Use sufficiently large font size so that text is readable from a distance
- Avoid large amounts of text
- Avoid placing content at the margins in case the projector is misaligned
- Avoid complex transitions and reveals

Software tools

PowerPoint and KeyNote are two of the best software tools for designing presentations. LaTeX's beamer package can also be used to make nice presentations, but it is hard to layout graphics nicely in LaTeX. Prezi can also make nice presentations, but they Prezi encourages extraneous, distracting transitions. Prezi is better suited to recorded video webinars.

Further information

There are many additional resources on how to give scientific presentations

- Designing effective scientific presentations
- The secret structure of great talks
- Duarte Design's five rules for presentations
- Designing PowerPoint slides for a scientific presentation

2.5.5 Visualizing data

- Leverage the human visual system's abilities to process visual information
- Use easily understandable data encodings
  - Utilize position and length
  - Use a small number of easily discriminable colors
  - Avoid 3D and animation
- Display data in its context
  - Use appropriate scales
  - Compare
  - Lay data out on familiar maps such as geographical and pathway maps
- Use small multiples to visualize additional dimensions
- Avoid red/green color palettes to accommodate colorblindness
- Avoid distracting viewers with unnecessary data and other unnecessary visual marks

Interactive exploratory data visualization

Static visualizations are helpful to depicting data. However, static visualizations are generally limited to a few dimensions. Consequently, static visualizations can generally only depict small fraction of large data sets. Alternatively, interactive data visualizations can enable exploration of larger and higher dimensional datasets. See d3js.org for inspiring examples of interactive data visualizations, The major disadvantages of interactive data visualization are that are more complex and take more time to create.

Software tools

Below are several recommend tools for creating data visualizations:

- Exploring data
  - Tableau
- Creating specific plots
  - Python and matplotlib: useful for plotting data
- Combining plots into figures
  - Illustrator
  - Inkscape

- Creating interactive data visualizations
  - JavaScript and D3.js
  - ipyvega

Exercises

- Use matplotlib to create a static visualization
- Use illustrator to combine multiple static visualizations
- Use ipyvega to create an interactive visualization

Further information

- Data visualization course by Jeff Heer
- The Visual Display of Quantitative Information by Edward Tufte

2.5.6 Formatting textual documents with LaTeX

LaTeX is a powerful tool for generating beautiful textual documents. LaTeX is particularly useful for generating textual documents that require large amounts of math, algorithms, and references. Parts of LaTeX's power comes from the large number of additional packages that are available from CTAN, the LaTeX package repository.

Required software

Run the following commands to install the software required for this tutorial:

```
sudo apt-get install texlive
```

Tutorial

Below are several LaTeX tutorials

- A beginner's guide to LaTeX
- LaTeX tutorial
- LaTeX video tutorials

Online, collaborative LaTeX editing

Recently, several companies have developed good tools for collaboratively editing LaTeX documents in real-time in the cloud. One of the best of these tools is Overleaf. Overleaf merges some of the best features of LaTeX, Google Drive, and Git.

2.5.7 Drawing vector graphics with Adobe Illustrator and Inkscape

Broadly, there are two types of graphics: vector graphics and raster graphics. Vector graphics are combinations of elemental graphical objects such as lines, curves, arcs, ellipses, text, strokes, fills, and gradients. Consequently, vector graphics are vector graphics are infinitely scalable (i.e. they never become pixelated when they are enlarged) and well-suited for scientific diagrams. Two of the most common vector graphic editing programs are Illustrator and Inkscape. Some of the most common file formats used to store vector graphics include .ai, .eps, .pdf, and .svg.

Raster graphics are matrices of colored pixels. Raster graphics are commonly used to describe photographs as well as to display graphics originally created as vector graphics in websites and other documents. Some of the most common raster graphic editing programs are Photoshop and Gimp. Photoshop is a commercial program developed by Adobe and Gimp is a free, open-source program. Photoshop is the standard among graphics professionals, but Gimp is easier to learn and sufficient for our needs as scientists. Some of the most common file formats used to store raster graphics are .gif, .jpg, and .png format.

In this tutorial, we will teach you how to draw vector graphics with Illustrator and Inkscape by drawing a picture of a cell.

Key concepts

- Raster graphics vs. Vector graphics
- Fundamental vector graphic objects
- Color models
- Selecting and editing graphical objects
- Grouping objects
- Masking objects
- Exporting diagrams

Fundamental vector graphic objects

As introduced above, vector graphics are composed of the following elemental graphical objects:

- Paths
  - Bezier curves
  - Lines
- Shapes
  - Circles
  - Ellipses
  - Polygons
  - Rectangles
- Text
- Fills and strokes
  - Solid
  - Gradient
- Groups

- Masks
- Layers

Color models

Vector graphics can be described using one of several common color models:

- CMYK (Cyan, Magenta, Yellow, Black): CMYK is a subtractive color model which represents colors as tuple of cyan, magenta, yellow, and black values, each of which ranges from 0 to 100%. CMYK is the native color model for many printers. For this reason, some journals prefer CMYK.
- RGB (Red, Green, Blue): RGB is an additive color model which represents colors as tuples of red, green, and blue values, each of which ranges from 0 to 255. RGB is the native color model for many monitors. For this reason, RGB is most common color model for web-based graphics and some journals prefer RGB.
- HSB (Hue, Saturation, Brightness) / HSV (Hue, Saturation, Value): HSB is a cylindrical color model which makes it easy to manipulate saturation and brightness separately from color. For this reason, HSB is useful to designing color palettes.
- Grayscale: is an additive color model which represents colors as integers between 0 and 255.

Illustrator supports all four of these color models, and documents created in any one of these color models can be exported to any of the other color models. Inkscape only supports the RGB color model.

We recommend using RGB except for submissions to journals which require CMYK figures.

Vector graphics drawing tools: Illustrator vs Inkscape

Illustrator and Inkscape are two of the most popular vector graphics editing programs. Illustrator is a commercial program developed by Adobe and Inkscape is a free, open-source program. Illustrator is the industry standard and is more powerful than Inkscape.

Illustrator and Inkscape share many of the same core functionality. However, Inkscape lacks many of the more advanced features of Illustrator. In addition, Illustrator is substantially faster than Inkscape, especially for large graphics. This is critical for editing complex plots generated by graphing utilities such as matplotlib. Furthermore, there are substantially more tutorials and examples for Illustrator than for Inkscape. However, some people find Inkscape easier to learn. Below is a list of some of the limitations of Inkscape compared to Illustrator.

- Significantly slower for complex graphics
- Interface is less polished
- Poor default options for subscripts, superscripts
- No linked text boxes for continued text flow
- No tool to edit spacing of dashes
- No CMYK support
- No native support for a binary file format such as .ai
- Less control over .png export
- No .gif, .jpg export

Vector graphics file formats

Some of the most common file formats used to represent vector graphics include AI, EPS, PDF, and SVG. AI is a proprietary and efficient binary format that is used by Illustrator. SVG (Scalable Vector Graphics) is an XML-based open standard for describing vector graphics and the native format for Inkscape. Because SVG is an XML format, SVG is easy to generate programmatically. However, because SVG is an XML format, SVG is also inefficient and poorly suited to large graphics such as complex plots created by matplotlib. EPS and PDF are binary vectors graphics formats that are supported by both Illustrator and Inkscape. Consequently, these formats should be used to provide graphics to journals.

Required software

This tutorial requires Illustrator and Inkscape.

On Ubuntu, you can use this command to install Inkscape:

```
sudo apt-get install inkscape
```

Illustrator exercise

In this exercise we will learn how to use Illustrate by drawing a digram of a cell

1. Open Illustrator
2. Set the canvas units size
3. Use the ellipsis tool to draw a cell
4. Set the stroke and fill color
5. Add a drop shadow
6. Add a straight line into the cell
7. Add an arrow head to the line
8. Turn the line into a curve
9. Copy the line to create a line out of the cell
10. Add a textual label on top of the cell
11. Create a mask to highlight the area that you want to highlight
    1. Draw a rectangle over the area you want to highlight
    2. Select the rectangle and all of the graphical element below it
    3. Create the mask
12. Save the diagram in AI format
13. Save the diagram to PDF format to use in manuscripts
14. Export the diagram to PNG format to use in PowerPoint and websites

Other useful features

- Selecting similar objects
- Joining lines
- Placing other documents

Screen capture

Open the screen capture in a separate page

Inkscape exercise

In this exercise we will learn how to use Inkscape by drawing a digram of a cell

1. Open Inkscape
2. Set the size of the canvas
    1. Open "File" >> "Document Properties..."
    2. Set "Units" to "in"
    3. Set "Width" to "7.5"
    4. Set "Height" to "5"
    5. Close the window
    6. Type "5" to fit the canvas to your screen
3. Draw the cell membrane
    1. Select the ellipse tool
    2. Drag an ellipse over the canvas
    3. Right click on the ellipse and select "Fill and Stroke..." to edit the line and fill colors and line style of the membrane.
        1. Increase the stroke width of the membrane
        2. Change the stroke style of the membrane to dashed
        3. Apply a radial gradient fill to the body of the membrane
        4. Adjust the center and shape of the radial gradient
    4. Add a drop shadow to the cell by selecting "Filters" >> "Shadows and Glows" >> "Drop Shadow..."
4. Draw an arrow into the cell
    1. Select the Bezier curves tool
    2. Select one or more points on the canvas. Optionally, hold down the control key to draw a straight line.
    3. Double click to finish the curve
    4. Optionally, use the "Align and Distribute" tool to straighten the line
    5. Use the edit path tools to fine tune the curve
    6. Right click on the line and select "Fill and Stroke..." >> "Stroke style" to apply arrow markers to the line

5. Create an arrow which points out of the cell by copying the first arrow
    1. Left click on the first arrow and hold down
    2. While still holding down the left mouse button, click the space bar
    3. Begin dragging your mouse
    4. Press down the control key to constraint the dragging so that the second arrow is vertically aligned with the first
6. Vertically align the cell and lines
    1. Open the "Align and distribute objects" window
    2. Select both the cell and line
    3. Click the "Center on horizontal axis" button to align the objects
7. Add a label to the cell
    1. Select the text tool
    2. Click on the canvas where you want the text to appear
    3. Type "Cell"
    4. Right click on the label and select "Text and Font..." to adjust the font type, font size, text color, and text alignment
    5. Use the dropper tool to copy the cell stroke color to the text
    6. Bring the text in front of the cell by selecting "Object" >> "Raise to Top"
8. Group the cell and label
    1. Select the cell and label
    2. Select "Object" >> "Group"
    3. Now you can move the objects together
    4. Double click on the combine object to access the individual cell and label objects
9. Highlight a specific part of the cell
    1. Draw a rectangle over the portion of the cell that you would like to highlight
    2. Select both this new rectangle and the cell
    3. Right click on the objects and select "Set Mask"
10. Save the diagram
    1. Select "File" >> "Save"
    2. For simple graphics, choose the "Inkscape SVG (.svg)" format. For complex graphics, choose the "Compressed Inkscape SVG (.svgz)" format.
11. Export the diagram
    1. Select "File" >> "Export PNG Image..."
    2. Set the desired export size and resolution

Other useful features

Selecting other objects with the same fill and/or stroke

- Select an object
- Right click on the object and select “Select Same” >> “Fill and Stroke”

Joining lines

- Use the “Edit paths by node” to select a node in a curve
- Hold to the shift key and select another node in another curve
- Click the “Join selected nodes” button to join the curves

Embedding graphics

- Select “File” >> “Import”
- Select the file that you wish to import
- Select whether to “Embed” or “Link” the imported file

Additional tutorials

Illustrator

Kevin Bonham has several helpful tutorials videos designed for scientists. Skill Developer also has a large number of brief tutorial videos.

Inkscape

Derek Banas has several helpful short tutorial videos. The Inkscape Tutorials Blog has numerous examples of how to draw a variety of graphics.

2.5.8 Editing raster graphics with Gimp

As described in Section 2.5.7, there are two types of graphics: vector graphics (generally diagrams in .ai, .eps, .pdf, or .svg format) and raster graphics (generally photos in .gif, .jpg, or .png format). This tutorial will teach you how to edit raster graphics with Gimp, an open-source raster graphic editing program, by creating a head shot for a website.

Concepts

The exercise below will introduce you to the following key concepts of raster graphic editing:

- Zooming
- Adding layers
- Selecting regions

- Expanding selections
- Clearing selections
- Filling selections
- Changing the color mode
- Cropping images
- Rescaling images

Required software

This tutorial requires Gimp.

On Ubuntu, Gimp can be installed by running this command:

```
sudo apt-get install gimp
```

Exercise

In this exercise, we will learn how to edit raster graphics by creating a head shot of President Obama for a website. This will include selecting the face, removing the background, cropping the image, resizing the image, and exporting the image for fast.

1. Download a photo of `President Obama`
2. Open Gimp
3. Open the photo
4. Add an alpha channel to enable a transparent background ("Layer" >> "Transparency" >> "Add a alpha channel")
5. Use the Intelligent scissors select tool ("Tools" >> "Selection Tools" >> "Intelligent scissors") to select only the face. Zoom in to select the face precisely. Note, Gimp provides several additional selection tools including tools to select rectangles and ovals and to select regions by color.
6. Type enter to accept the selection
7. Feather the selection so that the transition between the face and background is not so abrupt when we cut out the background ("Select" >> "Feather...")
8. Select the background by inverting the selection ("Select" >> "Invert")
9. Delete the background ("Edit" >> "Clear")
10. Change the color mode to gray scale ("Image" >> "Mode" >> Select "Grayscale")
11. Crop the head and make the image square ("Tools" >> "Transform Tools" >> "Crop").
    1. Disable "Current layer only"
    2. Drag a rectangle over the photo
    3. In the "Tool Options" pane,
        1. Set "Size" >> 10, 10
        2. Enable "Fixed" >> "Aspect ratio"
    4. Adjust the size and shape of the selection box

5. Type enter to crop the image

12. Scale the image (“Image” >> “Scale Image…”)
13. Export the image (“File” >> “Export As …”)

Screen capture

Open the screen capture in a separate page

Additional tutorials

Learn GIMP has several helpful short tutorial videos.

CHAPTER 3

Fundamentals of cell modeling

3.1 Data aggregation

Building dynamical models of individual cells requires multivariate temporal data that captures the dynamics and variation of individual cells to inform the structure of the model and the value of each parameter. Extensive data is available to build cell models. However, this data is highly heterogeneous and incomplete because there is no single technology that can capture the high-dimensional dynamics and variation of individual cells. Furthermore, this data is highly dispersed across a large number of databases and individual manuscripts. Thus, a major challenge in cell modeling is to aggregate and merge this data into a consistent understanding of cell biology. In particular, we must merge data that is incomplete, that was obtained using multiple measurement methods for multiple genetic and environmental conditions and species and measured with different granularities, that includes relatively little dynamic data, and that includes relatively little single-cell data.

Broadly, there are two types of data that can be used to build cell models: experimental observations and prediction tools. The major advantages of prediction tools are that they provide complete data and that they can provide data for the exact genetics and environment that you wish to model. However, only a limited number of prediction tools are available.

Broadly, there are two types of data sources that can be used to build cell models: individual manuscripts and databases. The major advantage of databases is that other researchers have already done much of the time-consuming and tedious work needed to extract data from manuscripts. Thus, databases can save you a lot of time. However, there are many types of data that are not contained in databases and the most databases do not contain all of the contextual information (e.g. environmental condition of the measurement) that is needed to interpret their data.

The goal of this tutorial is to familiarize you with some of the most important data types and data sources for cell modeling.

3.1.1 Common data types and data sources

- Metabolites
  - Structures: Mass-spectrometry; ChEBI, PubChem
  - Concentrations: Mass-spectrometry; ECMDB, YMDB, HMDB

- RNA
  - Sequences: Sequencing; GenBank
  - Modifications: Sequencing, mass-spectrometry: Modomics
  - Concentrations: Microarray, RNA-seq; Array Express, GEO
  - Localization: FISH, microscopy; individual papers
  - Half-lives: Microarray, RNA-seq; individual papers
- Proteins
  - Sequences: UniProt
  - Modifications: Mass-spectrometry; UniMod
  - 3D structures: X-ray crystallography, NMR; PDB
  - Complexes: chromatography, yeast 2 hybrid; EcoCyc, UniProt
  - Localization: microscopy; PSORT, WolfSort
  - Concentrations: mass-spectrometry; Pax-DB
  - Half-lives: mass-spectrometry; individual papers
- Interactions
  - DNA-Protein: ChIP-seq; DBD, DBTBS
  - Protein-metabolite: DrugBank, STICH, SuperTarget, UniProt
  - Protein-Protein: BioGRID, DIP, IntAct, STRING
- Reactions:
  - Catalysis: UniProt
  - Stoichiometry: BioCyc, KEGG
  - Kinetics: BRENDA, SABIO-RK, BioNumbers

3.1.2 Finding data sources

There are several meta-databases which contain lists of data sources that are helpful for finding appropriate data source

- BioCatalogue
- BioMart
- BioMoby
- BioSWR
- ELIXIR
- NAR Database Summary

3.1.3 Finding relevant data for models

When you aggregate data to inform a model, it is important to aggregate data that is relevant to the model. Where possible try to find data that was observed for

- Taxonomically close organisms

- Similar genetic variants
- Chemically similar species and reactions
- Similar environmental conditions: temperature, pH, media, pressure

3.1.4 Data aggregation tools

BIOSERVICES is a helpful tool for aggregating data from approximately 25 of the largest molecular biology databases. However, BIOSERVICES only supports a few databases and provides minimal support for identifying relevant data for a model. Unfortunately, there are few tools for aggregating relevant data for models. Consequently, we must develop better tools for aggregating the data needed to build models.

3.1.5 Determining the consensus of multiple observations

In some cases, you may be lucky enough to find multiple observations to estimate a parameter value. In this case, we recommend estimating the parameter value by calculating the mean of the individual observations weighted by their relevance (taxonomic distance, species/reaction similarity, environmental similarity, etc.) to the model.

3.1.6 Exercise

1. Download a model from BioModels
2. Ignore the provided parameter values
3. Use the databases and prediction tools listed above to estimate the values of all of the parameters of the model
    - Determine was metadata is provided about the observed organism, genetic, environmental conditions, etc.
    - Try to identify data that were observed under similar conditions to the model
4. Track the provenance of each value that you identify
    - Observed value and uncertainty
    - Observed units
    - Observed species
    - Observed condition
    - Measurement method
    - Reference

3.2 Input data organization

Building large models requires a large amount of input data to inform the structure of the model and the value of each parameter. Consequently, it is helpful to organize this data into a readily understandable and computable database.

Unfortunately, there are few tools specifically designed to organize the input data needed for mechanistic models. However, Pathway/genome databases (PGDBs) or model organism databases (MOD) are conceptually similar and the existing PGDB tools provide much of the functionality needed to organize the input data for mechanistic models. In fact, the Pathway Tools PDB tool includes a module called MetaFlux which can be used to build flux balance analysis models of metabolism. In particular, PGDBs can track detailed molecular information at the genomic-scale for individual organisms. Some of the major limitations of the existing PGDBs are that they provide limited support for non-metabolic pathways and that they provide limited support for quantitative data.

3.2.1 Schema

The input data used to build models can be organized with the following schema

- Value
- Uncertainty
- Units
- Genetic conditions
  - Taxon
  - Variant
- Environmental conditions
  - Temperature
  - pH
  - Media
- Localization
  - Intracellular compartment
  - Tissue
- Timepoint
  - Cell cycle phase
  - Growth phase
  - Time post-perturbation
- Measurement method
  - Parameters
  - Version
- Experiment: collection of values observed in the same experiment
- Reference

Several ontologies such as the CCO and CL can be used to describe components of the schema.

3.2.2 Software tools

Below are some of the best tools for organizing the input data used to build models. Unfortunately, all of these tools have significant limitations. Consequently, we must develop better tools for organizing the input data used to build models.

- GMOD
- Pathway Tools
- SEEK
- WholeCellKB

3.2.3 Exercises

EcoCyc and Pathway Tools

1. Browse the webpages of BioCyc
2. Observe the types of data EcoCyc contains and how it is organized
3. Read the schema documentation

WholeCellKB

1. Browse the webpages of WholeCellKB
2. Observe the types of data WholeCellKB contains and how it is organized
3. Read the schema documentation

3.3 Model design

Once you have aggregated data to build a model and organized this data into a computable form, the next step of the modeling process is to design your model. The goal of model design is to select the most likely model for a system given everything you know about the system. To avoid overfitting, this selection can be done using cross validation and/or by penalizing larger models to preferentially choose smaller, more parsimonious models. However, it is often difficult to formalize all of your prior knowledge and your confidence in that knowledge and there is often insufficient data to utilize cross validation.

Broadly, there are two families of approaches to designing models: data-driven model design and expert-driven model design. Data-driven model design is a formal mathematical approach to model design that tries to identify the most likely model of a system given prior information about that system. The advantages of data-driven model design are that this approach is rigorous, automated, unbiased, and scalable to large models. However, this approach requires large amounts of data, typically far more than is available and this approach does not leverage heterogeneous prior information effectively. The advantage of expert-driven model design is that it leverages heterogeneous prior information effectively, including information that has not been described formally. However, expert-driven model design can be time-consuming and often leads to models that are biased toward the modeler's preconceptions.

Due to the limitations of data-driven and expert-driven model design, several groups have developed hybrid model design approaches which automatically generate model design suggestions for modelers to review and accept or reject. For example, Henry et al. developed Model SEED to automatically seed expert-driven FBA metabolism models from models of related organisms, Latendresse et al. developed MetaFlux to seed expert-designed FBA metabolism models from PGDBs, and Kumar et al. developed GapFind to highlight gaps in expert-designed models.

Given that we do not yet have sufficient data to learn models, we must scale expert-driven model building to large models by decomposing models into multiple submodels, programmatically building models from structured sources of prior information, and automatically.

3.3.1 Software tools

Below are some of the most commonly used model design tools

- Automated model design
  - bnlearn
  - pgmpy

    - scikit-learn
- Manual model design
    - Cell Collective: online, collaborative environment for building logical models
    - CellDesigner
    - COPASI
    - JWS Online
    - PhysioDesigner
    - VirtualCell
- Programmatic model design
    - MetaFlux: tool for designing FBA metabolism models for PGDBs
    - PySB
- Hybrid model design
    - KBase
    - Model SEED
    - RAVEN

3.3.2 Exercises

Required software

- COPASI
- Pathway Tools
- Python
- Pip
- Pip packages
    - pgmpy
    - scikit-learn

Expert-driven model design with COPASI

1. Select a model from BioModels
2. Use the COPASI GUI to recreate your chosen model

PGDB-driven model design with MetaFlux

1. Download and install Pathway Tools which contains MetaFlux
2. Download EcoCyc
3. Follow the MetaFlux tutorial and use MetaFlux to construct an FBA model of *Escherichia coli*

Formal model selection

See the sckit-learn tutorial on model selection.

Bayesian network structure learning

See the pgmpy tutorial on learning Bayesian networks.

3.4 Model calibration

Once you have designed the structure of your model, the next step of the modeling process is to calibrate the model to identify/estimate the values of its parameters. Formally, models can be calibrated by identifying the set of parameter values which minimizes the difference between experimental observations and the model's predictions.

Numerous methods have been developed to optimize arbitrary mathematical functions. This includes simple methods such as gradient descent for identifying the optimum of convex functions and scatter search methods for identifying local optima of non-convex problems. Nevertheless, model calibration remains a challenging problem because this often requires optimizing flat high-dimensional non-convex functions, especially to calibrate computationally-expensive, large models.

The goal of this tutorial is to introduce the fundamentals of model calibration.

3.4.1 Key concepts

- Model calibration / model identification / parameter estimation
- Parameter identifiability
- Model calibration formalisms
  - Least squares
  - Maximum likelihood
- Model reduction
- Numerical optimization
  - Gradient descent
  - Scatter search

3.4.2 Calibration data

Before discussing how to calibrate single-cell models, it is important to note the nature of the data available to calibrate cell models. First, the majority of the data available to calibrate single-cell models directly relate to just one or a few model parameters. This means that most model parameters can be estimated from just a small number of values and, generally, only a few model parameters have to be estimated from physiological data.

Second, although cell models are stochastic and dynamic, most of the data available to build cell models represents the time- and population-average of large groups of cells. Consequently, cell models can be calibrated by calibrating their mean behavior which is significantly less complex and computationally-cheaper problem.

3.4.3 Approximate, multi-stage parameter estimation

In general, calibrating high-dimensional models is computationally-expensive. To reduce the computational cost of parameter estimation, we recommend the following approximate model calibration approach.

1. Do not design parameters whose values have not been measured. Instead, describe that biology qualitatively using fewer parameters.
2. Leverage genomic data to estimate the value of each individual parameter.
3. Generate a reduced model of the time and population average of the full model, decompose this reduced model into reduced pathway submodels, and iteratively optimize the parameters of these reduced submodels until convergence to estimate the joint values of the parameters. The first round of this optimization should be initialized with the parameter values calculated in Step 2.
4. Numerically optimize the parameter values of the full model, starting from the parameter values estimated in Step 3. The primary goal of this final step is to identify parameters who behavior is not adequately captured by the reduced submodels.

Univariate parameter estimates

The first step of this approach is to estimate the value of each individual parameter using the small number of molecular measurements which relate directly to each parameter. For example, we can estimate the transcription initiation rate of each RNA transcript by the ratio of its expression and half-life, each measured by microarray of RNA-seq.

Pathway joint parameter estimates

The second step of this approach to generate a set of reduced pathway submodels to estimate the joint values of the parameters, seeded by the univariate parameter estimates obtained in the previous step.

Model reduction

Because most of the data available to build cells models represents the time- and population-average of large groups of cells, we can calibrate cell models by calibrating reduced models which represent their time- and population average. Such reduced models have the parameters as the corresponding full dynamical model, but they have no time or single-cell dimension.

Broadly, there are two ways to generate such reduced approximate models. (1) We can computationally learn a reduced model by simulating the full model and fitting a function to the results or (2) we can manually reduce the full model by analytically estimating is mean behavior. The latter approach is more biased, but can requires substantially less numerical simulation.

Model decomposition

We can further reduce the dimensionality of model calibration by approximating the reduced model as a set of independent pathway submodels and iteratively calibrating these reduced submodels until convergence. Formally, these submodels can be generated using graph clustering. Alternatively, modelers can be asked to manually define these submodels.

Numerical optimization

Once we have generated these reduced model, we can use one of several numerical optimization methods and software tools to estimate the joint values of the parameters of each of the submodels.

- AMIGO2
- COPASI
- MEIGO
- saCeSS

Global joint parameter estimates

Once we have estimated the joint values of the parameters of each of the submodels, we can use the same numerical optimization algorithms outlined above to estimate the joint values of the parameters across the entire model. The main goal of this final step is to estimate parameters who behavior is not adequately captured by the reduced pathway submodels.

3.4.4 Exercise

- Obtain BioPreDyn-Bench, a suite of benchmark model calibration problems
- Use AMIGO and COPASI to calibrate the benchmark *Escherichia coli* model

3.5 Model representation

There are multiple ways to represent a model.

3.5.1 Custom numerical simulation code

At the lowest level, a model can be represented as a numerical simulation algorithm. For example, a stochastic model could be represented in Python code which implements the Gillespie algorithm. This approach provides a modeler with the most control over the numerical simulation of the model which can be helpful for efficiently simulating large models, but this approach leads to models that are difficult for other scientists to understand because they may need to read a large amount of code. Furthermore, because this approach has the disadvantage that it requires the modeler devote significant time writing and testing a large amount of code.

Exercise

Write a program which implements the Gillespie algorithm and uses this to simulate a stochastic model.

3.5.2 Standard numerical simulation packages

Alternatively, models can be described more abstractly using a modeling language and then simulating using a simulator which is able to interpret that language. Continuing with our stochastic simulation example, one slightly more abstract way to represent a stochastic model is as a function which calculates the rate of each reaction and a vector of initial conditions. Models represented in this form can then be simulated with the StochPy package. Compared to the lowest level of representation, this approach requires significantly less code and less unit testing, freeing the modeler

to focus more of their efforts on building models and using them to discovery biology. However, models described this way cannot easily be simulated by other simulators and this representation obscures the biological semantics of models.

Exercise

Write a program that uses StochPy to simulate a stochastic model.

3.5.3 Enumerated modeling languages

Alternatively, models can be described using domain-specific modeling languages such as CellML and SBML (Systems Biology Markup Language) that are supported by many programs and that explicitly capture the biological semantics of models. SBML models are composed of the elements listed below, and the SBML specification precisely defines how to simulate models encoded in SBML which can be accomplished by compiling models to a lower level.

- Compartments
- Species
- Reactions
- Kinetic laws
- Parameters
- Rules
- Constraints
- Events
- Function definitions
- Unit definitions
- Annotations

This approach creates models that are much more comprehensible than models that are described directly in terms of their numerical integration. However, for large models this approach can require intractably many species and reactions to represent all possible states and reactions. Furthermore, this enumerated representation becomes highly inefficient when the state space gets very large.

Exercise

Encode the same stochastic model in SBML and simulate the model using COPASI.

3.5.4 Ruled-based modeling languages

Alternatively, models can be described using rule patterns which describe families of related species and reactions. This enables large models with large state spaces to be concisely represented. This approach also emphasizes the biology/chemistry/physics which is pertinent to each rule. Furthermore, models that are described with rules can be efficiently simulated using network-free simulation which is an agent-based simulation technique which takes advantage of the typical low occupancy of models with large state spaces. The most popular rule-based modeling languages include BioNetGen and Kappa. The most popular network-free simulators including NFSim and KaSim.

However, all of the existing rule-based languages have a few critical limitations. First, they provide little support for semantic annotations. Second, they only support one specific type of combinatorial complexity, namely the combinatorial complexity that arises from having multiple binding sites per protein. In particular, the existing languages cannot easily described reactions that involve DNA, RNA, or protein sequences such as protein-DNA binding reactions.

Exercise

Encode the same stochastic model in BioNetGen.

3.5.5 Rule-based modeling API

The PySB rule-based modeling API overcomes some of the limitations of BioNetGen and Kappa by allowing modelers to define their own higher level abstractions which could be used to represent other types of combinatorial complexity. However, this requires modelers to develop their own unique higher level abstractions which again requires modeler to read lots of code to understand a model. Furthermore, models that are described in PySB have to be complied to BioNetGen models which diminishes the advantages of network-free simulation.

Exercise

Describe the same stochastic model with PySB.

3.5.6 High-level rule-based modeling language

To overcome the above limitations, we are developing a new higher-level rule-based modeling language and a corresponding higher-level network-free simulator.

3.6 Model annotation

In addition to describing models in an understandable format, it is also important to annotate models so that other modelers can understand their biological meaning and provenance.

Consider the following model of protein expression

$$\begin{aligned}\frac{dx}{dx} &= k - \frac{\ln 2}{\tau} x \\ k &= 10\,\mathrm{s}^{-1} \\ \tau &= 18,000\,\mathrm{s}\end{aligned}$$

Given just this equation, it is not clear what x represents, how the values of k and τ where determined, or what algorithm should be used to simulate the model. For example, no information is provided about whether x is one specific protein, the entire proteome, an RNA, or a metabolite. To enable others to understand the biological meaning of a model, we must provide additional annotations about the semantic meaning and provenance of each variable, equation, and parameter.

3.6.1 Component-level semantic annotations of species, reactions, and parameters

The most common way to communicate the biological meaning of a species or reaction is to annotate the species or reaction with a reference to an external database such as ChEBI for small molecules or UniProt for proteins. SBML supports these annotations as do several software programs including COPASI and VCell.

However, this approach to annotating models via references to external databases a few limitations

- Annotations are limited by the limited content of the databases. For example, ChEBI only contains a fraction of all possible small molecules.
- Many databases have insufficient resolution to represent variants of species. For example, UniProt does not have separate entries for each splice variants of a gene and even if it did, UniProt could never represent all possible splice variants. Similar, ChEBI does not include every proton isomer of every molecule.
- This approach focuses on species and reaction instances and does not support species or reaction rules.

Instead, we recommend annotating the meanings of species and reactions based on their absolute physical structure.

- Small molecules: InChI-encoded structures
- DNA, RNA, proteins: Sequences of references to InChI-encoded structures of nucleic and amino acids
- Complexes: Stoichiometric subunit composition
- Reactions: stoichiometries of reactants and products

3.6.2 Model-level semantic annotations

Several ontologies have been developed to help modelers describe the semantics of entire models. Below are some of the most useful ontologies for cell modeling. See BioPortal for a comprehensive list of ontologies.

- CCO: Cell cycle ontology: can be used to describe cell cycle phases represented by a model
- CL: Cell: can be used to describe cell type represented by a model
- PO: Pathway ontology: can be used to describe the pathways represented by a model

Several software programs can be used to annotate the semantics of entire models. See http://www.ebi.ac.uk/biomodels-main/annotation for a comparison of these programs.

- COPASI
- Saint
- SBMLEditor
- semanticSBML
- SemGen

3.6.3 Model provenance annotations

Currently (June 2017), there are no good formats or ontologies for describing the data sources and assumptions used to build models. Currently, we recommend that modelers track this information themselves and embed this information into their model definitions using custom annotations.

3.6.4 Simulation algorithm annotations

The KiSAO ontology can be used to describe how to simulate a model. KiSAO is supported by the Simulation Experiment Description Markup Language (SED-ML).

3.6.5 Exercises

1. Download a curated model from BioModels and obtain the paper which reported the model.
2. Remove all of the annotation from the model.
3. Using the paper, databases such as ChEBI and UniProt, and ontologies such as the PO, annotate the semantic meaning of each species and reaction.
4. Using the paper, embed custom annotations which describe the provenance of the model including the assumptions that the modelers made and the data sources that the modelers used to build and calibrate the model.

3.7 Model composition

Abstraction and composition are essential strategies for building large engineered systems such as big software programs. Abstraction allows engineers to build highly functional systems with complex and sophisticated internal implementations while only exposing the functional features that users of the systems need. The exposed features are called the system's external interface. Accessing the interface lets users obtain all the benefits of a system's sophisticated implementation without needing to understand the implementation. For example, the Python collection data structures like list, dictionary, and set are all built using sophisticated implementations, but Python programmers just use simple operations like insert, find and delete.

Composition of abstracted modules enables engineers to build complex systems by combining multiple parts, each of which has a simple external interface. In this way, abstraction and composition enable engineers to transform challenging high-dimensional problems into multiple, simpler lower dimensional problems. In particular, abstraction and composition enable teams of engineers to collaboratively build complex systems by enabling smaller groups or individual engineers to independently and simultaneously building individual components.

Because models are engineering software systems, abstraction and composition can also be powerful approaches to build large models. In particular, abstraction and composition can enable teams of modelers to work together to build large models by enabling smaller groups of modelers to model individual components.

Biological pathways are natural subsystems to abstract in dynamical biochemical models because they tend to interact tightly on fast timescales, and teams of microbiologists tend to focus their expertise on pathways. Furthermore, composition of models of pathways is a reasonable approximation due to the relatively fast dynamics of individual cellular pathways compared to dynamics of interactions between pathways. For example, the timescale of transcription is 10^1 s whereas the timescale of the transcriptome, which is the timescale of the impact of RNA on translation, is 10^2 s.

There are several specific motivations of building models via composition:

- Composition facilitates collaborative model design among multiple modelers by enabling each modeler to develop, calibrate, and test separate submodels. In particular, this enables each modeler to focus on a specific portion of the modeler without having to know all of the details of all of the other model components.
- Composition facilitates hybrid simulation of multiple mathematicallly distinct models. In particular, this can be a powerful strategy for modeling systems that involve either heterogeneous scales, heterogeneous amounts of scientific interest, or heterogeneous granularities of prior knowledge.
- Composition can reduce the dimensionality of model calibration by facilitating the calibration of separate subsets of models.
- Composition can reduce the dimensionality of model verification by facilitating the calibration of separate subsets of models.

Broadly, there are two types of model composition: composition of mathematically-like models and composition of mathematically-dissimilar models into hybrid or multi-algorithmic models. Mathematically-like models can be

merged analytically simply by taking the union of their variables/species and equations/reactions. Mathematically-dissimilar models must be merged computationally by concurrently integrating the individual models. In addition to merging models mathematically and/or computationally, it is often also necessary to align the models to a common namespace and representation.

3.7.1 Model composition procedure

Below are the steps to merging models.

1. Align the models to a common namespace by annotate the species and reactions using common ontologies
2. Make all of the implicit connections among the models implicit. For example, to combine a metabolism model with a signaling model, make ATP an explicit component of the signaling model and update the effective rate constants accordingly.
3. Identify the common species and reactions among the models
4. Enumerate all emergent combinatorial complexity from the model merging
5. Align the assumptions, granularity, and mathematical representation of the models
    1. Align the assumptions of all of the models. This is typically challenging to do because the assumptions underlying models are rarely explicitly stated.
    2. Align the granularities of all common species and reactions
    3. Convert all models into explicit time-driven models. For example, convert Boolean models into stochastic models by assuming typical time and copy number scales.
6. Mathematically and/or computationally merge the models
    1. Merge all of the mathematically-like models analytically by computing the unions of their species and reactions. For example, ODE models can be merged by taking the union of the state variables and summing the differentials across the models.
    2. Computationally merge the groups of mathematically-dissimilar models by concurrently integrating the models (see the multi-algorithm simulation tutorial).
7. Calibrate the combined model. Potentially this could be done be reusing the data that was used to calibrate the individual models. However, this data is rarely published.
8. Validate the combined model. This could also potentially be done using the same data that was used to validate the individual models. However, this data is rarely published.

Unfortunately, it is often challenging to merge published pathway models because published models are often not sufficiently annotated to understand the semantic meaning of each species and reaction and every assumption, because published pathway models often ignore important connections among pathways by lumping them into effective rate constants, and because published pathway models are often calibrated to represented different organisms and environmental conditions. Furthermore, only a few pathways including metabolism, signaling, and cell cycle regulation have good dynamic models that would be useful to merge into whole-cell models.

Instead, we recommend building whole-cell models by designing submodels from scratch explicitly for the purpose of model composition. This ensures that models are described using compatible assumptions, namespaces, granularities, and mathematical representations and calibrated to represent a single common organism and environmental condition.

3.7.2 Software tools

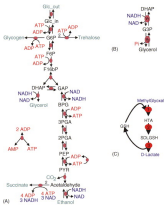
Several software tools have been developed to help researchers merge models and simulate merge models.

- Model merging: The tools below help users merge models. However, these tools only help users carryout the simplest model merging tasks, namely annotating the semantic meaning of model components and identifying common model components. These software programs do not help models carryout the more complicated tasks of resolving inconsistent assumptions and granularities and recalibrating models.
  - semanticSBML: helps users annotate models and identify common elements
  - SemGen: helps users annotate models and identify common elements
- Numerical simulation of composite multi-algorithmic models
  - E-Cell: multi-algorithmic simulator that uses a nested simulation algorithm
  - iBioSim: implements a hierarchical SSA algorithm that can simulate a specific class of merged stochastic models
  - COPASI: partitions a biochemical network into a high particle count subnet simulated by ODE and a low particle count subnet simulated by SSA

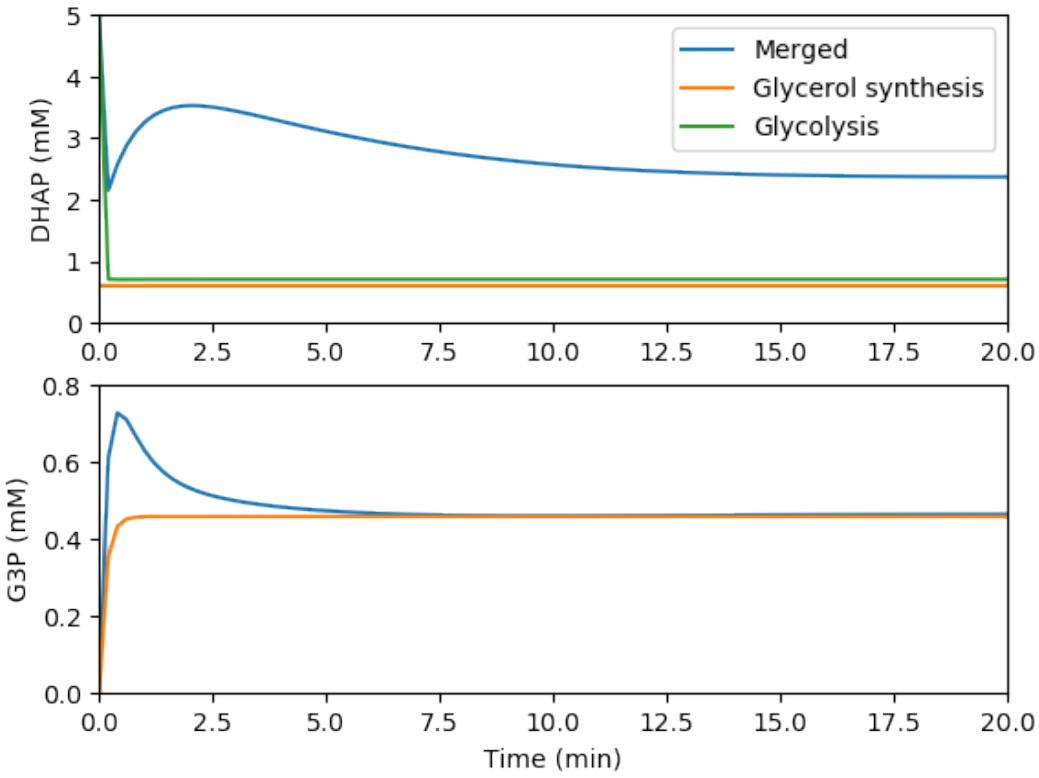
3.7.3 Exercises

Merging metabolic models

In this exercise, you will learn how to merge models by working through the nuances of merging two separately published models of glycolysis (A) and glycerol synthesis (B).



1. Read the papers which describe the individual models
   - Cronwright et al., 2002
   - Teusink et al., 2000
2. Obtain the original models from JWS online
   - Cronwright model
   - Teusink model
3. Identify the common species and reactions among the models
4. Merge the corresponding variables and equations
5. Compare your merged model with our solution
6. Simulate the individual and merged models
7. Compare the predictions of the individual and merged models. You should see results similar to those below.



8. Read this paper which describes an even larger merged model that includes a third submodel: Snoep et al., 2006

Merging electrophysiological models

In this exercise. you will learn how to merge models by working through the nuances of merging three separately published models of the electrophysiology, calcium dynamics, and tension development of cardiac myocytes.

1. Read the papers which describe the merged model
    - Terkildsen et al., 2008
    - Niederer et al., 2007
    - Neal et al., 2015
2. Read the papers which describe the original models
    - Pandit et al., 2001
    - Hinch et al., 2004
    - Niederer et al., 2006
3. Obtain the original models in CellML format from the CellML model repository
    - Pandit model
    - Hinch model
    - Niederer model
4. Identify the common species and reactions among the models by annotating the model components against a single namespace

5. Merge the corresponding variables and equations
6. Simulate the merged model
7. Compare your simulation results to those reported in Terkildsen et al., 2008; Niederer et al., 2007; and Neal et al., 2015.

3.8 Mathematical representations and simulation algorithms

There is wide range of mathematical representations that can be used model cells, ranging from coarse-grained representations such as Boolean networks to fine-grained representations such as spatial stochastic models. Below are several considerations for choosing a mathematical representation for a model.

- What is the model's purpose? What properties of the cell are being analyzed by the model?
- What are the relevant physical scales of the biology? Deterministic models are valid approximations when the fluctuations are small which often occurs when the flux scale is large. Conversely, stochastic models should be used when the fluctuations are large and/or they critically impact behavior.
- How much information is known about the biology? If significant information is known, then fine-grained models are possible. Conversely, if the biology is not well-characterized a fine-grained representation may be infeasible and a coarse-grained representation should be used.
- How much computational cost is acceptable? If you need a model to execute quickly or you anticipate running large numbers of simulations, it may be best to use a coarse-grained representation. Alternatively, if you want to build the most detailed model possible, you should use a fine-grained representation.
- What simulation software tools are available? Several tools are available to simulate single-algorithm models. However, there are few tools for simulating hybrid models. If you don't want to spend a lot of energy writing numerical simulation code, we recommend you focus on well-established mathematical representations.

The goal of this tutorial is to introduce you to the most common mathematical representations, simulation algorithms, simulation experiment description formats, and simulation software programs for cell modeling. In addition, this tutorial will highlight the advantages and disadvantages of each method and tool, particularly for large-scale modeling.

3.8.1 Boolean/logical models

Boolean models are composed of composed Boolean variables and logical functions which describe how the value of each variable is influenced by other variables. Logical or multi-level models are a generalization of Boolean models in which the variables can have two or more possible values. Importantly, Boolean models have no quantitative parameters or explicit timescales. Similarly, logical models have few quantitative parameters. Consequently, Boolean and logical models are often used to describe biological systems that have not yet been quantitatively characterized, as well as to build large models that contain many more variables/species and functions/interactions than could be calibrated in a more quantitative model.

Boolean models are simulated with timestep algorithms which divide simulations into many small timesteps during which the logical functions are evaluated and the Boolean variables are updated according to a specific *update scheme*. Below are some of the most commonly used update schemes:

- Synchronous updating: during each timestep, all of the functions are evaluated with the same node states and all of the nodes are updated with the results of the functions
- Asynchronous updating: during each timestep, only one or a subset of the logical functions are evaluated and only the corresponding subset of nodes are updated
  - Deterministic asynchronous updating: the timesteps iterate over the functions/rules in a deterministic order

  - Random asynchronous updating: at each timestep, the simulator randomly chooses one or more functions/nodes to update

There are also several generalizations of Boolean and logical-valued models including probabilistic Boolean models that describe each variable as a probability and temporal Boolean models that have explicit time scales.

3.8.2 Ordinary differential equations (ODEs)

Ordinary differential equations (ODEs) is one of the most commonly used approaches for modeling dynamical systems. ODE models are based on microscopic analyses of how the concentration of each species in the cell changes over time in response to the concentrations of other species. Because ODE assume that cells are well-mixed and that they behave deterministically, ODE models are most appropriate for small systems that involve large concentrations and high fluxes.

ODE models can be simulated by numerically integrating the differential equations. The most basic ODE integration method is Euler's method. Euler's method is a time stepped algorithm in which the next state is computed by adding the current state and the multiplication of the current differentials with the timestep.

$$y(t + \Delta t) = y(t) + \Delta t \frac{dy}{dt}$$

Euler's method estimates $y(t + \Delta t)$ using a first-order approximation. ODE models can be simulated more accurately using higher order estimates, or Taylor series, of $y(t+\Delta t)$. One of the most popular algorithms which implements this approach is the Runge-Kutta 4th order method. Yet more advanced integration methods select the time step adaptively. Some of the most sophisticated ODE integration packages include ODEPACK (lsoda) and Sundials (vode). These packages can be used in Python via scipy's integrate module.

3.8.3 Stochastic simulation

Like ODE models, stochastic models are based on microscopic analyses of how the concentration of each species in the cell changes over time in response to the concentrations of other species. However, stochastic simulation algorithms relax the assumption that cells behave deterministically. Instead, stochastic simulation algorithms assume that the rate of each reaction is Poisson-distributed. As a result, stochastic simulation algorithms generate sequences of reaction events at which the simulated state discretely changes.

Stochastic simulation should be used to model systems that are sensitive to small fluctuations such as systems that involve small concentrations and small fluxes. However, stochastic simulation can be computationally expensive for large numbers due to the needs to resolve the exact order of every reaction and the need to run multiple simulations to sample the distribution of predicted cell behaviors.

Because stochastic simulations are random, stochastic models should generally be simulated multiple times to sample the distribution of predicted cell behaviors. In general, these simulations should be run both using different random number generator seeds and different random initial conditions.

Gillespie Algorithm / Stochastic Simulation Algorithm / Direct Method

The simplest way to stochastically simulate a model is to iteratively

1. Calculate the instantaneous rate of each reaction, p, also known as the reaction *propensities*
2. Sum these rates
3. Sample the time to the next reaction by sampling from a exponential distribution with parameter $1/\sum p$ by utilizing these mathematical facts
   - The sum of independent Poisson processes with parameters λ_1 and λ_2 is a Poisson process with parameter $\lambda = \lambda_1 + \lambda_2$, and

- The time to the next event of a Poisson is exponentially distributed with parameter $1/\lambda$.

4. Sample the next reaction from a multinomial distribution with parameter $p/\sum p$ equal to the condition probability that each reaction fires given that a reaction fires.

Below is pseudo-code for this algorithm which is also known as the Gillespie algorithm, the Stochastic Simulation Algorithm (SSA), and the direct method:

```
import numpy

# represent the reaction and rate laws of the model
reaction_stochiometries = numpy.array([ ... ])
kinetic_laws = [...]

# initialize the time and cell state
time = 0
copy_numbers = numpy.array([ ... ])

while time < time_max:
    # calculate reaction properties/rates
    propensities = [kinetic_law(copy_numbers) for kinetic_law in kinetic_laws]
    total_propensity = sum(propensities)

    # select the length of the time step from an exponential distributuon
    dt = numpy.random.exponential(1 / total_propensity)

    # select the next reaction to fire
    i_reaction = numpy.random.choice(len(propensities), p=propensities / total_
↪propensity)

    # reject the selected reaction if there are insufficient copies of the reactants␣
↪for the reaction

    # update the time and cell state based on the selected reaction
    time += dt
    copy_numbers += reaction_stochiometries[:, i_reaction]
```

Gillespie first reaction method

Rather than sampling the time to the next reaction and then selecting the next reaction, alternatively we can stochastically simulate a model by (1) sampling the putative time to the next firing of each reaction and (b) firing the reaction with the minimum putative next reaction time. This algorithm is mathematically equivalent to the Gillespie algorithm. However, it is slower than the Gillespie algorithm because it draws more random number samples during each iteration.:

```
import numpy

# represent the reaction and rate laws of the model
reaction_stochiometries = numpy.array([ ... ])
kinetic_laws = [...]

# initialize the time and cell state
time = 0
copy_numbers = numpy.array([ ... ])

while time < time_max:
    # calculate reaction properties/rates
```

(continues on next page)

(continued from previous page)

```
    propensities = [kinetic_law(copy_numbers) for kinetic_law in kinetic_laws]

    # calculate putative next reaction times for each reaction
    dt = numpy.random.exponential(1 / propensities)

    # select the next reaction to fire
    i_reaction = numpy.argmin(dt)

    # reject the selected reaction if there are insufficient copies of the reactants␣
↪for the reaction

    # update the time and cell state based on the selected reaction
    time += dt[i_reaction]
    copy_numbers += reaction_stochiometries[:, i_reaction]
```

Gibson-Bruck Next Reaction Method

The Gibson-Bruck Next Reaction Method is a computational optimization of the Gillespie first reaction method which uses (a) a dependency graph to only recalculate rate laws and resample putative next reaction times when necessary, namely when the participants in the rate law are updated and (b) an *indexed priority queue* to minimize the computational cost of identifying the reaction with the lowest putative next reaction time and updating the data structure which stores these putative next reaction times.

An indexed priority queue is a data structure that provides efficient identification ($O(1)$) of the minimum value of the list and efficient updating of the list ($O(\log n)$. Indexed priority queues are implemented by the Python `pqdict` package. See the pqdict documentation for more information about indexed priority queues.

Note, the Gibson-Bruck Next Reaction Method is mathematically equivalent to the Gillespie direct and Gillespie first reaction methods.

Below is pseudo code for the Gibson-Bruck Next Reaction Method:

```
import numpy
import pqdict

# represent the reaction and rate laws of the model
reaction_stochiometries = numpy.array([ ... ])
kinetic_laws = [...]

# represent the dependency of the kinetic laws on the species
dependency_graph = numpy.array([...])

# initialize the time and cell state
time = 0
copy_numbers = numpy.array([ ... ])

# calculate reaction properties/rates
propensities = [kinetic_law(copy_numbers) for kinetic_law in kinetic_laws]

# calculate putative next reaction times for each reaction
dt = pqdict.pqdict( ... numpy.random.exponential(1 / propensities) ... )

while time < time_max:
    # select the next reaction to fire
    i_reaction = numpy.argmin(dt)
```

(continues on next page)

(continued from previous page)

```
    # reject the selected reaction if there are insufficient copies of the reactants␣
↪for the reaction

    # update the time and cell state based on the selected reaction
    time += dt[i_reaction]
    copy_numbers += reaction_stochiometries[:, i_reaction]

    chosen_dt = dt[i_reaction]
    for species in reaction_stochiometries[:, i_reaction]:
        for reaction in dependency_graph[:, species]:
            old_propensity = propensities[reaction]
            propensities[reaction] = kinetic_laws[reaction]
            if reaction == i_reaction:
                dt[reaction] = numpy.random.exponential(1 / propensities[reaction])
            else:
                dt[reaction] = old_propensity / propensities[reaction] *␣
↪(dt[reaction] - chosen_dt)
```

Tau-leaping

In addition to the Gillespie algorithm, the Gillespie first reaction method, and the Gibson-Bruck method, there are many algorithms which approximate their results with significantly lower computational costs. One of the most common of these approximate simulation algorithms is the tau-leaping algorithm. The tau-leaping algorithm is a time-stepped algorithm similar to Euler's method which samples the number of firings of each reaction from a Poisson distribution with lambda equal to the product of the propensity of each reaction and the time step. Below is pseudocode for the tau-leaping algorithm:

```
# represent the reaction and rate laws of the model
reaction_stochiometries = numpy.array([ ... ])
kinetic_laws = [...]

# select the desired time step
dt = 1

# initialize the simulated state
time = 0
copy_numbers = numpy.array([...])

# iterate over time
while time < time_max:
    # calculate the rate of each reaction
    propensities = [kinetic_law(copy_numbers) for kinetic_law in kinetic_laws]

    # sample the number of firings of each reaction
    n_reactions = numpy.random.poisson(propensities * dt)

    # adjust the time step or reject reactions for which there are insufficient␣
↪reactants

    # advance the time and copy numbers
    time += dt
    copy_numbers += reaction_stochiometries * n_reactions
```

The tau-leaping algorithm can be improved by adaptively optimizing time step based on its sensitivity to the propen-

sities:

$$
\begin{aligned}
g_i &= -\min_j S_{ij} \\
\mu_i &= \sum_j S_{ij} R_j(x) \\
\sigma_i^2 &= \sum_j S_{ij}^2 R_j(x) \\
dt &= \min_i \left\{ \frac{\max\{\epsilon x_i / g_i, 1\}}{|\mu_i(x)|}, \frac{\max\{\epsilon x_i / g_i, 1\}^2}{\sigma_i^2} \right\}
\end{aligned}
$$

where x_i is the copy number of species i, S_{ij} is the stoichiometry of species i in reaction j, $R_j(x)$ is the rate law for reaction j, and $\epsilon \approx 0.03$ is the desired tolerance. See Cao, 2006 for more information.

3.8.4 Network-free simulation

Network-free simulation is a variant of stochastic simulation for simulating rule-based models in which the occupied species and active reactions are discovered dynamically during simulation rather than statistically enumerating all possible species and reactions prior to simulation. Network-free simulation is a mathematically equivalent algorithm for stochastic simulation. The key advantage of network-free simulation is that it can simulate models with very large or infinitely large state state spaces that cannot be simulated with conventional simulation algorithms.

3.8.5 Flux balance analysis (FBA)

Flux balance analysis (FBA) is a popular approach for simulating metabolism. Like ODE models, FBA is based on microscopic analyses of how the concentration of each species in the cell changes over time in response to the concentrations of other species. However, unlike ODE models which assume that each reaction can be annotated with a rate law, FBA does not assume that rate laws can be determined but does assume that cells have evolved to maximize their growth rates. In addition, FBA assumes that all species are at steady-state ($\frac{dx}{dt} = 0$), which greatly constrains the model, thereby reducing the amount of data needed. However, this assumption also prevents FBA from making predictions about the dynamics of metabolic networks.

Together, these assumptions enable FBA to determine reaction fluxes by posing an optimization problem that maximizes growth. To achieve this, FBA leverages two additional sets of data. First, it incorporates estimates of the minimum and maximum possible flux of each reaction which, for a subset of reactions, can be obtained from experimental observations. These constrain the optimization solution space. Second, FBA expresses the growth of a cell via an additional pseudo-reaction called the biomass reaction that represents the assembly of metabolites into the molecules which are incorporated in the growth of a cell. It can be calibrated based on the observed cellular composition. Together, these assumptions and additional data enable FBA to pose cell simulation as a linear optimization problem that can be solved with linear programming.

$$
\begin{aligned}
\text{Maximize } v_{\text{growth}} &= f'v \\
\text{Subject to} & \\
Sv &= 0 \\
v^{\min} \leq v &\leq v^{\max},
\end{aligned}
$$

where v_j is the flux of reaction j, f_μ is 1 for the biomass reaction and 0 otherwise, S_{ij} is the stoichiometry of species i in reaction j, and $v_j^{\min}$ and $v_j^{\max}$ are the upper and lower bounds of the flux of reaction j.

In addition, there are a variety of generalizations of FBA for using genomic and other experimental data to generate more accurate flux bounds (see review), and combined regulatory and FBA metabolism simulations (rFBA, PROM).

Dynamic FBA simulations (dFBA) enables dynamic models of metabolism by (1) assuming that cells quickly reach pseudo-steady states with their environment because their internal dynamics are significantly faster than that of the external environment, (2) iteratively forming and solving FBA models, and (3) updating the extracellular concentrations based on the predicted fluxes. Below is pseudo-code for dFBA:

```
Set the initial biomass concentration
Set the initial conditions of the environment
From the starting time to the final time
    Based on the current biomass concentration and environmental conditions,
        set the upper and lower bounds of the exchange reactions
    Solve for the maximum growth rate and optimal fluxes
    Update the biomass concentration based on the predicted growth rate
    Update the environmental conditions based on the predicted exchange fluxes
```

3.8.6 Hybrid/multi-algorithmic simulation

A hybrid or multi-algorithmic simulation is a co-simulation of multiple submodels which employ multiple simulation algorithms. For example, a model may contain a metabolism submodel integrated with the dFBA algorithm and transcription, translation and degradation submodels integrated with the stochastic simulation algorithm (SSA). Hybrid simulation often requires combining and synchronizing of discrete and continuous submodels that contain discrete and continuous variables. For example, a hybrid simulation could co-simulate a discrete stochastic submodel that uses the stochastic simulation algorithm and a deterministic continuous submodel that uses the Runge-Kutta 4th order method.

Hybrid simulation is a powerful strategy for managing heterogeneity in the scales of the model system, heterogeneity in a modeler's interest in specific aspects of the model system, heterogeneity in the amount of knowledge of specific aspects of the model system, and heterogeneous model design decisions among multiple collaborating modelers:

- Hybrid simulation can enable efficient simulation of models that span a range of scales, including low-concentration components whose dynamics are highly variable and high-concentration components whose dynamics exhibit little variation. In this case, the modeler or hybrid simulation algorithm selects the most appropriate simulation algorithm for each model component to optimize the trade off between computational accuracy and cost. Specifically, the simulator partitions the species and reactions into two or more submodels that partition species by their concentration ranges. For more information, we recommend reading recent papers about partitioned and slow-scale tau-leaping.
- Hybrid simulation can enable model components to be described with different resolutions either due to different levels of interest or prior knowledge about specific aspects of the model system. For example, hybrid simulation could be used to simultaneously but separately represent the known stochastic behavior of transcription and the known steady-state behavior of metabolism. In this case, a modeler would select SSA to represent transcription and dFBA to represent metabolism.
- Hybrid simulation can make it easier for modelers to build the submodels of a large model in parallel. Multiple collaborating investigators may independently develop submodel components of the full model, make mathematically different design decisions for their independent submodels, and separately calibrate and verify their submodels. Hybrid simulation can then simulate the entire large model.

Approaches to constructing hybrid simulations

Broadly, there are two ways to construct hybrid simulations:

- Hybrid simulations can be designed by the modeler(s) to maximize the utility of the model. In this case, the modeler(s) use their expert knowledge to assign each component of the model to a specific simulation algorithm. Examples of hybrid simulations that have been designed by modelers include integrated flux balance analysis (iFBA) and regulatory flux balance analysis (rFBA)

- Hybrid simulations can be structured by the simulation software to optimize the tradeoff between prediction accuracy and computational cost. In this case, the simulator automatically partitions the model into algorithmically-distinct submodels. Examples of hybrid simulations that are chosen by simulation software systems include partitioned tau-leaping and slow scale tau-leaping.

Numerical simulation

Hybrid simulation algorithms concurrently integrate a model's component submodels. One approach iterates over many small time steps and alternately integrates the submodels and synchronizes their states. To simplify this problem, submodels that should be simulated with the same simulation algorithm can first be analytically merged, thereby reducing the number of submodels that must be concurrently integrated. Thus, hybrid simulation broadly requires two steps:

1. Analytically merge all of the algorithmically-like submodels
2. Co-simulate or concurrently integrate the merged submodels

The goal of hybrid simulation is to represent the shared state and dynamics of a model composed of multiple submodels. Since submodels that employ different simulation algorithms must be integrated separately, achieving this goal requires that a hybrid simulator synchronize the execution of different submodels and exchange shared state among the submodels.

Below, we summarize several increasingly sophisticated hybrid simulation algorithms:

- Serial simulation: A global cell state is initialized at time 0. Divide the simulation of a cell into multiple small time steps. Within each time step, sequentially simulate all submodels. Each submodel transfers its local state updates to the global cell state. Optionally, randomize the order in which submodels are simulated within a time step. However, this algorithm introduces inaccuracies because it violates the arrow of time by integrating submodels based on different states within each time step.
- Partitioning and merging: Divide the simulation into multiple small time steps. With each time step, partition the pool of each species into separate pools for each submodel. Simulate the submodels independently using the independent pools. Update the global species pools by merging the submodel pools. Species can be partitioned uniformly, based on their utilization during the previous time point, or based on a preliminary integration of the submodels. This is a relatively simple algorithm to implement for models whose state only represents concentrations and/or species copy number. However, it can be challenging to partition and merge rule-based models whose states are represented by graphs.
- Interpolation: For models that are composed of one discrete and one continuous model, we can append the continuous model to the discrete model as a single pseudo reaction, fire the reaction immediately after each discrete reaction, and periodically calculate the coefficients of the pseudo reaction by numerically integrating the continuous model. Furthermore, we could adaptive choose the frequency at which the continuous model is simulated based on the sensitivity of its predictions to its inputs from the discrete model. Overall, we believe this algorithm provides a good balance of accuracy and speed.
- Scheduling: Build a higher-order stochastic model which contains one pseudo-reaction per submodel. Set the propensity of each pseudo-reaction to the total propensity of the submodel. Use the Gillespie method to schedule the firing of the pseudo-reactions/submodels. This is the strategy used by E-Cell (Takahashi et al., 2004).
- Upscaling: For models that are composed of one discrete and one continuous model, we can append all of the continuous reactions model to the discrete model; periodically set their propensities by evaluating their kinetic laws, or in the case of FBA, calculating the optimal flux distribution; and select and fire the continuous reactions using the same mechanism as the discrete reactions. This algorithm is compelling, but is computationally expensive due to the need to resolve the order of every reaction including every continuous reaction.

Synchronizing discrete and continuous variables

Synchronizing discrete and continuous models can require rounding continuous variables and/or rates. In such cases, we recommend stochastically rounding the decimal portion of each the continuous-valued rate, weighted by the decimal portion of the rate. This method has the following advantages over other potential methods:

- This method will not prevent models from reaching rare states unlike deterministic rounding which will deterministically round down small probabilities.
- This method satisfies mass, atom, and charge balance unlike rounding the values of the model variables.
- This method will only inject a small amount of stochastic variation into the model unlike other potential stochastic rounding schemes such as Poisson-distributed random rounding.

See Vasudeva et al., 2003 for additional discussion about rounding continuous quantities for synchronization with discrete quantities.

Simulating individual submodels

For calibration and testing, it is useful to simulate individual submodels of a larger model. To meaningfully simulate a submodel, you must *mock* the impact of all of the other submodels that the submodel interacts with. This can be done by creating coarse-grained versions of all of the other submodels which mimic their external behavior at a lower computational cost.

3.8.7 Reproducing stochastic simulations

To run stochastic simulations reproducibly, every simulation input must be identical, including the order in which the reactions are defined and the state of the random number generator.

The state of the `numpy` random number generator can be set using the `seed` method:

```
numpy.random.seed(0)
```

3.8.8 Simulation descriptions

To simulate a model, we must specify every aspect of the simulation including

- The model that we want to simulate
- Any modifications of the model that we wish to simulate (e.g. modified parameter values)
- The initial conditions of the simulation
- The desired simulation algorithm
- The parameters of the simulation algorithm such as the initial seed and state of the random number generator

The Simulation Experiment Description Markup Language (SED-ML) is one of the most commonly used languages for describing simulations. SED-ML is primarily designed to describe simulations of XML-based models such as models encoded in CellML and SBML. However, SED-ML can be used to describe the simulation of any model. Simulation Experiment Specification via a Scala Layer (SESSL) is competing language simulation description language.

3.8.9 Software tools

Below is a list of some of the most commonly used simulation software programs for cell modeling:

- Desktop programs:
  - CellNOpt: Boolean simulator
  - COPASI: ODE and stochastic simulator
  - ECell: multi-algorithmic simulator
  - iBioSim
  - NFSim: stochastic network-free simulator
  - VCell: ODE, stochastic, and spatial simulator
- Web-based programs
  - Cell Collective: online Boolean simulator
  - JWS Online: online ODE and stochastic simulator
- Python libraries
  - BooleanNet: Boolean simulation
  - COBRApy: FBA simulation
  - libRoadRunner: ODE simulation
  - LibSBMLSim: ODE simulation
  - PySB: rule-based simulation
  - StochPy: stochastic simulation

3.8.10 Exercises

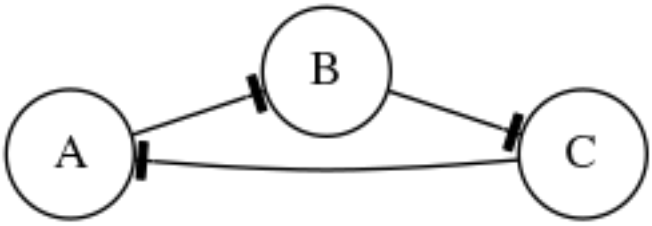
Required software

For Ubuntu, the following commands can be used to install all of the software required for the exercises:

```
sudo apt-get install \
    python \
    python-pip
sudo pip install \
    matplotlib \
    numpy \
    optlang \
    scipy
```

Boolean simulation

In this exercise we will simulate the gene Boolean network shown below using both synchronous and asynchronous updating.



First, define a set of functions which represent the edges of the network:

```
regulatory_functions = {
    'A': lambda nodes: not nodes['C'],
    'B': lambda nodes: not nodes['A'],
    'C': lambda nodes: not nodes['B'],
}
```

Second, define the initial state of the network:

```
initial_state = {
    'A': False,
    'B': True,
    'C': True
}
```

Third, write a Boolean simulator and synchronous, asynchronous deterministic, and asynchronous random update schemes by completing the code fragments below:

```
def simulate(regulatory_functions, initial_state, n_steps, update_scheme):
    """ Simulates a Boolean network for :obj:`n_steps` using :obj:`update_scheme`

    Args:
        regulatory_functions (:obj:`dict` of :obj:`str`, :obj:`function`): dictionary␣
↪of regulatory functions for each species
        initial_state (:obj:`dict` of :obj:`str`, :obj:`bool`): dictionary of initial␣
↪values of each species
        n_steps (:obj:`int`): number of steps to simulate
        update_scheme (:obj:`method`): update schema

    Returns:
        :obj:`tuple`:

            * :obj:`numpy.ndarray`: array of step numbers
            * :obj:`dict` of :obj:`str`, :obj:`numpy.ndarray`: dictionary of␣
↪histories of each species
    """

    # initialize data structures to store predicted time course and copy initial␣
↪state to state history
    ...

    # set current state to initial state
    ...

    # iterate over time steps
    for step in range(1, n_steps + 1):
        # update state according to :obj:`update_scheme`
        ...
```

(continues on next page)

(continued from previous page)

```
        # store current value
        ...

    # return predicted dynamics
    ...


def sync_update_scheme(regulatory_functions, step, current_state):
    """ Synchronously update species values

    Args:
        regulatory_functions (:obj:`dict` of :obj:`str`, :obj:`function`): dictionary␣
↪of regulatory functions for each species
        step (:obj:`int`): step iteration
        current_state (:obj:`dict` of :obj:`str`, :obj:`bool`): dictionary of values␣
↪of each species

    Returns:
         :obj:`dict` of :obj:`str`, :obj:`bool`: dictionary of values of each species
    """
    # calculate next state
    ...

    # return state
    return next_state


def deterministic_async_update_scheme(regulatory_functions, step, current_state):
    """ Asynchronously update species values in a deterministic order

    Args:
        regulatory_functions (:obj:`dict` of :obj:`str`, :obj:`function`): dictionary␣
↪of regulatory functions for each species
        step (:obj:`int`): step iteration
        current_state (:obj:`dict` of :obj:`str`, :obj:`bool`): dictionary of values␣
↪of each species

    Returns:
         :obj:`dict` of :obj:`str`, :obj:`bool`: dictionary of values of each species
    """
    # calculate next state
    ...

    # return state
    return current_state


def random_async_update_scheme(regulatory_functions, step, current_state):
    """ Asynchronously update species values in a random order

    Args:
        regulatory_functions (:obj:`dict` of :obj:`str`, :obj:`function`): dictionary␣
↪of regulatory functions for each species
        step (:obj:`int`): step iteration
        current_state (:obj:`dict` of :obj:`str`, :obj:`bool`): dictionary of values␣
↪of each species
```

(continues on next page)

(continued from previous page)

```
    Returns:
        :obj:`dict` of :obj:`str`, :obj:`bool`: dictionary of values of each species
    """
    # calculate next state
    ...

    # return state
    return current_state
```

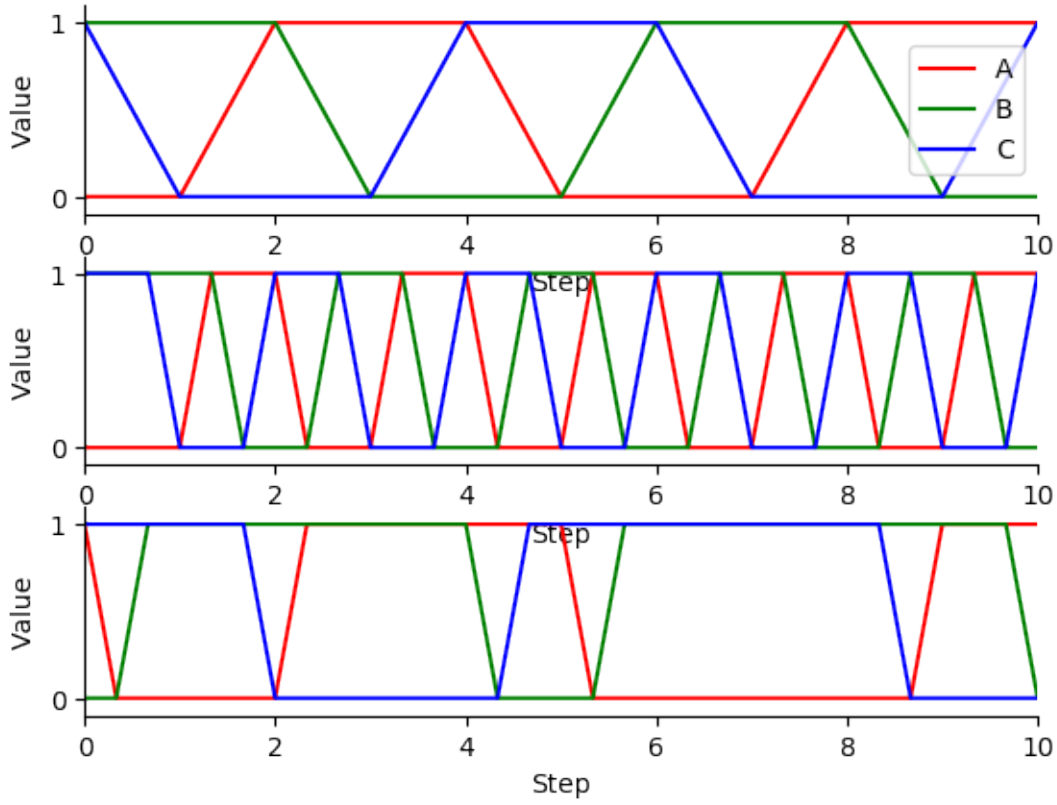
Fourth, seed numpy's random number generator so that we can reproducibly simulate the model:

```
numpy.random.seed(0)
```

Fifth, use the simulation functions to simulate the model:

```
n_steps = 10
sync_time_hist, sync_hist = simulate(regulatory_functions, initial_state, 20, sync_
↪update_scheme)
det_async_time_hist, det_async_hist = simulate(regulatory_functions, initial_state,␣
↪20 * 3, deterministic_async_update_scheme)
rand_sync_time_hist, rand_sync_hist = simulate(regulatory_functions, initial_state,␣
↪20 * 3, random_async_update_scheme)

det_async_time_hist = det_async_time_hist / 3
rand_sync_time_hist = rand_sync_time_hist / 3
```

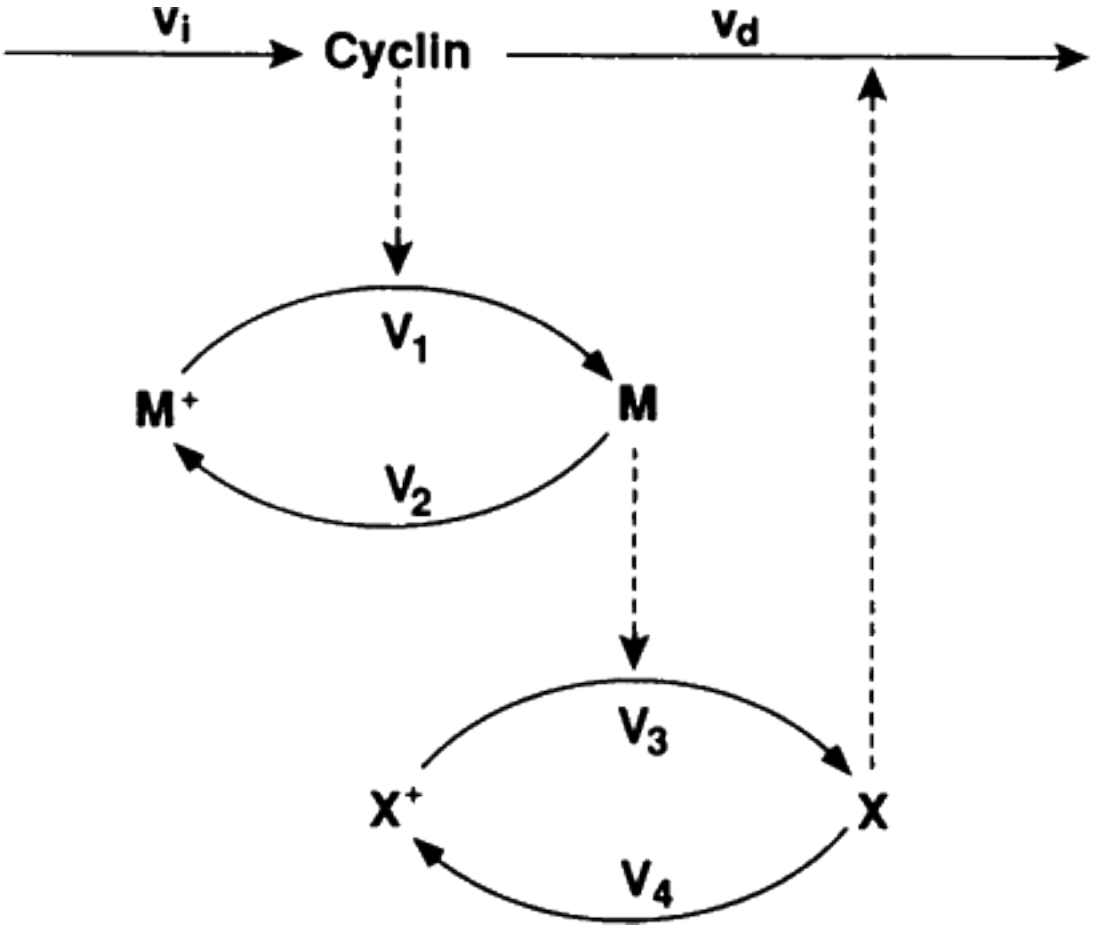
Next, use `matplotlib` to plot the simulation results. You should see results similar to those below.



Finally, compare the simulation results from the different update schemes. How do they differ? Which ones reach steady states or repetitive oscillations?

ODE simulation

In this exercise we will use lsoda to simulate the Goldbeter 1991 cell cycle model of cyclin, cdc2, and cyclin protease (doi: 10.1073/pnas.88.20.9107, BIOMD0000000003).



First, open the BioModels entry for the model in your web browser. Identify the reactions, their rate laws, the parameter values, and the initial conditions of the model. Note, that the model uses two assignment rules for $V1$ and $V3$ which are not displayed on the BioModels page. These assignment rules must be identified from the SBML version of the model which can be exported from the BioModels page.

Second, write a function to calculate the time derivative of the cyclin, cd2, and protease concentrations/activities by completing the code fragment below:

```
def d_conc_d_t(concs, time):
    """ Calculate differentials for Goldbeter 1991 cell cycle model
    (`BIOMD0000000003 <http://www.ebi.ac.uk/biomodels-main/BIOMD0000000003>`_)

    Args:
        time (obj:`float`): time
        concs (:obj:`numpy.ndarray`): array of current concentrations

    Returns:
```

(continues on next page)

(continued from previous page)

```
        :obj:`numpy.ndarray
    """
    ...
```

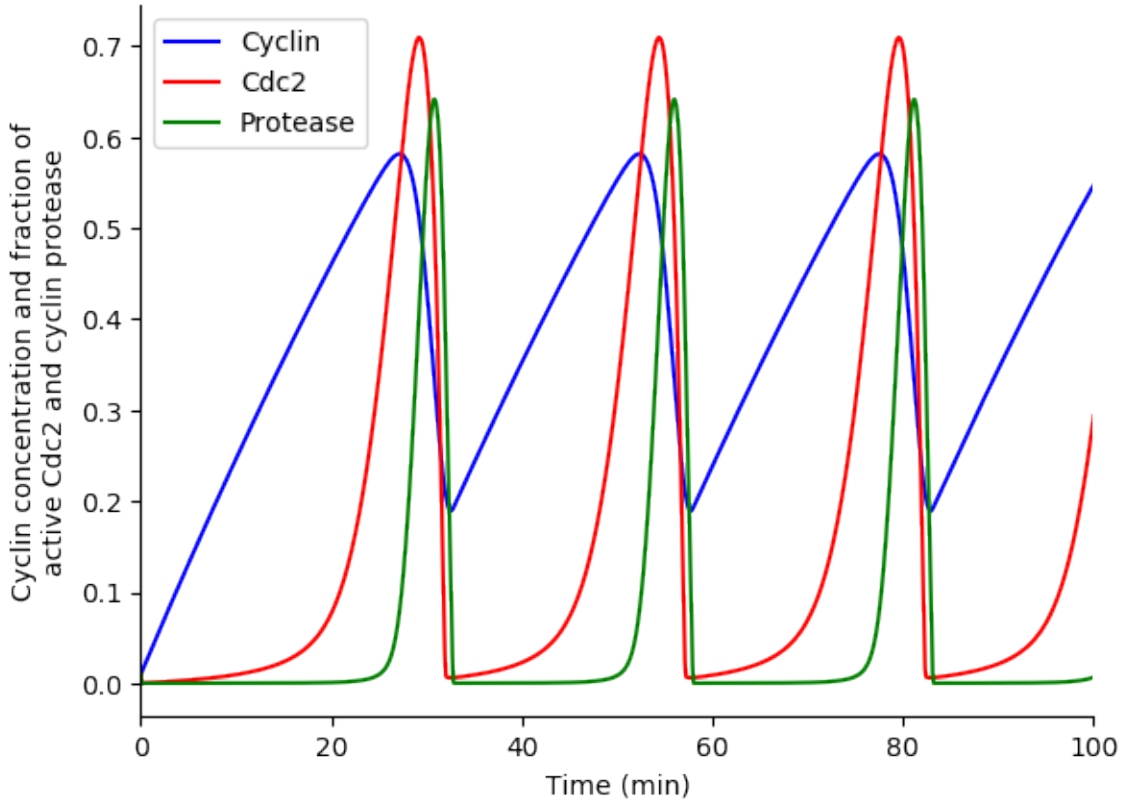
Next, create a vector to represent the initial conditions by completing this code fragment:

```
init_concs = ...
```

Next, use `scipy.integrate.odeint` which implements the lsoda algorithm to simulate the model by completing this code fragment:

```
time_max = 100
time_step = 0.1
time_hist = numpy.linspace(0., time_max, time_max / time_step + 1)
conc_hist = scipy.integrate.odeint(...)
```

Finally, use `matplotlib` to plot the simulation results. You should see results similar to those below.



Stochastic simulation

In this exercise we will simulate the stochastic synthesis and degradation of a single RNA using the Gillespie algorithm.

$$\begin{aligned}
\text{Synthesis: } \emptyset &\rightarrow \text{mRNA} \\
\text{Degradation: mRNA} &\rightarrow \emptyset \\
r_{\text{syn}} &= k_{\text{syn}} \\
r_{\text{deg}} &= k_{\text{deg}}\text{mRNA} \\
k_{\text{syn}} &= 2.5\,\text{s}^{-1} \\
k_{\text{deg}} &= 0.5\,\text{s}^{-1}\,\text{molecule}^{-1} \\
\text{mRNA}(t=0) &= 10\,\text{molecules}
\end{aligned}$$

First, define data structures to represent the stoichiometries and rate laws of the reactions:

```
reaction_stochiometries = [1, -1]

k_syn = 2.5 # 1/s
k_deg = 0.5 # 1/s/molecule
def kinetic_laws(copy_number):
    return numpy.array([
        k_syn,
        k_deg * copy_number,
        ])
```

Second, define the initial copy number:

```
init_copy_number = 10
```

Third, implement the Gillespie algorithm by completing the code skeleton below:

```
def simulate(reaction_stochiometries, kinetic_laws, init_copy_number, time_max, time_
↪step):
    """ Run a stochastic simulation

    Args:
       reaction_stochiometries (:obj:`list` of :obj:`int`): list of stoichiometries␣
↪of the protein in each reaction
       kinetic_laws (:obj:`list` of :obj:`function`): list of kinetic law function
       init_copy_number (:obj:`int`): initial copy number
       time_max (:obj:`float`): simulation length
       time_step (:obj:`float`): frequency to record predicted dynamics

    Returns:
        :obj:`tuple`:

            * :obj:`numpy.ndarray`: time points
            * :obj:`numpy.ndarray`: predicted copy number at each time point
    """

    # data structure to store predicted copy numbers
    time_hist = numpy.linspace(0., time_max, time_max / time_step + 1)
    copy_number_hist = numpy.full(int(time_max / time_step + 1), numpy.nan)
    copy_number_hist[0] = init_copy_number

    # initial conditions
    time = 0
```

(continues on next page)

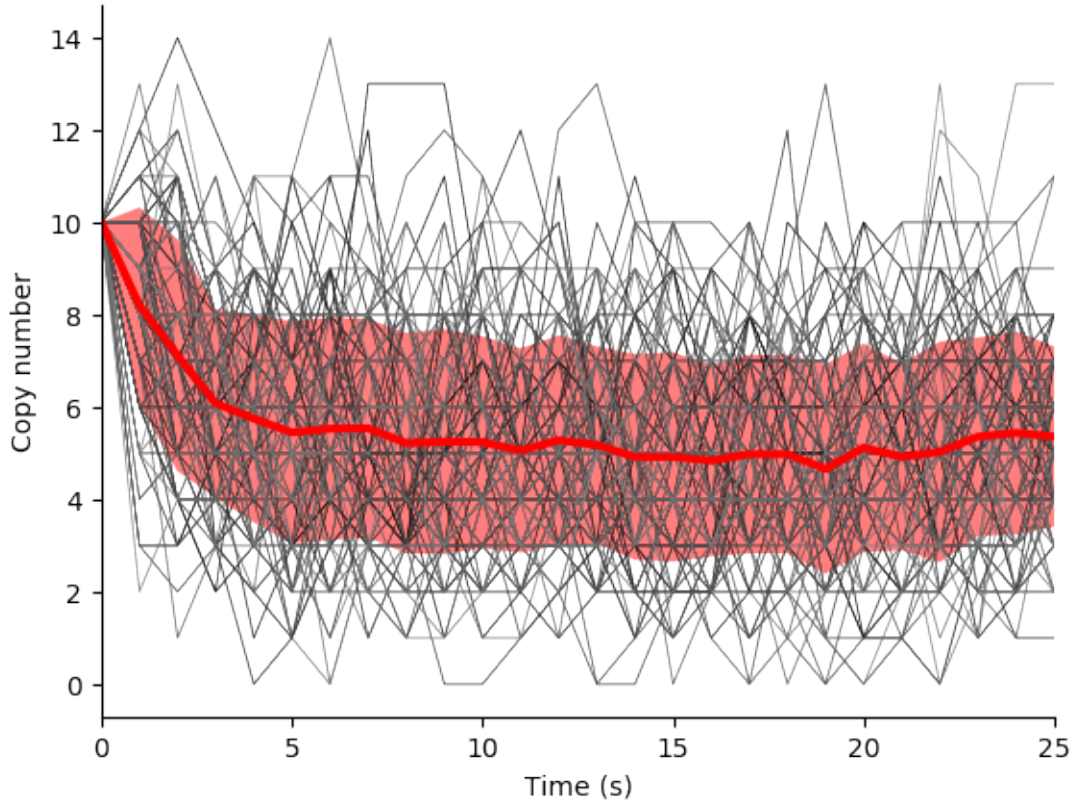
(continued from previous page)

```
    copy_number = init_copy_number

    # iterate over time
    while time < time_max:
        # calculate reaction properties/rates
        propensities = ...

        # calculate total propensity
        total_propensity = ...

        # select the length of the time step from an exponential distributuon
        dt = ...

        # select the next reaction to fire
        i_reaction = ...

        # update the time and copy number based on the selected reaction
        time += ..
        copy_number += ...

        # store copy number history
        #print(time)
        if time < time_max:
            copy_number_hist[int(numpy.ceil(time / time_step)):] = copy_number

    return (time_hist, copy_number_hist)
```

Fourth, seed random number generator so that we can reproducibly simulate the model:

```
numpy.random.seed(0)
```

Fifth, use the `simulate` function to run 100-25 s simulations, store the results, and use `matplotlib` to visualize the results. You should see results similar to those below.



Finally, examine the result simulation results. Is the simulation approaching steady-state? How could you analytically calculate the steady-state?

Deterministic, probalistic, and stochastic simulation of mRNA and protein synthesis and degradation

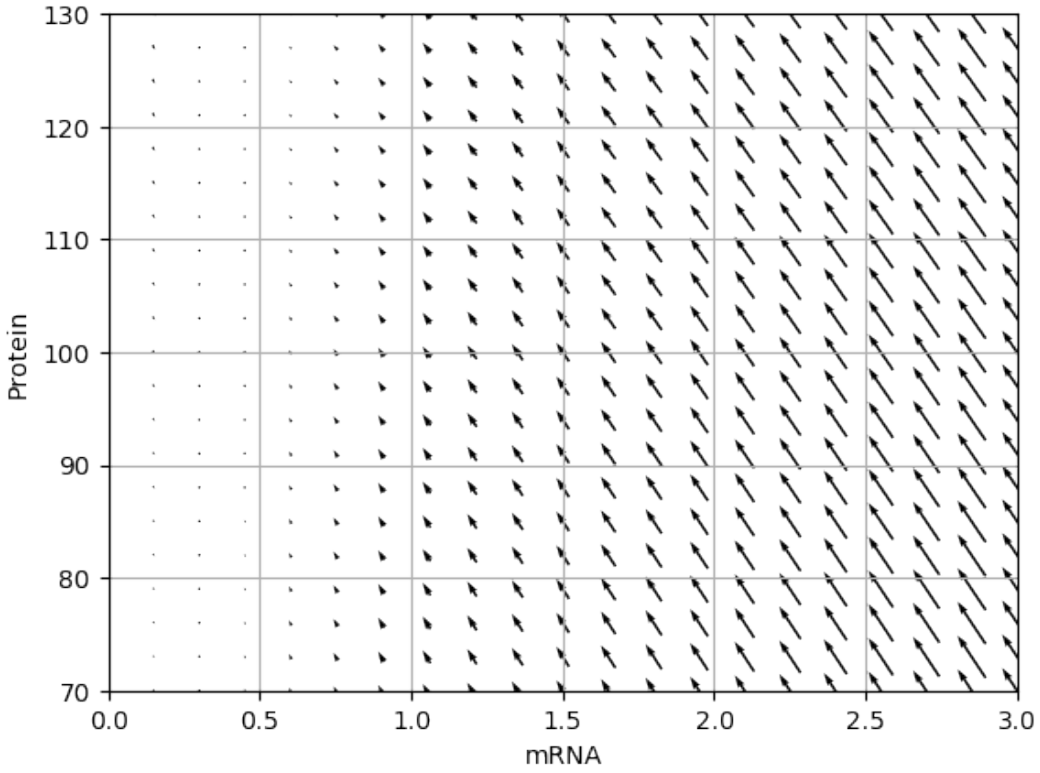
In this exercise, you will compare deterministic, probalistic, and stochastic simulations of a simple model of mRNA and protein synthesis and degradation:

- Zeroth order mRNA synthesis: k_m
- First order protein synthesis: mk_n
- First order RNA degradation: $m\gamma_m$
- First order protein degradation: $n\gamma_n$
- $k_m = 5$ h^{-1}
- $k_n = 20$ molecule^{-1} h^{-1}
- $\gamma_m = \frac{\ln 2}{3}$ min^{-1}
- $\gamma_n = \frac{\ln 2}{10}$ h^{-1}
- $m(t = 0) = 1$
- $n(t = 0) = 98$

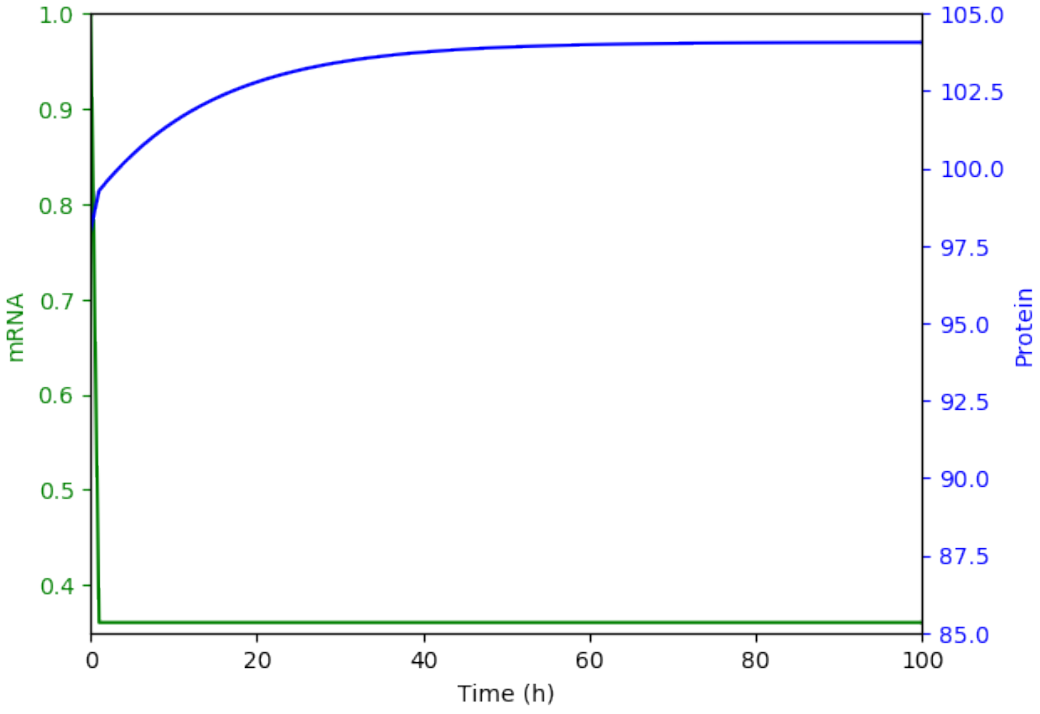
Please follow our solution for more detailed instructions.

1. Implement an ODE simulation of this model

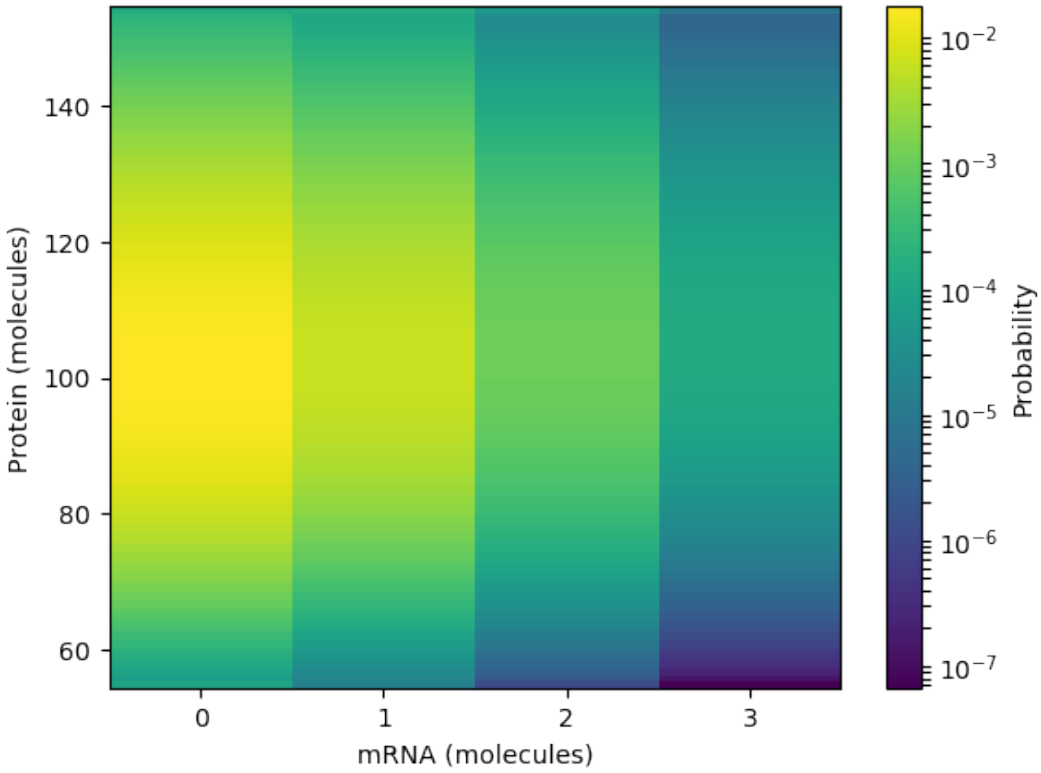
2. Draw the vector field for the ODE model. You should obtain results similar to those below.



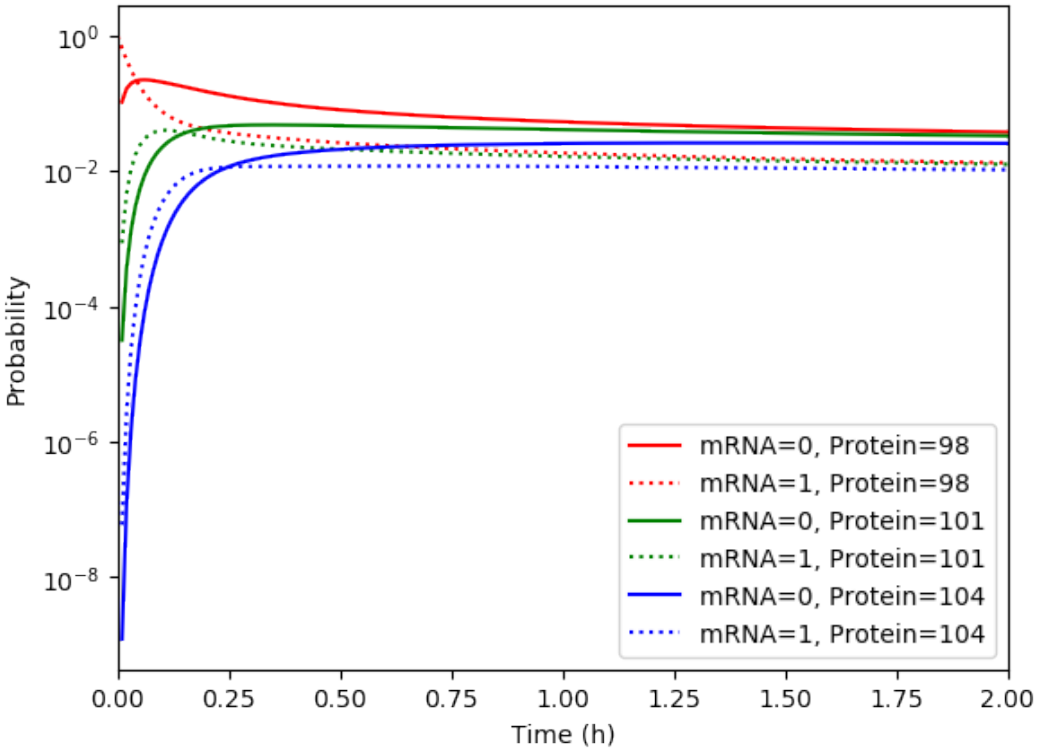
3. Analyze the Jacobian of the ODE model to determine the critical point of the model and its stability
4. Numerically simulate the ODE model. You should obtain results similar to those below.



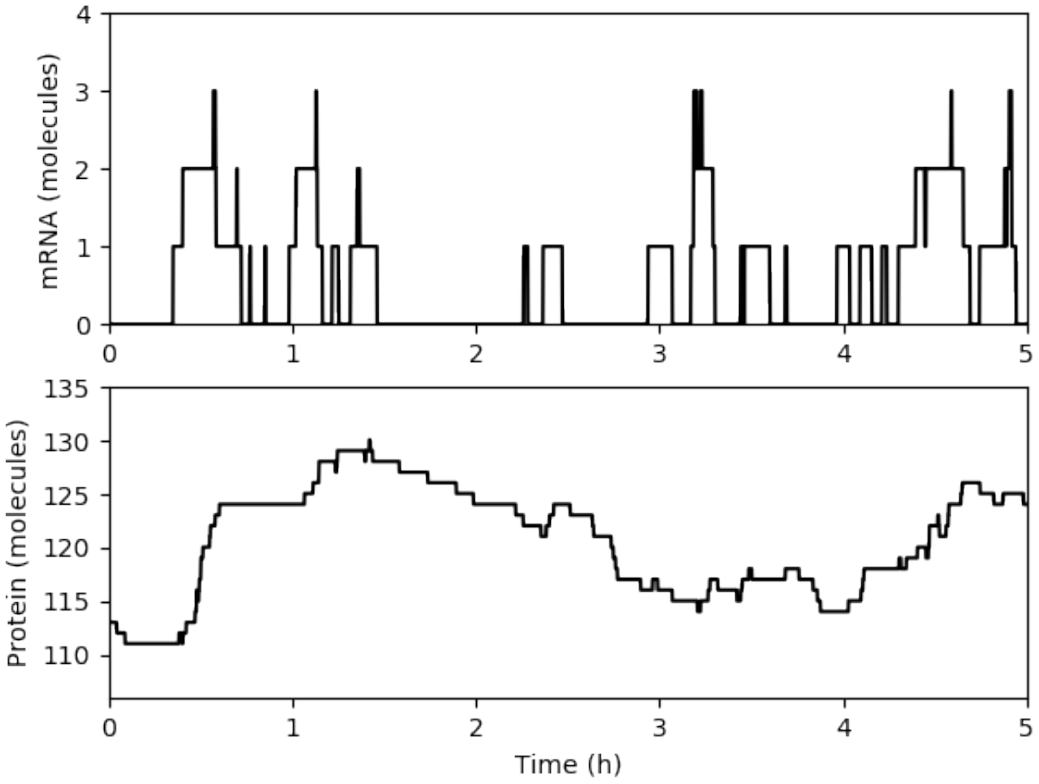
5. Generate the set of chemical master equation ODEs for the same model for $m = 0, 1, ..., 3$ and $n = 54, 55, ..., 154$
6. Calculate and plot the steady probability distribution. You should obtain results similar to those below.



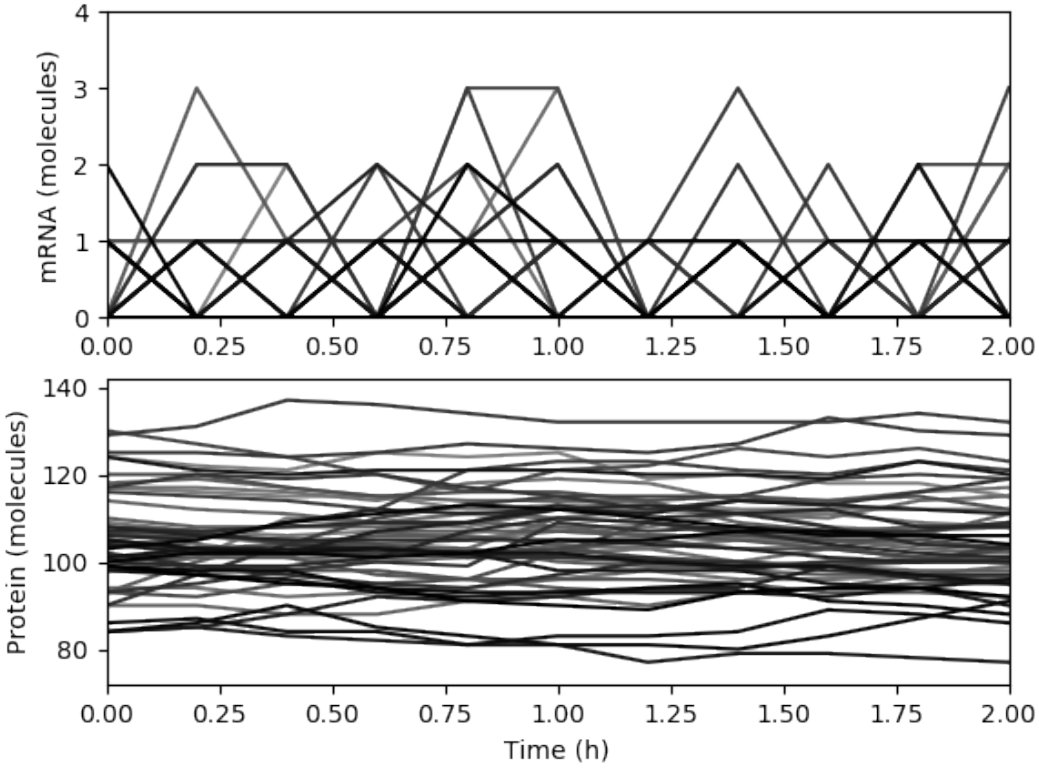
7. Numerically simulate the chemical master equation ODEs with $P_{1,98} = 1$. You should obtain results similar to those below.



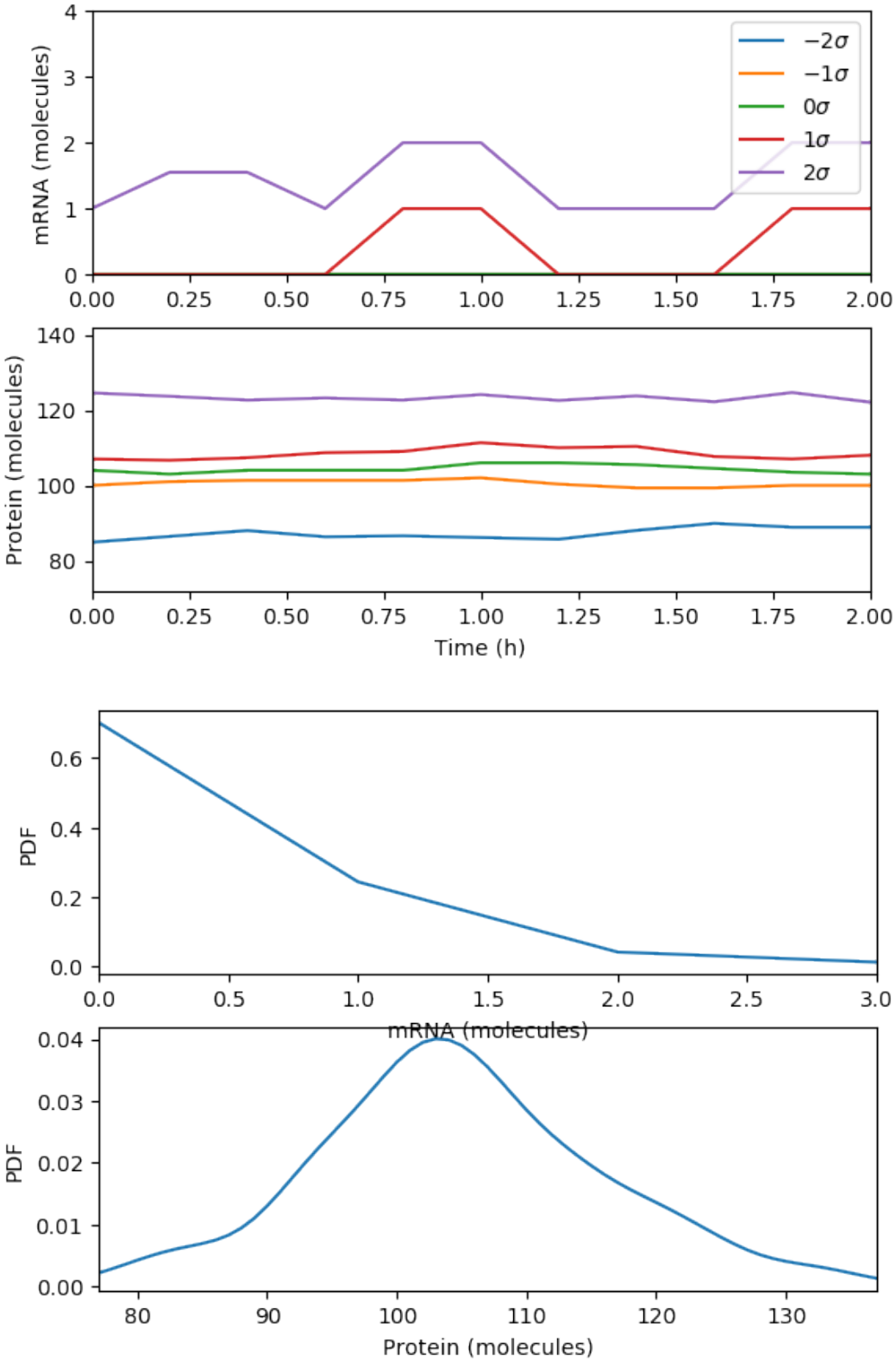
8. Implement an SSA simulation of the same model.
9. Numerically simulate one trajectory. You should obtain results similar to those below.



10. Numerically simulate one trajectory. You should obtain results similar to those below.



11. Average the results over the trajectories and then over time and plot the results. You should obtain results similar to those below.



12. Compare the simulations.

    - What is the steady-state of each of the simulations? How long does it take to reach those steady states?

- Which of the simulations captures protein bursts?

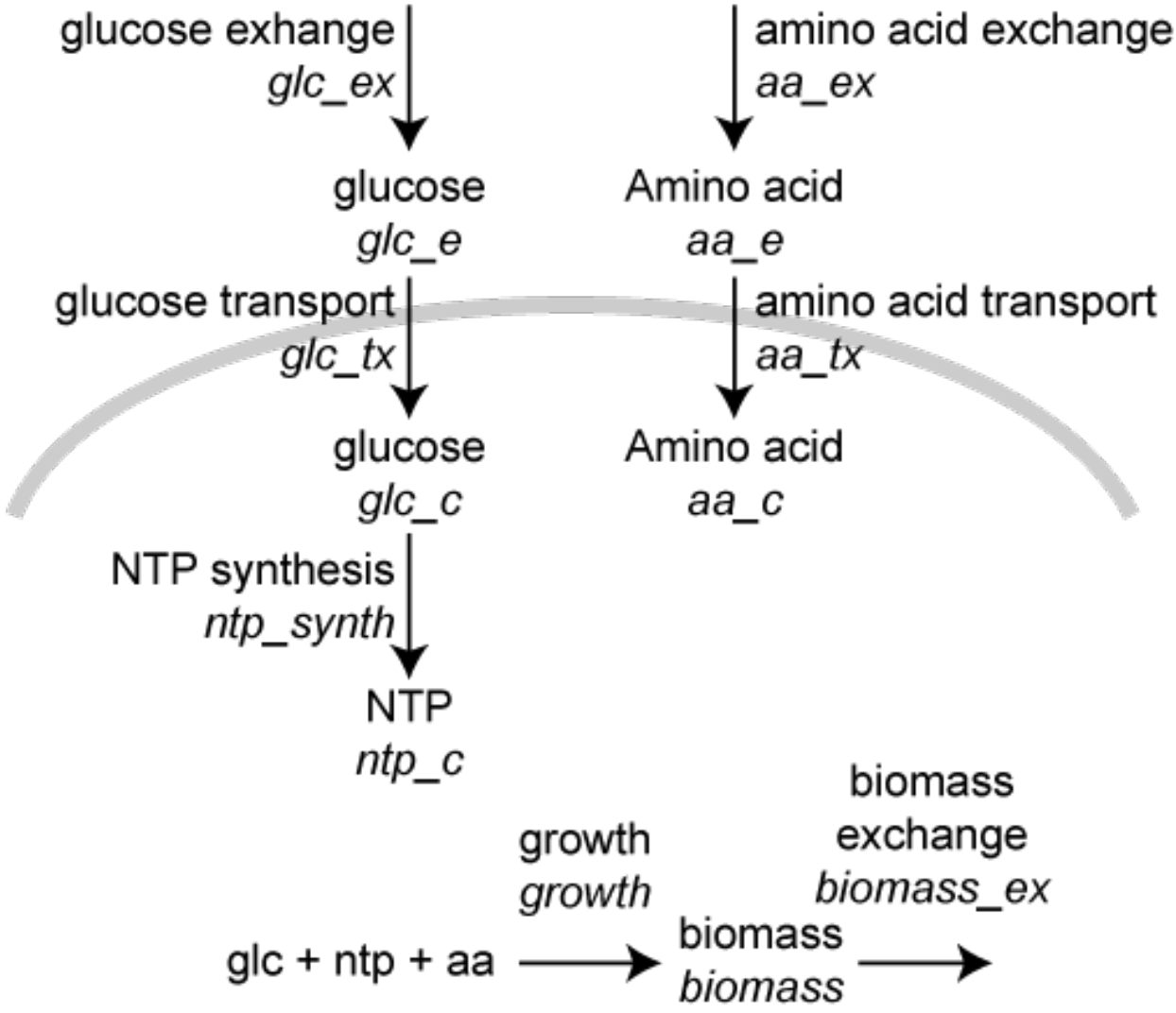
Network-free simulation of rule-based models

Please see the PySB tutorial to learn how to simulate rule-based models from Python.

Dynamic FBA (dFBA) simulation

In this exercise we will use the optlang package, which provides a modeling language for solving mathematical optimization problems, to simulate the mock metabolic dFBA model illustrated below. Instead of setting up an optimization problem with a reaction flux vector and a stoichiometric matrix and using a linear solver, as presented in the Flux balance analysis section above, this approach explicitly declares steady-state constraints on species concentrations and uses a general-purpose solver to optimize growth.

where vjvj is the flux of reaction jj, fμfμ is 1 for the biomass reaction and 0 otherwise, SijSij is the of species



First, create an `optlang` model:

```
# import optlang
import optlang

# create a model
model = optlang.Model()
```

Second, add a variable for each reaction flux. For example, the following two commands will create variables for the glucose transport and exchange reactions:

```
glc_tx = optlang.Variable('glc_tx', lb=0)
glc_ex = optlang.Variable('glc_ex', lb=0)
```

`lb=0` indicates that these variables have a lower bound of `0`.

Next, constrain the rate of change of each species to 0, as FBA assumes. For example, the following command will create a variable for the rate of change of the extracellular concentration of glucose and constrain that rate to zero by constraining as the difference between the fluxes of reaction(s) that produce the specie and those that consume it to 0:

```
glc_e = optlang.Constraint(glc_ex - glc_tx, lb=0, ub=0)
model.add([glc_e])
```

Next, set the objective to maximize growth:

```
model.objective = optlang.Objective(growth, direction='max')
```

Next, set the initial conditions:

```
init_conc_glc_e = 200.
init_conc_aa_e = 120.
init_conc_biomass = 1.
```

Next, setup a data structure to store the predicted concentrations:

```
time_max = 70
time_hist = numpy.array(range(time_max + 1))
flux_hist = numpy.full((time_max + 1, len(model.variables)), numpy.nan)
growth_hist = numpy.full(time_max + 1, numpy.nan)
conc_hist = numpy.full((time_max + 1, 3), numpy.nan)
```

Next, complete the code fragment below to simulate the model:

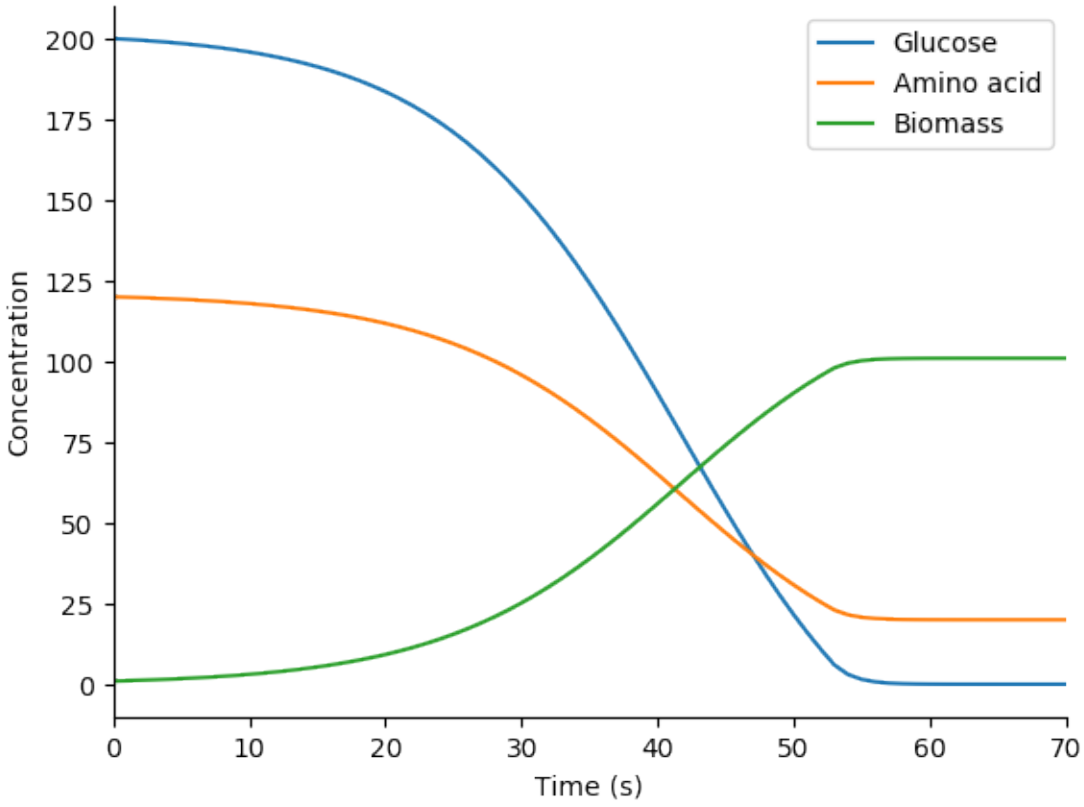
```
variable_names = model.variables.keys()
conc_glc_e = init_conc_glc_e
conc_aa_e = init_conc_aa_e
conc_biomass = init_conc_biomass
for i_time in range(time_max + 1):
    # constrain fluxes based on avialable nutrients
    glc_tx.ub = ...
    aa_tx.ub = ...

    # solve for the maximum growth and optimal fluxes
    status = model.optimize()
    assert(status == 'optimal')

    # store history
    growth_hist[i_time] = model.objective.value
    for i_var, var_name in enumerate(variable_names):
```

(continues on next page)

(continued from previous page)

```
        flux_hist[i_time, i_var] = model.variables[var_name].primal

    conc_hist[i_time, 0] = conc_glc_e
    conc_hist[i_time, 1] = conc_aa_e
    conc_hist[i_time, 2] = conc_biomass

    # update concentrations
    conc_glc_e -= ...
    conc_aa_e -= ...
    conc_biomass += ...
```

Next, use `matplotlib` to plot the predicted concentration dynamics. You should see results similar to those below.

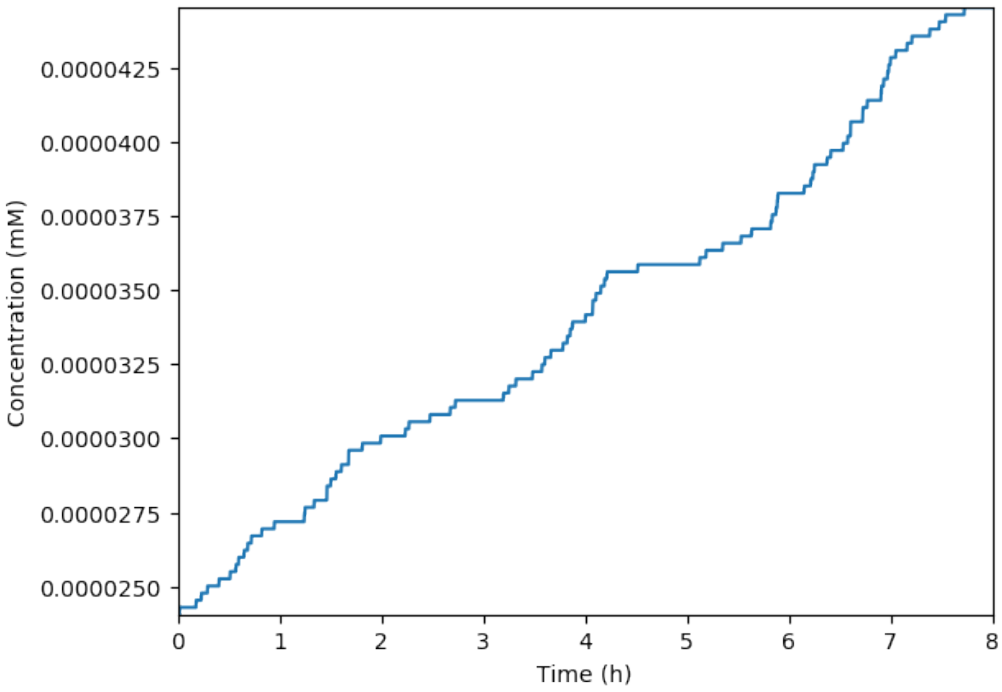


Hybrid simulation

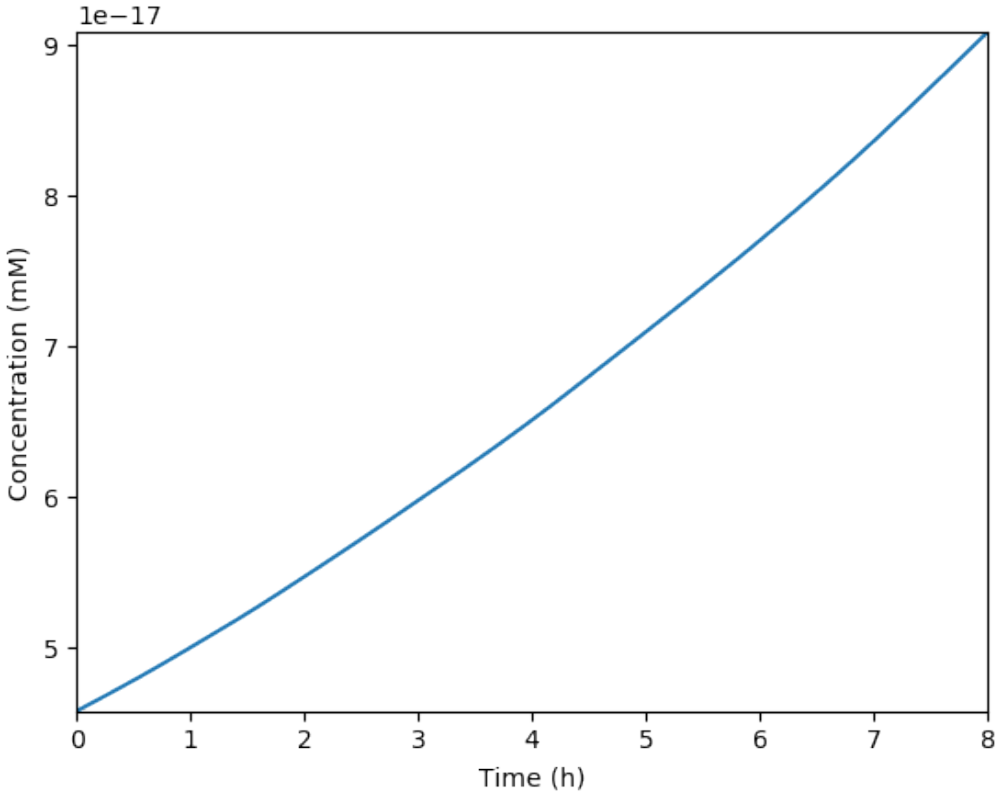
In this exercise you will implement a hybrid simulation of metabolism, transcription, translation, and RNA degradation.

Please follow our solution for more detailed instructions.

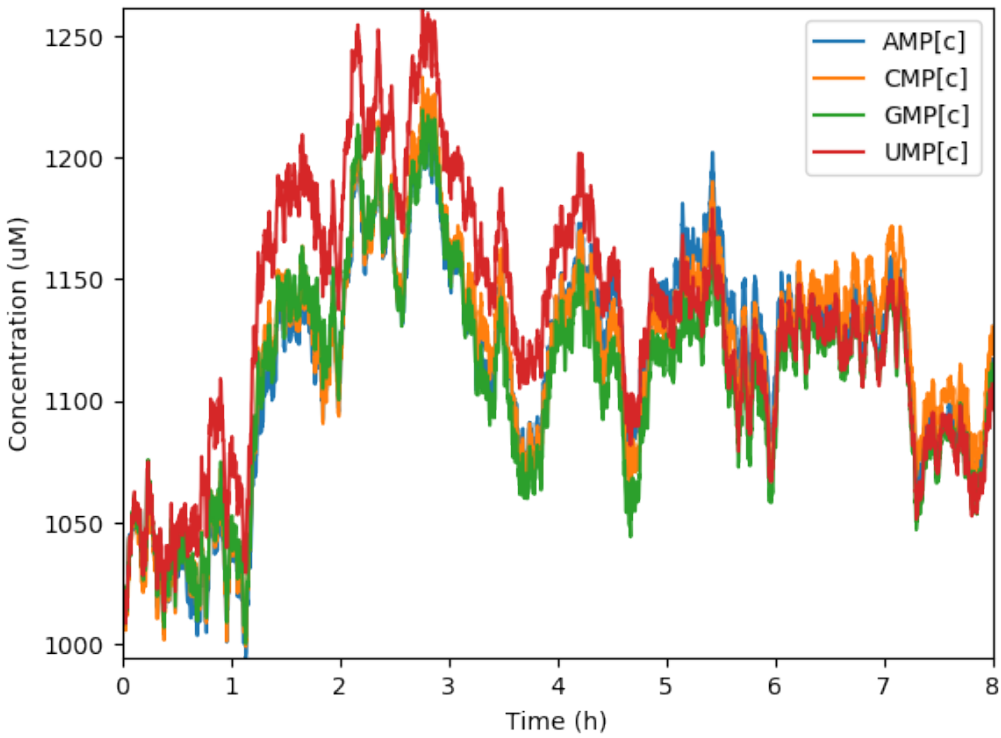
1. Implement an hybrid SSA/FBA simulation of this model
2. Plot the predicted growth, volume, NMPs, NTPs, amino acids, protein, and RNA. You should see results similar to those below.
    - Predicted growth



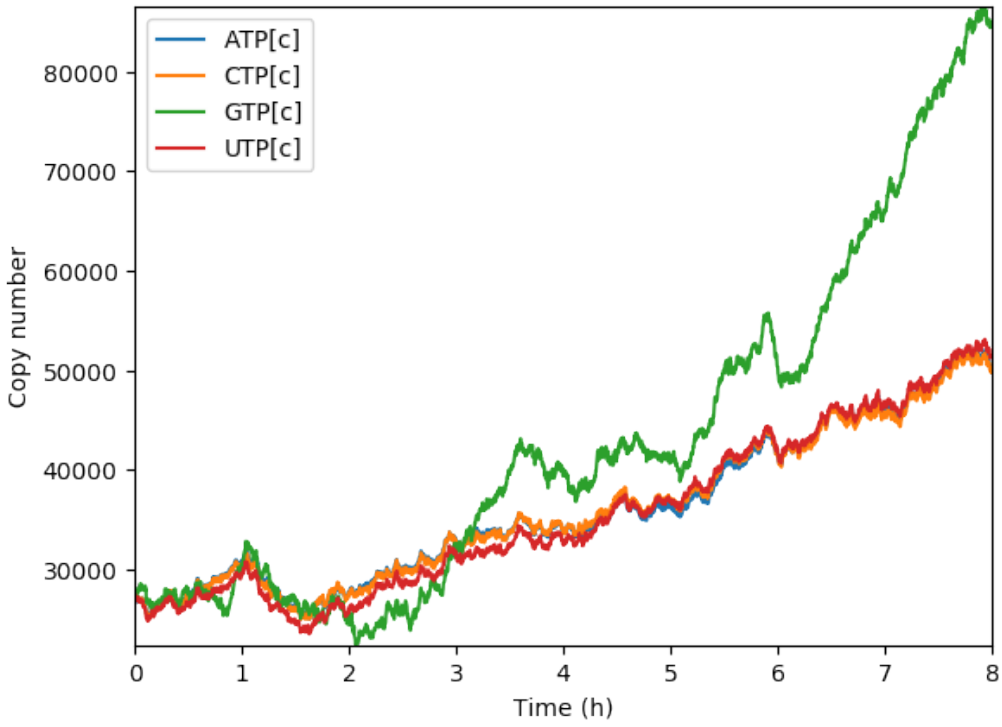
- Predicted volume



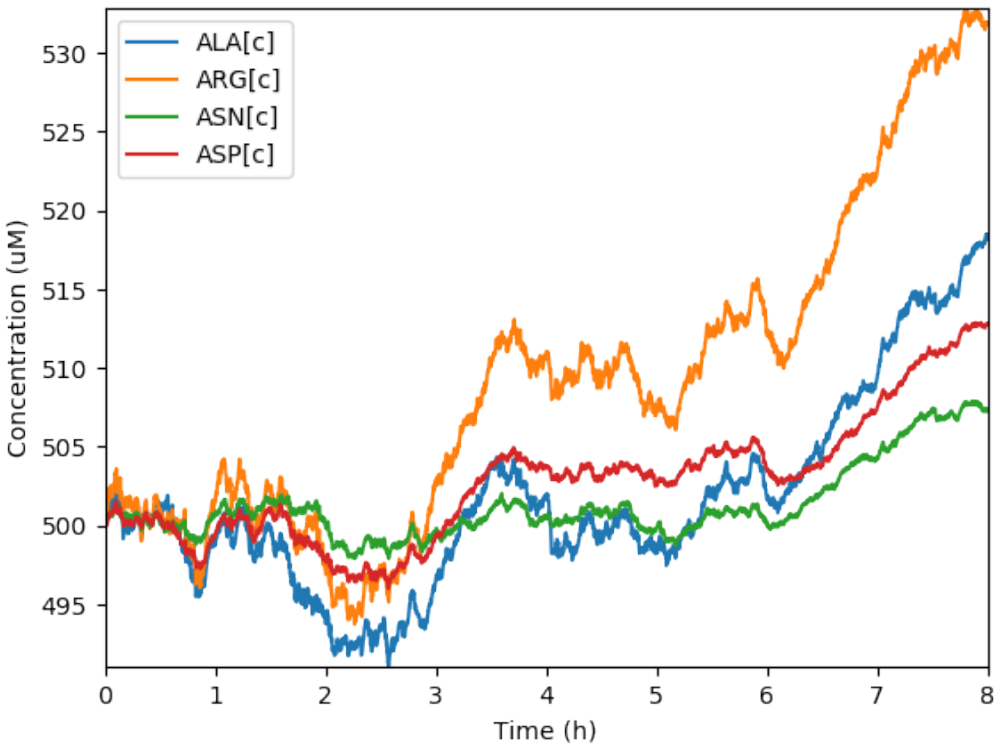
- Predicted NMPs



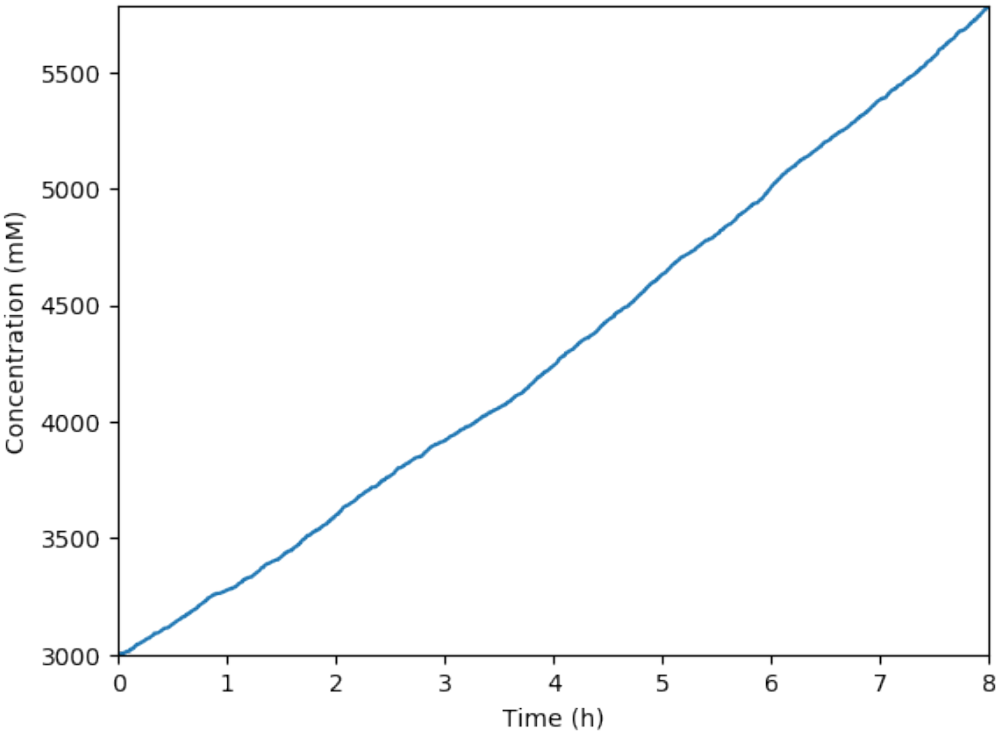
- Predicted NTPs



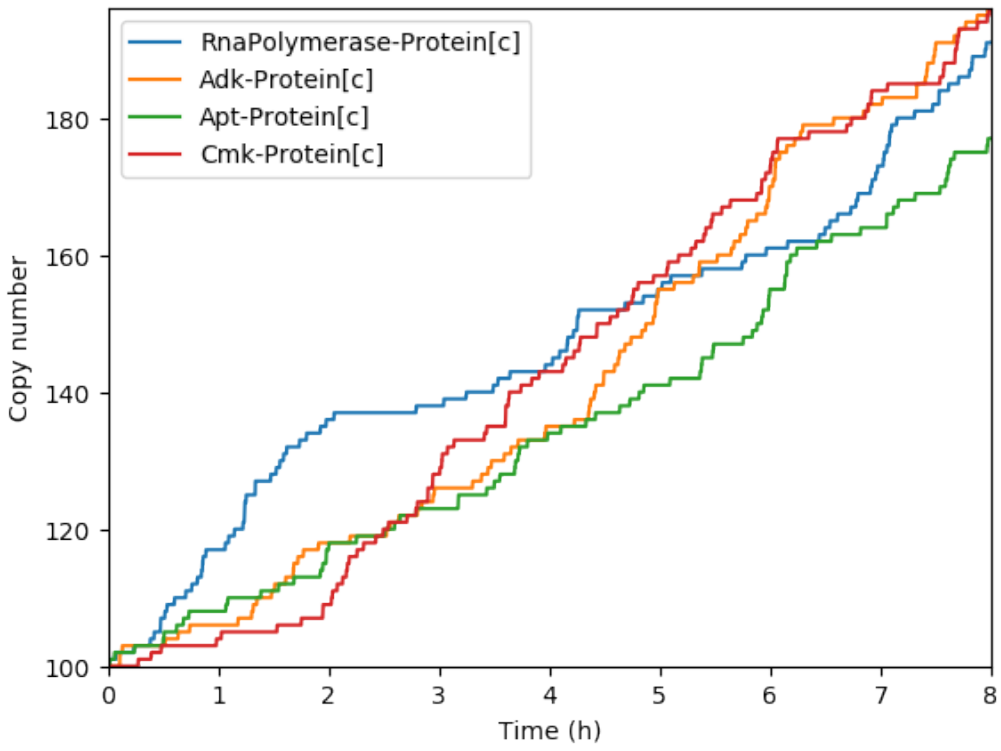
- Predicted amino acids



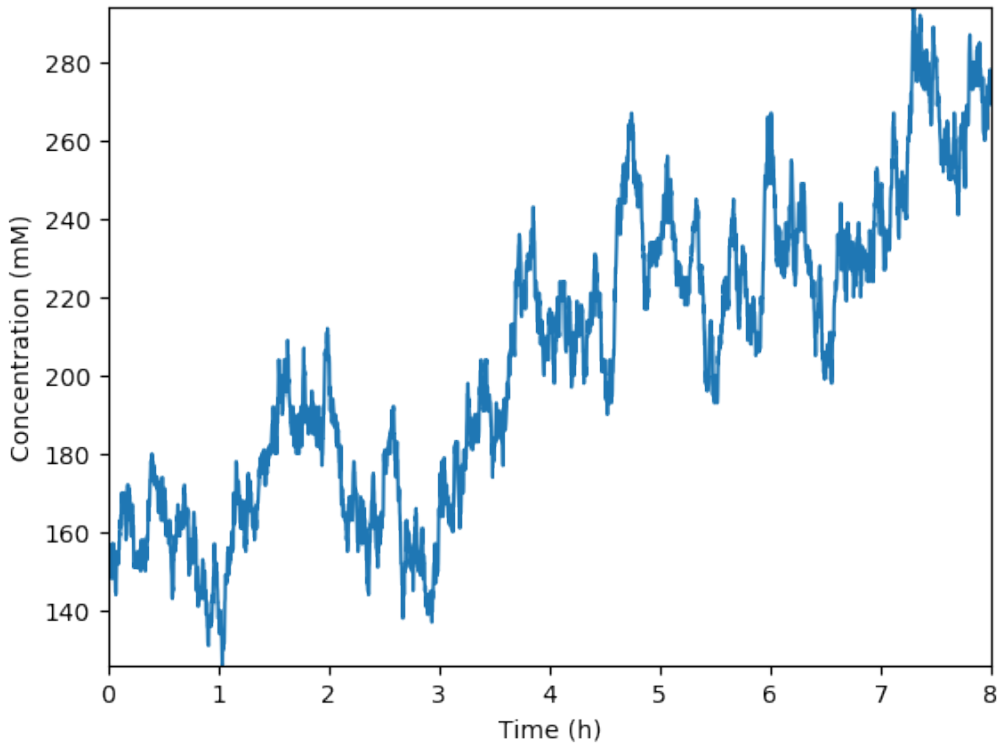
- Predicted total protein



- Predicted key proteins



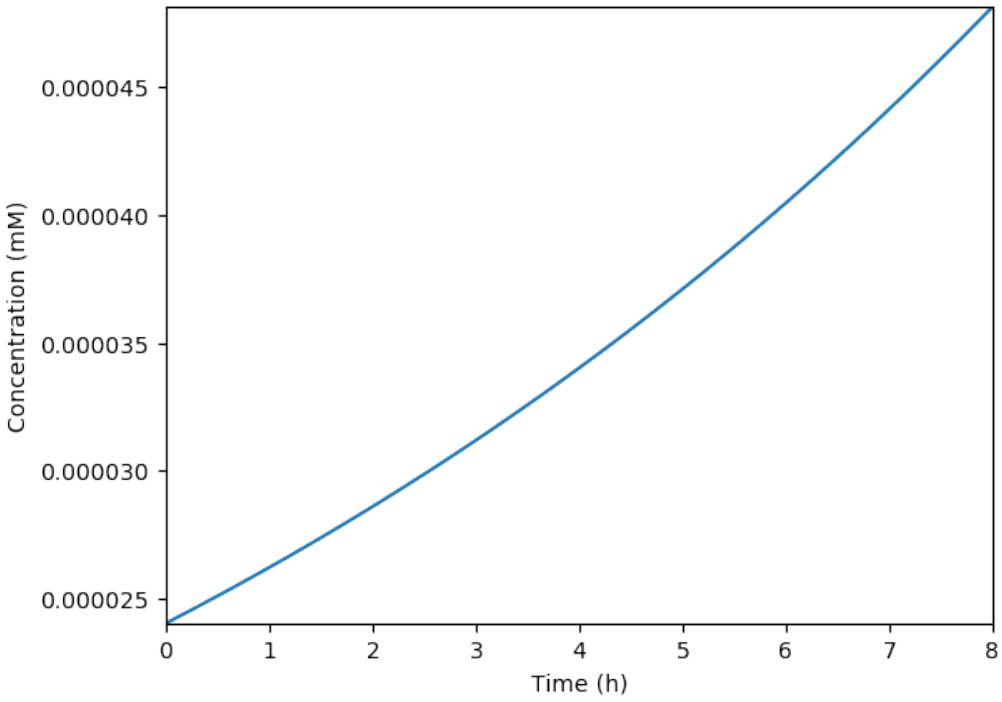
- Predicted total RNA



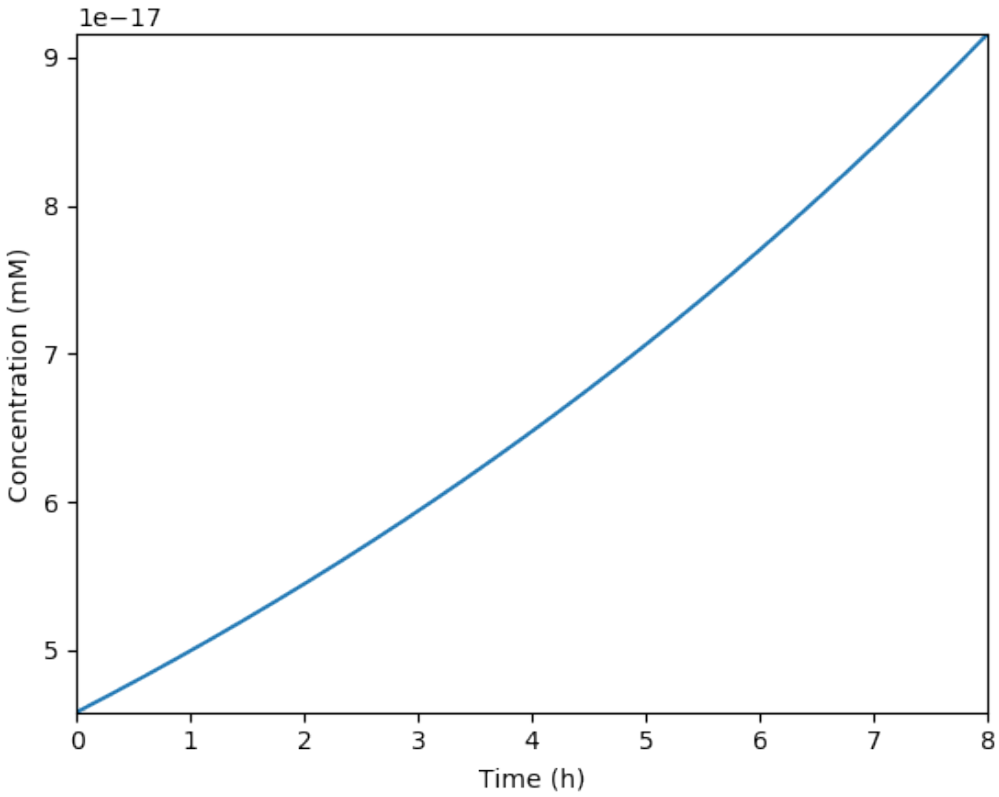
3. Implement a simulation of just the metabolism submodel coupled to mocked versions of the transcription, translation, and RNA degradation submodels.

4. Plot the predicted growth, volume, NMPs, NTPs, amino acids, protein, and RNA. You should see results similar to those below.

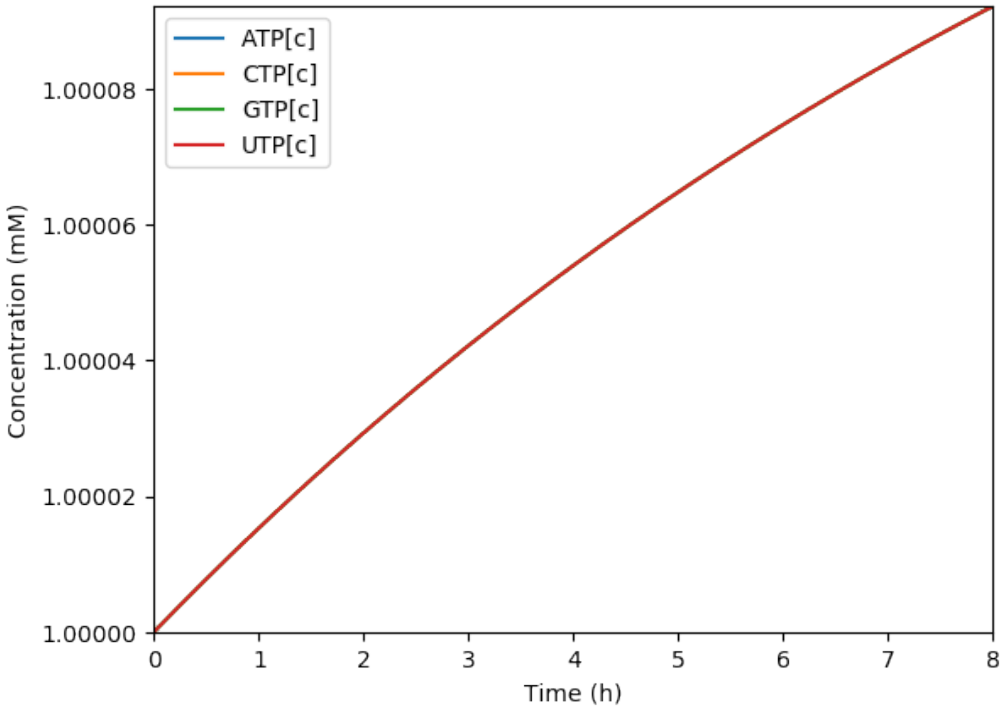
    - Predicted growth



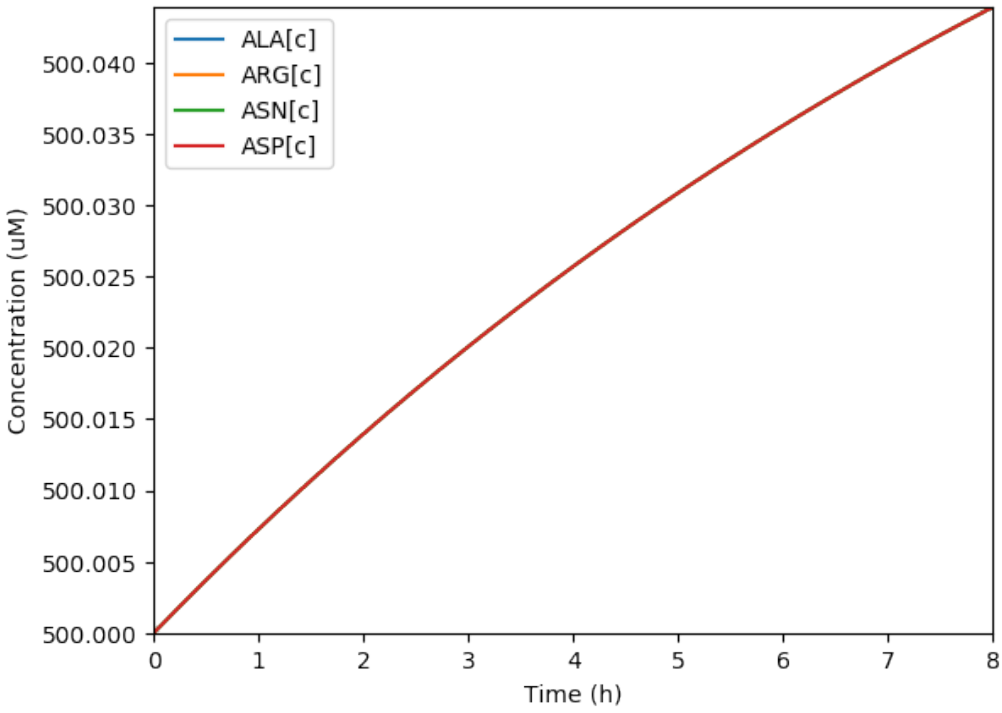
    - Predicted volume



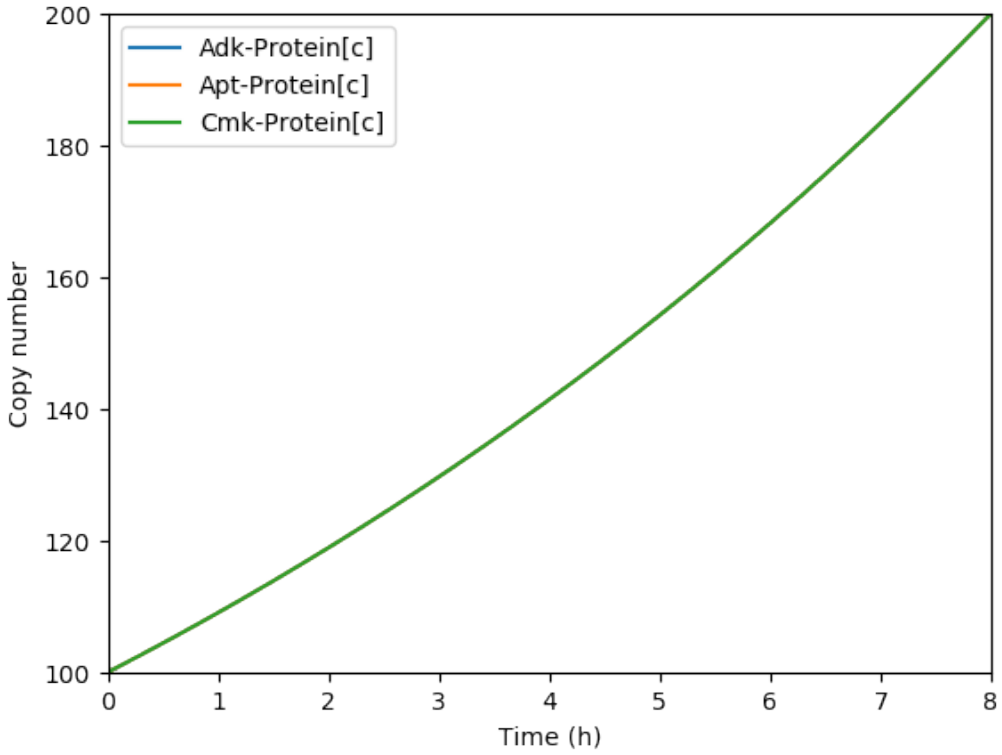
- Predicted NTPs



- Predicted amino acids



- Predicted key proteins



3.9 Model testing

To generate confidence in a model's predictive capabilities, it is essential to simulate the model and validate that the model makes accurate predictions or debug the until it does make accurate predictions. Formally, model testing can be divided into two stages: *validating* that a model accurate captures the known biology by confirming that the model recapitulates previous experimental observations and 'verifying' that a model accurately predicts previously unknown biology by confirming that the model can accurately predict the outcomes of new experiments.

Because models are engineered software systems, the same approaches that have been developed to validate software can also be used to validate models. This includes both unit testing and formal verification. Within each approach, it is helpful to begin by testing the behaviors of individual model components and then test increasingly large assemblies of components. See Section 2.2.9 for model information about how to use unit testing to validate software and models.

3.9.1 Testing composite models

Composite models can be tested by first testing each submodel and then testing the combined model. In general, individual submodels should be tested by mocking their interfaces with the other submodels.

3.9.2 Exercise

1. Build a model of the stochastic dynamics of the expression of a single transcript and the cell volume over the cell cycle
2. Simulate the model using the Gillespie algorithm
3. Calibrate the model so that the average transcript copy number doubles on the same time scale as that of the volume
4. Use unit testing to validate that the model is consistent with cell theory (i.e. that the transcript copy number and volume grow at the same time scale)

3.10 Logging simulation results

Broadly, there are two approaches to logging and analyzing simulation results: (a) log specific observables for specific analyses and (b) log all observables and organize the data to facilitate all possible analyses. The first approach is comparatively simple to implement, generates results sets of modest sizes, and doesn't require specialized software for organizing and analyzing simulation results. However, this approach couples simulation and the analysis of simulation results and thus requires modelers to rerun simulations each time they want to run a different analysis. We recommend following the latter approach. This approach decouples simulation from the analysis of simulations which makes it easier for models to explore analyses. This is particularly helpful when you don't yet know what you would like to analyze or where you would like to learn interesting biology from unbiased analysis of simulation data.

3.11 Organizing simulation results

To facilitate analysis of large simulation result sets including identifying relevant simulations to analyze, retrieving slices of simulation results, and calculating statistics about simulation results, it is helpful to organize simulation results into a database. Such databases should store simulation results, as well as all of the metadata needed to understand and reproduce the simulation results.

Currently, there are only a few primitive database for simulation results including

- Bookshelf
- Dynameomics
- SEEK
- WholeCellSimDB

3.11.1 Exercise

Read the WholeCellSimDB schema.

3.12 Quickly analyzing large simulation results

As described above, currently there are only a few primitive database for organizing simulation results. In addition to develop better simulation results databases, we must develop better tools for search, extracting, and reducing data stored in such databases. We recommend that such tools be developed using distributed computing frameworks such as Spark.

3.13 Rule-based Modeling with BioNetGen, BNGL, and RuleBender

3.13.1 Description of Rule-based Modeling, BioNetGen, BNGL, and RuleBender

As the Summary of the The BioNetGen Online Manual[1] says:

> Rule-based modeling involves the representation of molecules as structured objects and molecular interactions as rules for transforming the attributes of these objects. The approach is notable in that it allows one to systematically incorporate site-specific details about protein-protein interactions into a model for the dynamics of a signal-transduction system, but the method has other applications as well, such as following the fates of individual carbon atoms in metabolic reactions. The consequences of protein-protein interactions are difficult to specify and track with a conventional modeling approach because of the large number of protein phosphoforms and protein complexes that these interactions potentially generate.

3.13.2 BNGL and RuleBender Basic Functionality

This section demonstrates the basic functionality of the BioNetGen language (BNGL) and RuleBender, which is the BioNetGen IDE.

Obtain BioNetGen, RuleBender, and NFsim by following the directions at BioNetGen quick start. Java and Perl must be installed. The BioNetGen Quick Reference Guide summarizes much of BioNetGen and its ecosystem in four pages.

Run RuleBender, which can:

- Edit and save BNGL models
- Create new BNGL models
- Run BNGL models in a simulator
- Print simulator debugging and logging outputs
- Plot species population predictions generated by simulation

[1] Contributors: James R. Faeder, Michael L. Blinov, William S. Hlavacek, Leonard A. Harris, Justin S. Hogg, Arshi Arora

- Diagram and inspect molecular types and rules

RuleBender comes with many sample models written in BNGL. Open the birth-death model by clicking on File -> New BioNegGen Project -> OK -> Select a Sample -> birth-death -> Type Project Name 'birth-death_test'.

The birth-death model contains just one specie, `A`, with no initial population:

```
begin species
  A() 0
end species
```

And it contains just two rules:

```
birth: 0 -> A() kp1
death: A() -> 0 km1
```

It's a simple model. Birth creates species `A`, while death destroys it. Irreversible rules have the form:

```
rule_name: reactants -> products forward_rate_law
```

Reversible rules have the form:

```
rule_name: reactants <-> products forward_rate_law, reverse_rate_law
```

The null (`0`) reactants and products in these rules allow `A` to be created from nothing and destroyed without a trace. These rules aren't mass balanced, of course, but ignore that for now.

Simulate the model. Both ODE and SSA solvers are run, each for 50 seconds:

```
saveConcentrations()
simulate({suffix=>"ode",method=>"ode",t_end=>50,n_steps=>500})
resetConcentrations()
simulate({suffix=>"ssa",method=>"ssa",t_end=>50,n_steps=>500})
```

Before you look at the results, try to answer these questions:

- What population do you expect the model will predict for `A`?
- How do you expect the population to vary over the 50 sec for the ODE and SSA models?

Now look at the results by clicking on the `birth-death_ode.gdat` and `birth-death_ssa.gdat` and tabs. Did you anticipate the model's predictions?

The ODE solver predicts that the population of `A` grows smoothly and rises asymptotically to 50. (The y-axis label says 'Concentration', but it should say 'population'. The model doesn't have a volume, so it cannot calculate concentration.)

Does this make sense? The system reaches equilibrium when `A` is created at the same rate that it is destroyed. In equilibrium, $dA(t)/dt = 0$. The `birth` rule creates A at the constant rate of `kp1` or 10/sec. And the `death` rule destroys the existing A at a rate of `km1` or 0.2 of per second. In this continuous ODE model, the quantity of A is then given by

$$A(t) = 10t - 0.2At$$

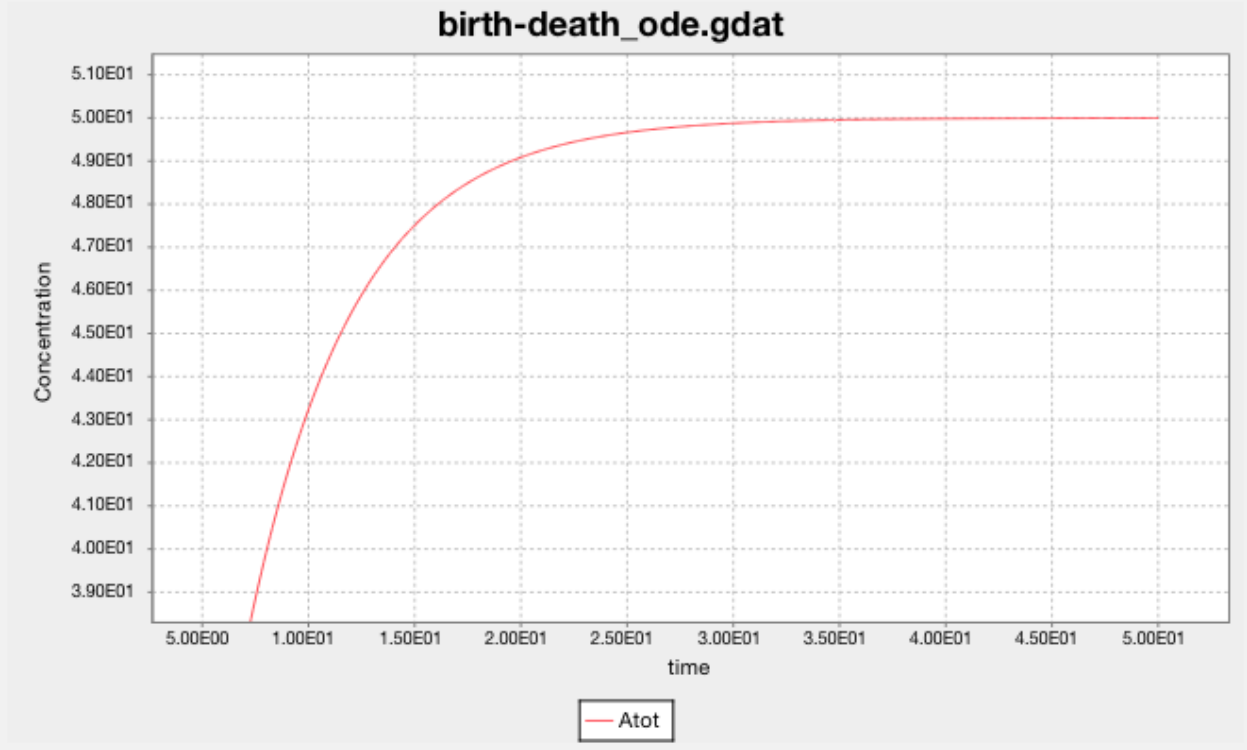
Taking the time derivative and setting it equal to 0, we get

$$0 = 10 - 0.2A$$

whose solution is $A = 50$, as predicted by the ODE solver.

Note that ODE integrators approximate species populations (or concentrations) as non-negative real numbers, whereas populations are really non-negative integers. You can visualize this in RuleBender by zooming in on the

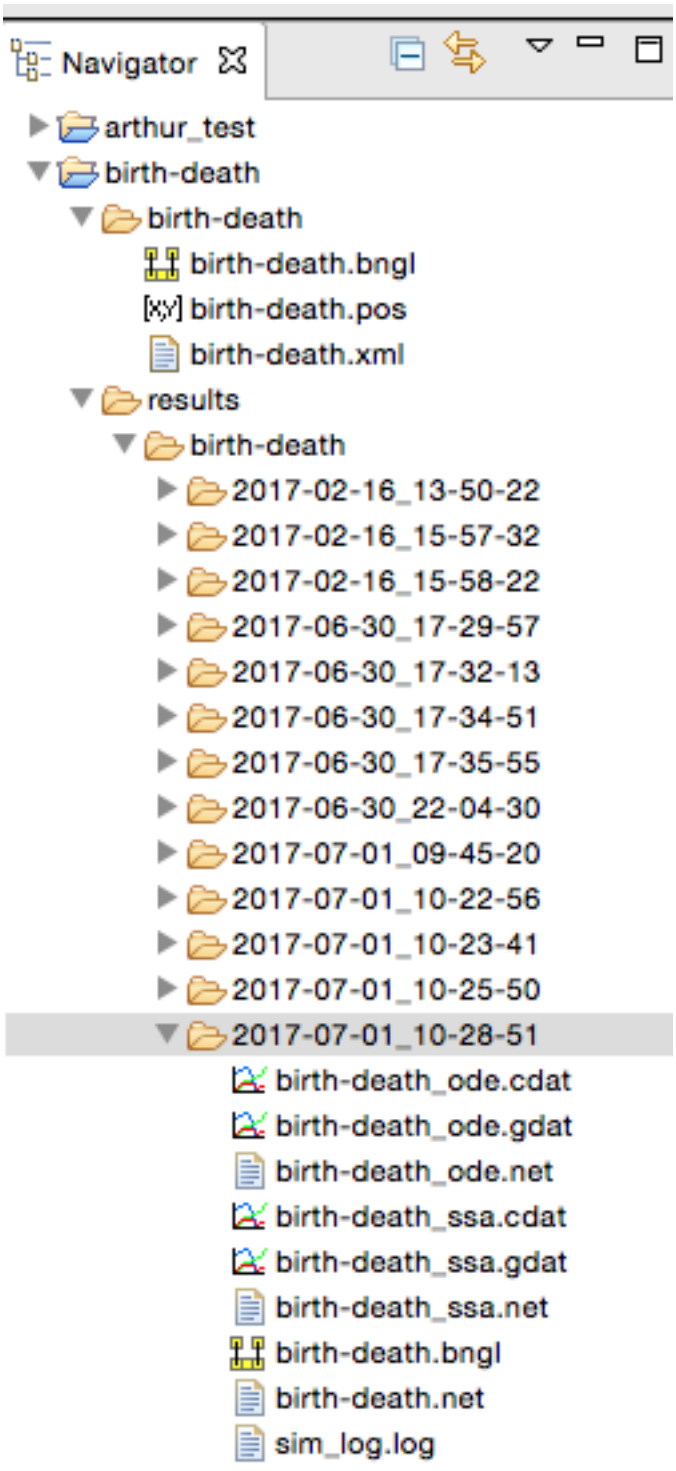
`birth-death_ode.gdat` plot. Select a rectangle of the plot from upper-left to lower-right to see the smooth, real-valued predictions.



Look at SSA's predictions for the population of `A`. What happens when you greatly increase the simulation's end time (`t_end`)?

Zooming in shows that SSA predicts the population as a non-negative integer at each sampled time step.

You can find the raw inputs and outputs for each simulation run in the results subdirectory of the `birth-death` project.



Raw predictions are in `cdat` files. Data for the above plot are in `birth-death_ssa.cdat`:

```
3.860000000000e+01  5.100000000000e+01
3.880000000000e+01  5.300000000000e+01
3.900000000000e+01  5.300000000000e+01
3.920000000000e+01  5.300000000000e+01
3.940000000000e+01  5.600000000000e+01
```

(continues on next page)

(continued from previous page)

```
3.960000000000e+01  5.500000000000e+01
3.980000000000e+01  5.500000000000e+01
4.000000000000e+01  5.500000000000e+01
4.020000000000e+01  5.300000000000e+01
4.040000000000e+01  5.300000000000e+01
4.060000000000e+01  5.500000000000e+01
4.080000000000e+01  5.400000000000e+01
4.100000000000e+01  5.300000000000e+01
4.120000000000e+01  5.300000000000e+01
4.140000000000e+01  5.300000000000e+01
```

3.13.3 Exercises

What happens when you change the rate law constants? Adjust them so the equilibrium population is 200. Adjust them so that the model does not reach an equilibrium. Can they be adjusted to reduce the variance of the SSA predictions?

Debugging models is easier when re-executing a simulation run produces identical output, that is, reproduces a previous run. Reproducibility is also an important feature of the scientific method. Can you program RuleBender and BNGL to exactly reproduce these SSA predictions? (I'm not aware of a way to do so.)

3.13.4 Molecular sites, their states, and bonds

In this section we create a new BNGL model. Follow chapter 3 of the Rule-Based Modeling of Signal Transduction: A Primer. Make a new project by clicking on and typing File -> New BioNegGen Project -> OK -> Select a Sample -> `template.bngl` -> Type Project Name 'signal_transduction'. `template.bngl` provides a template BNGL program with initialized blocks.

CHAPTER 4

Principles and methods of WC modeling

4.1 Units of WC models

Table 4.1 lists the units of each quantity in a WC model.

These units facilitate composition of multiple models of multiple cellular processes. In particular, these units represent dFBA models in volume (l) concentrations rather than in mass (gDCW) concentrations because this makes dFBA compatible with SSA and ODE models that are typically described in volume concentrations.

Table 4.1: Units of WC models.

Quantity	Attributes	Dimensions	Units	Units class
Time	`wc_lang.core.Model.time_units`	time	s	`wc_lang.core.TimeUnit`
Compartment density	`wc_lang.core.Compartment.density` `wc_lang.core.Compartment.density_units`	mass / volume	g l^-1	`wc_lang.core.DensityUnit`
Compartment volume	`wc_lang.core.Compartment.volume` `wc_lang.core.Compartment.volume_units`	volume	l^-1	`wc_lang.core.VolumeUnit`
Distribution of initial concentration of a species at the beginning of the cell cycle	`wc_lang.core.Concentration.distribution` `wc_lang.core.Concentration.mean` `wc_lang.core.Concentration.std` `wc_lang.core.Concentration.units`	molecule, molecule volume^{-1}	molecule, M	`wc_lang.core.ConcentrationUnit`
Species count	`wc_lang.core.Species.units`	molecule	molecule	`wc_lang.core.MoleculeCountUnit`
Observable count	`wc_lang.core.ObservableExpression.expression` `wc_lang.core.Observable.units`	molecule	molecule	`wc_lang.core.MoleculeCountUnit`
Observable coefficient	`wc_lang.core.ObservableExpression.expression`	*dimensionless*	*none*	`wc_lang.core.ObservableCoefficientUnit`
Reaction participant	`wc_lang.core.SpeciesCoefficient.coefficient`	*dimensionless*	*none*	`wc_lang.core.ReactionParticipantUnit`
Reaction rate	`wc_lang.core.RateLawExpression.expression`	time^{-1}	s^{-1},	`wc_lang.core.ReactionRateUnit`

4.2 Using the wc_lang package to define whole-cell models

This tutorial teaches you how to use the wc_lang package to access and create whole-cell models. wc_lang provides a foundation for defining, writing, reading and manipulating biochemical models composed of species, reactions, compartments and other parts of a biochemical system. It can be used to define models of entire cells, or models of smaller biochemical systems. wc_lang contains methods to read and write models from two types of files – Excel spreadsheet workbooks and sets of delimited files. It also includes methods that analyze or transform models – e.g., methods that validate, compare, and normalize them.

wc_lang depends heavily on the obj_tables package which defines a generic language for declaring interrelated Python objects, converting them to and from data records, transferring the records to and from files, and validating their values. obj_tables is essentially an object-relational mapping (ORM) system that stores data in files instead of databases. However, users of wc_lang do not need to use obj_tables directly.

4.2.1 Semantics of a wc_lang biochemical Model

A wc_lang biochemical model represents a biochemical system as Species (we indicate classes in wc_lang by capitalized names in fixed-width text) that get transformed by reactions.

A SpeciesType describes a biochemical molecule, including its name (following Python convention, attributes of classes are lowercase names), structure, molecular_weight, charge and other properties. The concentration of a SpeciesType in a compartment is stored by a Species instance that references instances of SpeciesType, Compartment, and Concentration, which provide the Species' location and concentration. A compartment may represent an organelle or a conceptual region of a model. Adjacency relationships among compartments are implied by reactions that transfer species among them, but physical relationships between compartments or their 3D positions are not represented.

The data in a wc_lang model is organized in a highly-interconnected graph of related Python objects, each of which is an obj_tables.core.Model instance. For example, a Species instance contains reaction_participants, which references each Reaction in which the Species participates. The graph contains many convenience relationships like this, which make it easy to follow the relationships between obj_tables.core.Model instances anywhere in a wc_lang model.

A wc_lang model also supports some metadata. Named Parameter entities store arbitrary values, such as input parameters. Published data sources used by a model should be recorded in Reference entities, or in a DatabaseReference objects that identify a biological or chemical database.

wc_lang models are typically used to describe the initial state of a model – a wc_lang description lacks any notion of time. More generally, a comprehensive wc_lang model should provide a complete description of a model, including its data sources and comments about model components.

4.2.2 wc_lang Classes Used to Define biochemical Models

This subsection enumerates the obj_tables.core.Model classes that store data in wc_lang models.

When using an existing model the attributes of these classes are frequently accessed, although their definitions are not typically imported. However, they must be imported when they are being instantiated programmatically.

Many of these classes implement the methods deserialize() and serialize(). deserialize() parses an object's string representation – as would be stored in a text file or spreadsheet representation of a biochemical model – into one or more obj_tables.core.Model instances. serialize() performs the reverse, converting a wc_lang class instance into a string representation. Thus, the deserialize() methods are used when reading models from files and serialize() is used when writing a model to disk. deserialize() returns an error when a string representation cannot be parsed into a Python object.

Static Enumerations

Static attributes of these classes are used as attributes of `wc_lang` model components.

`TaxonRank` The names of biological taxonomic ranks: *domain*, *kingdom*, *phylum*, etc.

`SubmodelAlgorithm` The names of algorithms that can integrate submodels: *dfba*, *ode*, and *ssa*.

`SpeciesTypeType` Types of species types: *metabolite*, *protein*, *dna*, *rna*, and *pseudo_species*.

`RateLawDirection` The direction of a reaction rate law: *backward* or *forward*.

`ReferenceType` Reference types, such as *article*, *book*, *online*, *proceedings*, etc.

`wc_lang` Model Components

These classes are instantiated as components of a `wc_lang` model. When a model is stored on disk all the instances of each class are usually stored in a separate table, either an Excel workbook's worksheet or delimiter-separated file. In the former case, the model is stored in one workbook, while in the latter it is stored in a set of files.

`Taxon` The taxonomic rank of a model.

`Submodel` A part of a whole-cell model which is to be simulated with a particular `algorithm` from the enumeration `SubmodelAlgorithm`. Each `Submodel` is associated with a `Compartment` that contains the `Species` it models, and all the reactions that transform them. A `Submodel` may also have parameters.

`Compartment` A named physical container in the biochemical system being modeled. It could represent an organelle, a cell's cytoplasm, or another physical or conceptual structure. It includes an `initial_volume` in liters, and references to the initial concentrations of the `Species` it contains. A compartment can have a semi-permeable membrane, which is modeled by reactions that transform reactant species in the compartment to product species in another compartment. These are called *membrane-transfer* reactions. A membrane-transfer reaction that moves species from compartment *x* to compartment *y* implies that *x* and *y* are adjacent.

`SpeciesType` The biochemical type of a species. It contains the type's `name`, `structure` – which is represented in InChI for metabolites and as sequences for DNA, RNA, and proteins, `empirical_formula`, `molecular_weight`, and `charge`. A species' `type` is drawn from the attributes of `SpeciesTypeType`.

`Species` A particular `SpeciesType` contained in a particular `Compartment` at a particular concentration.

`Concentration` The molar concentration (M) of a species.

`Reaction` A biochemical reaction. Each `Reaction` belongs to one `submodel`. It consists of a list of the species that participate in the reaction, stored as a list of references to `ReactionParticipant` instances in `participants`. A reaction that's simulated by a dynamic algorithm, such as an ODE system or SSA, must have a forward rate law. A Boolean indicates whether the reaction is thermodynamically `reversible`. If `reversible` is `True`, then the reaction must also have a backward rate law. Rate laws are stored in the `rate_laws` list, and their directions are drawn from the attributes of `RateLawDirection`.

`ReactionParticipant` `ReactionParticipant` combines a `Species` and its stoichiometric reaction coefficient. Coefficients are negative for reactants and positive for products.

`RateLaw` A rate law contains a textual `equation` which stores the mathematical expression of the rate law. It contains the `direction` of the rate law, encoded with a `RateLawDirection` attribute. `k_cat` and `k_m` attributes for a Michaelis–Menten kinetics model are provided, but their use isn't required.

`RateLawEquation` A rate law equation's `expression` contains a textual, mathematical expression of the rate law. A rate law can be used by more than one `Reaction`. The expression will be transcoded into a valid Python expression, stored in the `transcoded` attribute, and evaluated as a Python expression by a simulator. This evaluation must produce a number.

The expression is constructed from species names, compartment names, stoichiometric reaction coefficients, k_cat and k_m, and Python functions and mathematical operators. `SpeciesType` and `Compartment` names must be valid Python identifiers, and the entire expression must be a valid Python expression. A species composed of a `SpeciesType` named `species_x` located in a `Compartment` named `c` is written `species_x[c]`. When a rate law equation is evaluated during the simulation of a model the expression `species_x[c]` is interpreted as the current concentration of `species_x` in compartment `c`.

`Parameter` A `Parameter` holds an arbitrary floating point `value`. It is named, associated with a a set of `submodels`, and should include a modifier indicating the value's `units`.

`wc_lang` Model Data Sources

These classes record the sources of a model's data.

`Reference` A `Reference` holds a reference to a publication that contains data used in the model.

`DatabaseReference` A `Reference` describes a biological or chemical database that provided data for the model.

4.2.3 Using `wc_lang`

The following tutorial shows several ways to use `wc_lang`, including reading a model from disk, defining a model programmatically and writing it to disk, and using these models:

1. Install the required software for the tutorial:
   - Python
   - Pip
2. Install the tutorial and the whole-cell packages that it uses:

```
git clone https://github.com/KarrLab/intro_to_wc_modeling.git
pip install --upgrade \
    ipython \
    git+https://github.com/KarrLab/wc_lang.git#egg=wc_lang \
    git+https://github.com/KarrLab/wc_utils.git#egg=wc_utils
```

3. Change to the directory for this tutorial:

```
cd intro_to_wc_modeling/intro_to_wc_modeling/wc_modeling/wc_lang_tutorial
```

4. Open an interactive python interpreter:

```
ipython
```

5. Import the `os` and `wc_lang.io` modules:

```
import os
import wc_lang.io
```

6. Read and write models in Excel and delimited files

   `wc_lang` can read and write models from specially formatted Excel workbooks in which each worksheet represents one of the model component classes above, each row represents a class instance, each column represents an instance attribute, each cell represents the value of an attribute of an instance, and string identifiers are used to indicate relationships among objects. `wc_lang` can also read and write models from a specially formatted sets of delimiter-separated files.

In addition to defining a model, files that define models should contain all of the annotation needed to understand the biological semantic meaning of the model. Ideally, this should include:

- NCBI Taxonomy ID for the taxon
- Gene Ontology (GO) annotations for each submodel
- The structure of each species: InChI for small molecules; sequences for polymers
- Where possible, ChEBI ids for each small molecule
- Where possible, ids for each gene, transcript, and protein
- Where possible, EC numbers or KEGG ids for each reaction
- Cell Component Ontology (CCO) annotations for each compartment
- Systems Biology Ontology (SBO) annotations for each parameter
- The citations which support each model decision
- PubMed id, DOI, ISBN, or URL for each citation

This example illustrates how to read a model from an Excel file:

```
model = wc_lang.io.Reader().run(model_filename)[wc_lang.Model][0]
```

(You may ignore a `UserWarning` generated by these commands.)

If a model file is invalid (for example, it defines two species types with the same id, or a concentration that refers to a species type that is not defined), this operation will raise an exception which contains a list of all of the errors in the model definition.

To name a model stored in a set of delimiter-separated files, `wc_lang` uses a filename glob pattern that matches the files in the set. The supported delimiters are *commas* in .csv files and *tabs* in .tsv files. These files use the same format as the Excel workbook format, except that each worksheet is stored as a separate file. Excel workbooks are easier to read and edit interactively, but changes to delimiter-separated files can be tracked in code version control systems such as Git.

This example illustrates how to write a model to a set of .tsv files:

```
# 'examples_dir' is a directory
model_filename_pattern = os.path.join(examples_dir, 'example_model-*.tsv')
wc_lang.io.Writer().run(model_filename_pattern, model, data_repo_
↪metadata=False)
```

The glob pattern in `model_filename_pattern` matches these files:

```
example_model-Biomass components.tsv
example_model-Biomass reactions.tsv
example_model-Compartments.tsv
example_model-Concentrations.tsv
example_model-database references.tsv
example_model-Model.tsv
example_model-Parameters.tsv
example_model-Rate laws.tsv
example_model-Reactions.tsv
example_model-References.tsv
example_model-Species types.tsv
example_model-Submodels.tsv
example_model-Taxon.tsv
```

in `examples_dir`, each of which contains a component of the model.

Continuing the previous example, this command reads this set of .tsv files into a model:

```
model_from_tsv = wc_lang.io.Reader().run(model_filename_pattern)[wc_lang.
↪Model][0]
```

csv files can be used similarly.

7. Access properties of the model

   A `wc_lang` model (an instance of `wc_lang.core.Model`) has multiple attributes:

```
model.id                # the model's unique identifier
model.name              # its human readable name
model.version           # its version number
model.taxon             # the taxon of the organism being modeled
model.submodels         # a list of the model's submodels
model.compartments      # "  "   "  the model's compartments
model.species_types     # "  "   "  its species types
model.parameters        # "  "   "  its parameters
model.references        # "  "   "  publication sources for the model␣
↪instance
model.identifiers       # "  "   "  identifiers in external namespaces␣
↪for the model instance
```

   These provide access to the parts of a `wc_lang` model that are directly referenced by a model instance.

   `wc_lang` also provides some convenience methods that get all of the elements of a specific type which are part of a model. Each of these methods returns a list of the instances of requested type.

```
model.get_compartments()
model.get_species_types()
model.get_submodels()
model.get_species()
model.get_distribution_init_concentrations()
model.get_reactions()
model.get_dfba_obj_reactions()
model.get_rate_laws()
model.get_parameters()
model.get_references()
```

   For example, `get_reactions()` returns a list of all of the reactions in a model's submodels. As illustrated below, this can be used to obtain the id of each reaction and the name of its submodel:

```
reaction_identification = []
for reaction in model.get_reactions():
    reaction_identification.append('submodel name: {}, reaction id: {}'.
↪format(
        reaction.submodel.name, reaction.id))
```

8. Programmatically build a new model and edit its model properties

   You can also use the classes and methods in `wc_lang.core` to programmatically build and edit models. While modelers typically will not create models programmatically, creating model components in this way gives you a feeling for how models are built and will .

   The following illustrates how to program a trivial model with 1 compartment, 5 species types and one reaction:

```
# create a model with one submodel and one compartment
prog_model = wc_lang.Model(id='programmatic_model', name='Programmatic␣
↪model')

submodel = wc_lang.Submodel(id='submodel_1', model=prog_model)

cytosol = wc_lang.Compartment(id='c', name='Cytosol')

# create 5 species types
atp = wc_lang.SpeciesType(id='atp', name='ATP', model=prog_model)
adp = wc_lang.SpeciesType(id='adp', name='ADP', model=prog_model)
pi = wc_lang.SpeciesType(id='pi', name='Pi', model=prog_model)
h2o = wc_lang.SpeciesType(id='h2o', name='H2O', model=prog_model)
h = wc_lang.SpeciesType(id='h', name='H+', model=prog_model)

# create an 'ATP hydrolysis' reaction that uses these species types
atp_hydrolysis = wc_lang.Reaction(id='atp_hydrolysis', name='ATP␣
↪hydrolysis')

# add two reactants, which have negative stoichiometric coefficients
atp_hydrolysis.participants.create(
    species=wc_lang.Species(id='atp[c]', species_type=atp,␣
↪compartment=cytosol), coefficient=-1)
atp_hydrolysis.participants.create(
    species=wc_lang.Species(id='h2o[c]', species_type=h2o,␣
↪compartment=cytosol), coefficient=-1)

# add three products, with positive stoichiometric coefficients
atp_hydrolysis.participants.create(
    species=wc_lang.Species(id='adp[c]', species_type=adp,␣
↪compartment=cytosol), coefficient=1)
atp_hydrolysis.participants.create(
    species=wc_lang.Species(id='pi[c]', species_type=pi,␣
↪compartment=cytosol), coefficient=1)
atp_hydrolysis.participants.create(
    species=wc_lang.Species(id='h[c]', species_type=h,␣
↪compartment=cytosol), coefficient=1)
```

In this example `wc_lang.core.SpeciesType(id='atp', name='ATP', model=prog_model)` instantiates a `SpeciesType` instance with two string attributes and a `model` attribute that references an existing model. In addition, this expression adds the new `SpeciesType` to the model's species types, thereby showing how `obj_tables`'s underlying functionality automatically creates bi-directional references that make it easy to build and navigate `wc_lang` models, and making this assertion hold:

```
assert(atp in prog_model.get_species_types())
```

The example above illustrates another way to create and connect model components. Consider the expression:

```
atp_hydrolysis.participants.create(
    species=wc_lang.core.Species(species_type=atp, compartment=cytosol),␣
↪coefficient=-1)
```

`participants` is a Reaction instance attribute that stores a list of ReactionParticipant objects. In this expression `create` takes keyword arguments for the parameters used to instantiate a `ReactionParticipant`, instantiates a `ReactionParticipant`, and appends it to the list

in `atp_hydrolysis.participants`. These assertions hold after the 5 participants are added to the ATP hydrolysis reaction:

```
# 5 participants were added to the reaction
assert(len(atp_hydrolysis.participants) == 5)
first_reaction_participant = atp_hydrolysis.participants[0]
assert(first_reaction_participant.reactions[0] is atp_hydrolysis)
```

In general, the `create` method can be used to add model components to lists of related `wc_lang.BaseModel` objects. `create` takes keyword arguments and uses them to initialize the attributes of the component created. Thus, if `obj` has an attribute `attr` that stores a list of references to components of type `X`, this expression will create an instance of `X` and append it to the list:

```
obj.attr.create(**kwargs)
```

This simplifies model construction by avoiding creation of unnecessary identifiers for these components.

Similar code can be used to create any part of a model. All `wc_lang` objects that are subclassed from `wc_lang.BaseModel` (an alias for `obj_tables.core.Model`) can be instantiated in the normal fashion, as shown for `Model`, `Submodel`, `Compartment`, `SpeciesType` and `Reaction` above. Each subclass of `wc_lang.BaseModel` contains a `Meta` attribute that is a class which stores meta information about the subclass. The attributes that can be initialized when a `wc_lang.BaseModel` class is instantiated can be obtained from the class' `Meta` attribute, which is a dictionary that maps from attribute name to attribute instance:

```
wc_lang.Model.Meta.attributes.keys()
wc_lang.Submodel.Meta.attributes.keys()
wc_lang.SpeciesType.Meta.attributes.keys()
wc_lang.Compartment.Meta.attributes.keys()
```

For example, `Reaction` has the following attributes in `wc_lang.core.Reaction.Meta.attributes.keys()`:

```
['comments', 'id', 'max_flux', 'min_flux', 'name', 'participants',
↪'references',
    'reversible', 'submodel']
```

These attributes can also be set programmatically:

```
atp_hydrolysis.comments = 'example comments'
atp_hydrolysis.reversible = False
```

9. Viewing Models and their attributes

   All `wc_lang.BaseModel` instances can be viewed with `pprint()`, which outputs an indented representation that shows the attributes of a model, and indents and outputs connected models. To constrain the size of its output `pprint()` outputs the graph of interconnected models to a depth of `max_depth`, which defaults to 3. Model nodes at depth `max_depth+1` are represented by `<class name>: ...`, while deeper models are not traversed. And models re-encountered by `pprint()` are elided by `<attribute name>: --`. For example, after creating the reaction `atp_hydrolysis` above this expression

   ```
   atp_hydrolysis.participants[0].pprint(max_depth=1)
   ```

   creates this output:

```
ReactionParticipant:
   species:
      Species:
         species_type:
            SpeciesType: ...
         compartment:
            Compartment: ...
         concentration: None
         rate_law_equations:
         reaction_participants:
   coefficient: -1
   reactions:
      Reaction:
         id: atp_hydrolysis
         name: ATP hydrolysis
         submodel: None
         participants:
            ReactionParticipant: ...
            ReactionParticipant: ...
            ReactionParticipant: ...
            ReactionParticipant: ...
         reversible: False
         min_flux: nan
         max_flux: nan
         comments: example comments
         references:
         database_references:
         objective_functions:
         rate_laws:
```

This shows that the first `ReactionParticipant` in `atp_hydrolysis` has the attributes species, coefficient, and reactions, that the coefficient is -1, and that reactions is a list with one element which is the `atp_hydrolysis` reaction itself.

10. Validating a programmatically generated Model

    The `wc_lang.core.Model.validate` method determines whether a model is valid. If the model is invalid validate return a list of all of the model's errors. It performs the following checks:

    - Check that only one model and taxon are defined
    - Check that each submodel, compartment, species type, reaction, and reference is defined only once
    - Check that each the species type and compartment referenced in each concentration and reaction exist
    - Check that values of the correct types are provided for each attribute
      - `wc_lang.core.Compartment.initial_volume`: float
      - `wc_lang.core.Concentration.value`: float
      - `wc_lang.core.Parameter.value`: float
      - `wc_lang.core.RateLaw.k_cat`: float
      - `wc_lang.core.RateLaw.k_m`: float
      - `wc_lang.core.Reaction.reversible`: bool
      - `wc_lang.core.ReactionParticipant.coefficient`: float

- `wc_lang.core.Reference.year`: integer
- `wc_lang.core.SpeciesType.charge`: integer
- `wc_lang.core.SpeciesType.molecular_weight`: float

- Check that valid values are provided for each enumerated attribute
  - `wc_lang.core.RateLaw.direction`
  - `wc_lang.core.Reference.type`
  - `wc_lang.core.SpeciesType.type`
  - `wc_lang.core.Submodel.algorithm`
  - `wc_lang.core.Taxon.rank`

This example illustrates how to validate `prog_model`:

```
prog_model.validate()
```

11. Compare and difference Models

    `wc_lang` provides methods that determine if two models are semantically equal and report any semantic differences between two models. The `is_equal` method determines if two models are semantically equal (the two models recursively have the same attribute values, ignoring the order of the attributes which has no semantic meaning). The following code compares the semantic equality of `model` and `model_from_tsv`. Since `model_from_tsv` was generated by writing `model` to tsv files, `is_equal` should return `True`:

    ```
    assert(model.is_equal(model_from_tsv) == True)
    ```

    The `difference` method produces a textual description of the differences between two models. The following code excerpt prints the differences between `model` and `model_from_tsv`. Since they are equal, the differences should be the empty string:

    ```
    assert(model.difference(model_from_tsv) == '')
    ```

12. Normalize `model` into a reproducible order to facilitate reproducible numerical simulations

    The attribute order has no semantic meaning in `wc_lang`. However, numerical simulation results derived from models described in `wc_lang` can be sensitive to the attribute order. To facilitate reproducible simulation results, `wc_lang` provides a `normalize` to sort models into a reproducible order.

    The following code excerpt will normalize `model` into a reproducible order:

    ```
    model.normalize()
    ```

13. Please see http://code.karrlab.org for documentation of the entire `wc_lang` API.

4.3 Using *wc_env_manager* build, version, and sharing computing environments for WC modeling

WC modeling requires complex computing environments with numerous dependencies. *wc_env_manager* helps modelers and software developers setup customizable computing environments for developing, testing, and running whole-cell (WC) models and WC modeling software. This makes it easy for modelers and software developers to install and configure the numerous dependencies required for WC modeling. This helps modelers and software developers focus

on developing WC models and software tools, rather than on installing, configuring, and maintaining complicated dependencies.

In addition, *wc_env_manager* facilitates collaboration by helping WC modelers and software developers share a common base computing environment with third party dependencies. Furthermore, *wc_env_manager* helps software developers anticipate and debug issues in deployment by enabling developers to replicate similar environments to those used to test and deploy WC models and tools in systems such as Amazon EC2, CircleCI, and Heroku.

wc_env_manager uses Docker to setup a customizable computing environment that contains all of the software packages needed to run WC models and WC modeling software. This includes

- Required non-Python packages
- Required Python packages from PyPI and other sources
- WC models and WC modeling tools
- Optionally, local packages on the user's machine such as clones of these WC models and WC modeling tools

wc_env_manager supports both the development and deployment of WC models and WC modeling tools:

- **Development:** *wc_env_manager* can run WC models and WC modeling software that is located on the user's machine. This is useful for testing WC models and WC modeling software before committing it to GitHub.
- **Deployment:** *wc_env_manager* can run WC models and WC modeling software from external sources such as GitHub.

4.3.1 How *wc_env_manager* works

wc_env_manager is based on Docker images and containers which enable virtual environments within all major operating systems including Linux, Mac OSX, and Windows, and the DockerHub repository for versioning and sharing virtual environments.

1. *wc_env_manager* creates a Docker image, *wc_env_dependencies* with the third-party dependencies needed for WC modeling or pulls this image from DockerHub. This image represents an Ubuntu Linux machine.
2. *wc_env_manager* uses this Docker image to create another Docker image, *wc_env* with the WC models, WC modeling tools, and the configuration files and authentication keys needed for WC modeling.
3. *wc_env_manager* uses this image to create a Docker container to run WC models and WC modeling tools. Optionally, the container can have volumes mounted from the host to run code on the host inside the Docker container, which is helpful for using the container to test and debug WC models and tools.

The images and containers created by *wc_env_manager* can be customized using a configuration file.

4.3.2 Installing *wc_env_manager*

First, install the following requirements. See Section 6 for detailed instructions.

- git
- Docker
- Pip >= 10.0.1
- Python >= 3.5

Second, run the following command to install the latest version of *wc_env_manager* from GitHub:

```
pip install git+https://github.com/KarrLab/wc_env_manager.git#egg=wc_env_manager
```

4.3.3 Using *wc_env_manager* to build and share images for WC modeling

Administrators should follow these steps to build and disseminate the *wc_env* and *wc_env_dependencies* images.

1. Create contexts for building the *wc_env* and *wc_env_dependencies* Docker images.
2. Create Dockerfile templates for the *wc_env* and *wc_env_dependencies* Docker images.
3. Set the configuration for *wc_env_manager*.
4. Use *wc_env_manager* to build the *wc_env* and *wc_env_dependencies* Docker images.
5. Use *wc_env_manager* to push the *wc_env* and *wc_env_dependencies* Docker images to DockerHub.

Creating contexts for building the *wc_env* and *wc_env_dependencies* images

First, create contexts for building the images. This can include licenses and installers for proprietary software packages.

1. Prepare CPLEX installation
    a. Download CPLEX installer from https://ibm.onthehub.com
    b. Save the installer to the base image context
    c. Set the execution bit for the installer by running *chmod ugo+x /path/to/installer*
2. Prepare Gurobi installation
    a. Create license at http://www.gurobi.com/downloads/licenses/license-center
    b. Copy the license to the *gurobi_license* build argument for the base image in the *wc_env_manager* configuration
3. Prepare Mosek installation
    a. Request an academic license at https://license.mosek.com/academic/
    b. Receive a license by email
    c. Save the license to the context for the base image as *mosek.lic*
4. Prepare XPRESS installation
    a. Install the XPRESS license server on another machine
        i. Download XPRESS from https://clientarea.xpress.fico.com
        ii. Use the *xphostid* utility to get your host id
        iii. Use the host id to create a floating license at https://app.xpress.fico.com
        iv. Save the license file to the context for the base image as *xpauth.xpr*
        v. Run the installation program and follow the onscreen instructions
    b. Copy the IP address or hostname of the license server to the *xpress_license_server* build argument for the base image in the *wc_env_manager* configuration.
    c. Save the license file to the context for the base image as *xpauth.xpr*.
    d. Edit the server property in the first line of *xpauth.xpr* in the context for the base image. Set the property to the IP address or hostname of the license server.

Creating Dockerfile templates for *wc_env* and *wc_env_dependencies*

Second, create templates for the Dockerfiles to be rendered by Jinja, and save the Dockerfiles within the contexts for the images. The default templates illustrate how to create the Dockerfile templates.

- */path/to/wc_env_manager/wc_env_manager/assets/base-image/Dockerfile.template*
- */path/to/wc_env_manager/wc_env_manager/assets/image/Dockerfile.template*

Setting the configuration for *wc_env_manager*

Third, Set the configuration for *wc_env_manager* by creating a configuration file *./wc_env_manager.cfg* following the schema outlined in */path/to/wc_env_manager/wc_env_manager/config/core.schema.cfg* and the defaults in */path/to/wc_env_manager/wc_env_manager/config/core.default.cfg*.

- Set the repository and tags for *wc_env* and *wc_env_dependencies*.
- Set the paths for the Dockerfile templates.
- Set the contexts for building the Docker images and the files that should be copied into the images.
- Set the build arguments for building the Docker images. This can include licenses for proprietary software packages.
- Set the WC modeling packages that should be installed into *wc_env*.
- Set your DockerHub username and password.

Building the *wc_env* and *wc_env_dependencies* Docker images

Use the following command to build the *wc_env* and *wc_env_dependencies* images:

```
wc-env-manager build
```

Pushing the *wc_env* and *wc_env_dependencies* Docker images to DockerHub

Use the following command to push the *wc_env* and *wc_env_dependencies* images to GitHub:

```
wc-env-manager push
```

4.3.4 Using *wc_env_manager* to create and run Docker containers for WC modeling

Developers should follow these steps to build and use WC modeling computing environments (Docker images and containers) to test, debug, and run WC models and WC modeling tools.

1. Use *wc_env_manager* to pull existing WC modeling Docker images
2. Use *wc_env_manager* to create Docker containers with volumes mounted from the host and installations of software packages contained on the house
3. Run models and tools inside the Docker containers created by *wc_env_manager*

Pulling existing Docker images

First, use the following command to pull existing WC modeling Docker images. This will pull both the base image with third part dependencies, *wc_env_dependencies*, and the image with WC models and modeling tools, *wc_env*.:

```
wc-env-manager pull
```

The following commands can also be used to pull the individual images.:

```
wc-env-manager base-image pull
wc-env-manager image pull
```

Building containers for WC modeling

Second, set the configuration for the containers created by *wc_env_manager* by creating a configuration file *./wc_env_manager.cfg* following the schema outlined in */path/to/wc_env_manager/wc_env_manager/config/core.schema.cfg* and the defaults in */path/to/wc_env_manager/wc_env_manager/config/core.default.cfg*.

- Set the host paths that should be mounted into the containers. This should include the root directory of your clones of WC models and WC modeling tools (e.g. map host:~/Documents to container:/root/Documents-Host).
- Set the WC modeling packages that should be installed into *wc_env*. This should be specified in the pip requirements.txt format and should be specified in terms of paths within the container. The following example illustrates how install clones of *wc_lang* and *wc_utils* mounted from the host into the container.:

  ```
  /root/Documents-Host/wc_lang
  /root/Documents-Host/wc_utils
  ```

Third, use the following command to use *wc_env* to construct a Docker container.:

```
wc-env-manager container build
```

This will print out the id of the created container.

Using containers to run WC models and WC modeling tools

Fourth, use the following command to log in the container.:

```
cd /path/to/wc_env_manager
docker-compose up -d
docker-compose exec wc_env bash
```

Fifth, use the integrated WC modeling command line program, *wc_cli*, to run WC models and WC modeling tools. For example, the following command illustrates how to get help for the *wc_cli* program. See the *wc_cli* documentation for more information.:

```
container >> wc-cli --help
```

4.3.5 Using WC modeling computing environments with an external IDE such as PyCharm

The Docker images created with *wc_env_manager* can be used with external integrated development environments (IDEs) such as PyCharm. See the links below for instructions on how to use these tools with Docker images created

with *wc_env_manager*.

- Jupyter Notebook
- PyCharm Professional Edition
- Other IDEs:
  1. Install the IDE in a Docker image
  2. Use X11 forwarding to render graphical output from a Docker container to your host. See Using GUI's with Docker for more information.

4.3.6 Caveats and troubleshooting

- Code run in containers created by *wc_env_manager* can create host files and overwrite existing host files. This is because *wc_env_manager* mounts host directories into containers.
- Containers created by *wc_env_manager* can be used to run code located on your host machine. However, using different versions of Python between your host and the Docker containers can create Python caches and compiled Python files that are incompatible between your host and the Docker containers. Before switching between running code on your host your and the Docker containers, you may need to remove all `__pycache__` subdirectories and `*.pyc` files from host packages mounted into the containers.
- Code run in Docker containers will not have access to the absolute paths of your host and vice-versa. Consequently, arguments that represent absolute host paths or which contain absolute host paths must be mapped from absolute host paths to the equivalent container path. Similarly, outputs which represent or contain absolute container paths must be mapped to the equivalent host paths.
- Running code in containers created with *wc_env_manager* will be slower than running the same code on your host. This is because *wc_env_manager* is based on Docker containers, which add an additional layer of abstraction between your code and your processor.

CHAPTER 5

Appendix: Funding WC modeling research with grants and fellowships

5.1 Graduate fellowships

Below is a list of the largest graduate fellowships, their application deadlines.

- DOE Computational Science Graduate Fellowship (CSGF): Mid January
- DOD National Defense Science and Engineering (NDSEG) Fellowship: Early December
- NIH National Research Service Award (NRSA, F31): Early April, August, and December
- NSF Graduate Research Fellowship Program (GRFP): Early November

See http://www.karrlab.org/resources/grad-fellowships for advice on how to write successful applications and examples of successful applications.

5.2 Postdoctoral fellowships

Below is a list of the largest postdoctoral fellowships are their application deadlines.

- Damon Runyon Cancer Research Foundation: Mid August
- EMBO: Early August
- Helen Hay Whitney Foundation: June 30
- Human Frontiers Science Program: August
- Life Science Research Foundation: October 1
- Jane Coffin Childs Memorial Fund for Medical Research: February 1
- NIH National Research Service Award (NRSA, F32): Early April, August, and December

Below are several lists of additional smaller fellowships

- Berkeley

- Harvard
- MIT

5.3 Postdoc/faculty transition awards

Postdoc/faculty transition awards are designed to provide senior postdocs with 2 years of additional funding to complete their postdoctoral training and three years of funding to start their own independent laboratory at another institution. Below are the largest transition award programs, their application deadlines, and links to advice on how to write successful applications.

- Burroughs Welcome Fund (BWF) Career Award at the Scientific Interface (CASI)
- NIH Pathway to Independence Award (K99/R00)
  - 10 Things I Wish I Knew Before I Wrote My K99
  - Guide to the NIH Pathway to Independence Program (K99/R00)
  - How to write a K99
  - How to Write a Successful K99
  - How to write a K99/R00
  - Duke: NIH Pathway to Independence Award (K99/R00): Perspectives from Awardees
  - Stanford: How to Write a Successful NIH Career Development Award (K Award)
  - UCLA: K99 / R00 Award – Pathway to Independence

5.4 Grants

5.4.1 Funding streams for your lab

- NIH: National Institutes of Health
  - Biomedical science
  - 27 institutes and centers with specific foci
  - Investigator and sponsor-driven programs
- NSF: National Science Foundation
  - Basic science
  - Primarily investigator-driven programs
- Other governmental agencies
  - DOD: Department of Defense
  - DARPA: Defense Advanced Research Projects Agency
  - DOE: Department of Energy
- Non-profit organizations, e.g. Allen Foundation, Chan-Zuckerberg Initiative, Howard Hughes Medical Institute
  - Catalyze initial research rather than provide sustained funding
  - Often more risk- tolerant

- Your institution
  - Startup funds
  - Small seed grants
- Institutional training grants for graduate students and postdoctoral scholars
  - Very simple applications
  - Don't consume PI percent effort
- Graduate, postdoctoral fellowships
  - Relatively easy to win
  - Don't consume PI percent effort
- Industry contracts

5.4.2 Taxonomy of funding opportunities

- Scientific area
- Project vs. program
  - Project grants (e.g. NIH R01, R21) provide funds for a specific project with specific goals and a specific approach
  - Program grants (e.g. NIGMS MIRA) provide funds for an area of research and don't require specific goals or a specific approach. These grants provide more flexible funding to enable researchers to follow evolving scientific priorities.
- Investigator-driven vs. sponsor-driven
  - Investigator-driven programs enable researchers to propose projects within a broad scientific area. Often, the programs have few priorities and few restrictions and the sponsor aims to support the most impactful and innovative proposals
  - Sponsor-driven programs provide funding for strategically important research as determined by the sponsor. Generally, sponsor-driven progams have specific priorities and numerous restrictions.
- Scientific stage/risk
  - High-risk/high-reward (e.g. NIH Director's program, NSF RAISE program): significant funding for early ideas with substantial potential impact. These programs require little preliminary data, but a strong PI track record of impact is essential. Although high-risk/high-reward grants provide significant funding, NSF and NIH award few of these grants.
  - Early-stage (e.g. NIH R21): funding modest funding for early ideas with minimal prior evidence
  - Hardening
- Project size
  - New project (e.g. NIH R21): 1-2 people over 1-3 years
  - Established project (e.g. NIH R01): 2-3 people over 3-5 years
  - Collaborative project (e.g. NIH center): requires a larger number of people over 5+ years
- Investigator stage
  - Postdoc (e.g. NIH F32, NIH K99/R00)
  - Early career (e.g. NSF CAREER, NIGMS MIRA, NIH EIA, NIH New Innovator)

  - Mid-late career (e.g. NIGMS MIRA)

5.4.3 Funding programs for early career investigators

Below is a list of some of the biggest funding opportunities specifically for early career investigators

- NIGMS Maximizing Investigators' Research Award (MIRA, R35)
  - Funds research programs
  - $250,000 per year
  - 51% effort required
  - 6 page application that emphasizes the applicant, the major challenges in their field, and their proposed research program
- Director's Program New Innovator Award (DP2)
  - Funds high-risk, high-reward research
  - No preliminary data required
  - $300,000 per year
  - 25% effort required
  - 12 page application that emphasizes the applicant, significance, and innovation
  - Applicant must have NI and ESI status
- NSF Faculty Early Career Development Program
- DOE Early Career Research Program
- DOD Young Faculty Award
- Non-profit foundations
  - Beckman Young Investigators Program
  - Pew Scholars
  - Searle Scholars Program
  - Sloan Research Fellowships

See http://www.spo.berkeley.edu/fund/newfaculty.html for a list of additional smaller funding opportunities for new faculty.

In addition, many NIH institutes have lower funding thresholds for *New Investigators* (NI; applicants which have not yet received an R01 or equivalent) and *Early Stage Invesigators* (ESI; applicants within 10 years of the completion of their PhD or medical residency). Keep in mind that winning a grant as a Co-PI, would also terminate your NI status. For this reason, until you receive your first grant as the primary PI, it could be a good strategy not to submit proposals as a Co-PI and instead submit those proposals as a Co-Investigator.

5.4.4 Finding funding opportunities

Below are several resources for finding funding opportunities

- Funding agency websites
- NIH weekly email funding guide
- NSF email updates

- DOD, DARPA updates
- SPIN database of sponsored projects

5.4.5 Eligibility

Universities often only allow personnel with Principal Investigator (PI) status to submit proposals. This status is often only given to tenure track faculty. Some funding programs have additional requirements such as a minimum effort or a maximum time from start of the PI's first tenure track faculty position.

5.4.6 Deadlines

- NIH: generally, every 4 months
- NSF: continous submission
- Other: variable, see program announcements

5.4.7 Proposal process

1. Contact program staff: This can often be accomplished via email. In some cases, a letter of intent or pre-proposal is required.
2. Discuss idea with program staff: identify suitable funding programs and get feedback.
3. Submit proposal
4. Program staff assign proposal to panel
5. Peers make recommendations to program staff
6. Discuss and rebut concerns with program staff
7. Program staff make funding decisions
8. Resubmit proposal
    - NIH: Has formal system for resubmission, but you we will likely be assigned different reviewers
    - NSF: No formal resubmission system, but similar proposals can be submitted

5.4.8 Writing proposals

- Summaries
    - Specific Aims: 1 page for reviewers
    - Summary: 30 lines for public
    - Narrative: 6 lines for public
- Introduction to resubmission (NIH): 1 page
- Project description: 12 (NIH) – 15 (NSF) pages, 0.5" margins, 11pt
    - Problem statement
    - Anticipated impact and innovation
    - Background

  - Your prior work
  - Results of prior support (NSF)
  - Research plan: 2-5 aims
  - Education and outreach plan (NIH centers and NSF)
  - Timeline
  - Management plan: who will do what
  - Evaluation plan: how you will assess progress and success
  - Future directions
- Bibliography: unlimited
- Resource sharing plans: 1 page
  - Outline the products and how they will be tested, documented (examples, tutorials, API docs), and disseminated
  - Outline the timeline for dissemination
  - Model organism sharing plan: N/A
  - Genomic data sharing plan: N/A
  - Data sharing plan
  - Software sharing plan
- Mentoring plan (NSF): 1 page
- Budget justification: 1-2 pages
  - Brief description of the requested funds and why they are needed
- Biosketches of key personnel: 5 pages each
  - Personal statement
  - 3 major scientific contributions, each with 3 publications
- Letters of support from collaborators
- Major equipment: ~2 pages
  - Custom software
  - Computer cluster
- Facilities and other resources: ~2 pages
  - Scientific environment at university and in department
  - Computers and software
  - Lab and office facilities
  - Junior faculty mentorship and professional development for junior faculty
  - Central university resources such as library
  - Administrative support

5.4.9 Typical costs for budgets

- Personnel salary and fringe benefits (85-90% of total budget)
  - Student: $42,000/yr
  - Postdoc: $42-70,000/yr
  - Fringe benefits: 28.5%
- Recruiting: $1,000-1,500/visit
- Computer: $1,500
- Publication: ~$2,500
- Travel (~3% of total budget)
  - 1-2 conferences/person/yr
  - $2,500 per conference
- Materials & supplies: ~$200/yr
- Freelancers : $30/hr
  - Illustrators
  - Editors
  - Web designers
- Computer services
  - CircleCI: $600/yr
  - CodeClimate: $0
  - Coveralls: $300/yr
  - DreamHost (Web hosting and IP registration): $200/yr
  - Docker Hub: $0
  - GitHub: $600/yr
  - Google Drive: $20/yr
  - Minerva: $0
  - Read the Docs: $0
- Software
  - Adobe Creative Cloud: $240/yr
  - MATLAB: $100/yr
  - MS Office: free
- Indirect costs: 35-70%

5.4.10 Submitting proposals

Proposals must be submitted through your institutions Grants and Contracts Office using their online proposal submission system. These online systems are straightforward. They simply require you to upload each component of your proposal as a .docx or .pdf document and enter your budget using a set of webforms. Typically, proposals must be submitted internally 1-2 weeks in advance of the external deadline.

5.4.11 Peer review

1. Program officers assign each proposal to a panel with 10-20 scientists and up to ~50 proposals
2. Each proposal is assigned to 3 scientists in general area
    - Reviewers often do not have immediate expertise in the topic of the proposal
    - Each reviewer has ~10 12-15 page proposals
    - Reviewers read and score each proposal according to mutliple criteria
    - Reviewers often ignore specific program goals
3. Reviewers meet to align on a consensus score for each proposal which serves as a recommendation to the program staff
    - ~15 minutes of discussion per proposal
        1. First reviewer: Summarizes proposal and strengths and weaknesses
        2. Second reviewer: Summarizes additional strengths and weaknesses
        3. Scribe: Summarizes additional strengths and weaknesses
        4. Discussion about discrepant opinions
        5. Scribe: suggests score
        6. Reviewers agree to score
        7. Scribe writes summary of panel discussion
        8. Panel discussion is reviewed by program staff
        9. After all proposals are discussed,
            1. Broader discussion about important areas to support
            2. The panel identifies the very best proposals
    - Discussion led by panel chair (NIH) or program staff (NSF)
    - NSF: discuss all proposals
    - NIH: discuss only top-scoring proposals
4. The reviewers and program staff produce a written summary of the discussion
    - NSF: third reviewer
    - NIH: program staff
5. The program staff make the final funding decision informed by the recommendations from the reviewers. At this stage, the program staff review the written summaries. In addition, the program staff may ask for further information from applicants about concerns expressed by the reviewers.

5.4.12 Statistics (NIGMS)

- Success rate: 28% (including resubmission)
- Average R01: $237,000

5.4.13 Grant award process

1. Funding agency sends official notification to you and your university
2. Funding starts immediately
3. University sets up fund for award (immediate)
4. University adjusts effort based on proposed budget (immediate, retroactive to start date)
5. Open positions (~1 month) and begin interviewing
6. Hire staff (1-3 months, depending on visa needs)

5.4.14 Annual grant renewals

Annual progress reports

- Scientific progress and products (e.g. publications, presentations, websites)
- Dissemination efforts
- Outreach and education efforts
- List of participants and contributions
- Impact of the above
- Unanticipated challenges and revised plans
- Plans for next year

Annual non-competing renewal applications

- Budget

For many NIH program, competing renewal applications every ~5 years

- Full application

5.4.15 Advice for winning grants

- Focus on significant problems and propose innovative solutions
- Generate compelling proof-of-concept
- Publicize your proof-of-concept
- Identify topical funding mechanisms
- Thoroughly read the funding opportunity announcement
- Discuss your ideas with the program officers, especially for DARPA, DOD, and DOE
- Solicit examples of proposals that have been funded by the same program and solicit advice from previous winners. This is particularly helpful for the administrative sections of proposals.
- Dedicate significant grant writing time and allow extra time for unfamiliar opportunities
- Determine who your audience is and write for them
- Follow all of the directions in funding opportunity announcements
- Seek feedback for your colleagues and your lab

5.4.16 Advice for resubmissions

Below is our advice for submitting revised proposals

- Carefully read all of the reviewers concerns
- Keep in mind that blaming the reviewers is not productive. You can't change the reviewers or program officers, but you can change your proposal and how you present it.
- Keep in mind that reviewer concerns are often rooted in poor explanations rather than bad ideas. For this reason, reviewer concerns can often be addressed simply by clarifying the proposal.
- Synthesize and rank the reviewers' concerns
- Develop a revised plan that addresses all of the reviewers' concerns.
- Discuss your plans with the program officers
- Revise your proposal. This could require re-writing your entire proposal.

In addition to all of the content of the first submission, NIH resubmissions must include a 1-page "Introduction to resubmission". These documents should (a) summarize the reviewer's major concerns, (b) summarize your major revisions, and (c) provide a point-by-point summary of each of the reviewers' major concerns and describe how you have addressed them or why you believe they are unfounded.

To help reviewers identify the major changes to your proposal, you should mark these sections with vertical bars in the margins.

CHAPTER 6

Appendix: Installing the code in this primer and the required packages

Each tutorial outlines how to install all of the necessary software. The following is a consolidated guide to installing all of the software needed for the tutorials.

6.1 Requirements

Below is a list of all of the packages needed for the tutorials. Note, each tutorial only requires a subset of these packages. Please see the tutorials for information about the packages required for each tutorial.

- ChemAxon Marvin
- CPLEX
- Docker
- Gimp
- Git
- Illustrator
- Inkscape
- Meld
- Open Babel
- Pandoc
- Pip
- Python
- SUNDIALS, scikits.odes

In addition, the following packages are optional

- Cbc, CyLP

- Gurobi
- MINOS, solveME
- MOSEK
- SoPlex, soplex_cython
- XPRESS

6.2 How to install these tutorials

Run the following command to install the latest version from GitHub:

```
pip install https://github.com/KarrLab/wc_utils.git#egg=wc_utils
pip install https://github.com/KarrLab/obj_tables.git#egg=obj_tables
pip install https://github.com/KarrLab/wc_lang.git#egg=wc_lang
pip install https://github.com/KarrLab/intro_to_wc_modeling.git#egg=intro_to_wc_
↪modeling
```

6.3 Detailed instructions to install the tutorials and all of the requirements

1. Follow the instructions in *How to build a Ubuntu Linux image with Docker* to build a Linux virtual machine
2. Install several packages
    1. Install SSH:

    ```
    sudo apt-get install ssh
    ```

    2. Install Git:

    ```
    sudo apt-get install \
        git \
        libgnome-keyring-dev \
        meld
    ```

    3. Configure your Git user name and email:

    ```
    git config --global user.name "John Doe"
    git config --global user.email "johndoe@example.com"
    ```

    4. Configure Git to store your GitHub password:

    ```
    cd /usr/share/doc/git/contrib/credential/gnome-keyring
    sudo make
    git config --global credential.helper /usr/share/doc/git/contrib/credential/
    ↪gnome-keyring/git-credential-gnome-keyring
    ```

    5. Add the following to *~/.gitconfig* to configure Git to use meld to visualize differences:

```
[diff]
    tool = meld
[difftool]
    prompt = false
```

6. Install Docker:

```
# Docker
sudo apt-get install curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo usermod -aG docker $USER

# Docker Compose
sudo curl -L "https://github.com/docker/compose/releases/download/1.25.0/
↪docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

7. Install Python 3.7.5

   export python_version=3.7.5

   sudo apt-get install build-essential ca-certificates libbz2-dev libexpat1 libexpat1-dev libffi-dev libffi6 libreadline-dev libsqlite3-dev libssl-dev tar tk-dev wget zlib1g-dev liblzma-dev libtinfo-dev mime-support

   cd /tmp wget https://www.python.org/ftp/python/${python_version}/Python-${python_version}.tgz -O /tmp/Python-${python_version}.tgz tar xzf /tmp/Python-${python_version}.tgz cd /tmp/Python-${python_version} ./configure

   –prefix=/usr/local –enable-optimizations –enable-shared –enable-unicode=ucs4 –with-system-expat –with-system-ffi

   make sudo make install sudo ldconfig sudo pip3.7 install

   ipython pytest coverage

8. Install Open Babel 2.4.1:

```
export obabel_version_dash=2-4-1
export obabel_version_dot=2.4.1

sudo apt-get install \
    build-essential \
    cmake \
    libcairo2-dev \
    libeigen3-dev \
    libxml2-dev \
    tar \
    wget \
    zlib1g-dev

cd /tmp
wget https://github.com/openbabel/openbabel/archive/openbabel-${obabel_
↪version_dash}.tar.gz -O /tmp/openbabel-${obabel_version_dot}.tar.gz
```

(continues on next page)

(continued from previous page)

```
tar -xvvf /tmp/openbabel-${obabel_version_dot}.tar.gz
cd /tmp/openbabel-openbabel-${obabel_version_dash}
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig

sudo pip3.7 install openbabel
```

9. Install ChemAxon Marvin

    1. Install Java:

    ```
    sudo apt-get install openjdk-11-jdk openjdk-11-jre
    ```

    2. Download the installer from https://chemaxon.com/products/marvin/download

    3. Install ChemAxon Marvin:

    ```
    export version_marvin=19.25
    sudo dpkg -i ~/Downloads/marvin_linux_${version_marvin}.deb
    ```

    # Add Marvin to the Java class path:

    ```
    echo "export JAVA_HOME=/usr/lib/jvm/default-java" >> ~/.bash2rc
    echo "export CLASSPATH=\$CLASSPATH:/opt/chemaxon/marvinsuite/lib/
    ↪MarvinBeans.jar" >> ~/.bash2rc
    ```

    1. Obtain a license at https://docs.chemaxon.com/display/docs/About+ChemAxon+Licensing. Free 2-year licenses are available for academic research.

    2. Download your license from https://accounts.chemaxon.com/my/licenses

    3. Save your your license to `~/.chemaxon/license.cxl`

10. Install CPLEX 12.10 and the CPLEX Python binding

    1. Register for an academic account and download CPLEX from https://www.ibm.com/academic

    2. Install CPLEX:

    ```
    chmod ugo+x ~/Downloads/cplex_studio1210.linux-x86-64.bin
    sudo ~/Downloads/cplex_studio1210.linux-x86-64.bin
    ```

    3. Install the binding for Python 3.7:

    ```
    sudo python3.7 /opt/ibm/ILOG/CPLEX_Studio1210/python/setup.py install
    ```

11. Optionally, install the COIN-OR Cbc optimization package and the CyLP Python binding:

```
# set environment variables
echo "" >> ~/.bashrc
echo "# COIN-OR: CoinUtils, Cbc" >> ~/.bashrc
echo "export COIN_INSTALL_DIR=/opt/coin-or/cbc" >> ~/.bashrc
echo "export PATH=\"\${PATH}:/opt/coin-or/cbc/bin:/opt/coin-or/coinutils/bin\"
↪" >> ~/.bashrc
echo "export LD_LIBRARY_PATH=\"\${LD_LIBRARY_PATH}:/opt/coin-or/cbc/lib:/opt/
↪coin-or/coinutils/lib\"" >> ~/.bashrc
```

(continues on next page)

(continued from previous page)

```
~/.bashrc
ldconfig

# install utilities
sudo apt-get install wget

# CoinUtils
cd /tmp
wget --no-check-certificate https://www.coin-or.org/download/source/CoinUtils/
↪CoinUtils-2.10.14.tgz
tar -xvvf CoinUtils-2.10.14.tgz
cd CoinUtils-2.10.14
mkdir build
cd build
mkdir -p /opt/coin-or/coinutils
../configure -C --prefix=/opt/coin-or/coinutils --enable-gnu-packages
make
make install

# COIN-OR Cbc
/tmp
wget --no-check-certificate https://www.coin-or.org/download/source/Cbc/Cbc-2.
↪8.5.tgz
tar -xvvf Cbc-2.8.5.tgz
cd Cbc-2.8.5
mkdir build
cd build
../configure -C --prefix=/opt/coin-or/cbc --enable-gnu-packages
make
make install

# CyLP
pip install numpy scipy
pip install git+https://github.com/jjhelmus/CyLP.git@py3#egg=cylp
```

12. Optionally, install the Gurobi optimization package and the Gurobi Python binding

    1. Get a Gurobi license from http://www.gurobi.com. Gurobi provides free licenses for academic users.
    2. Install Gurobi:

```
sudo apt-get install wget
wget http://packages.gurobi.com/8.1/gurobi8.1.0_linux64.tar.gz
tar xvfz gurobi8.1.0_linux64.tar.gz
mv gurobi810 /opt/

echo "" >> ~/.bashrc
echo "# Gurobi" >> ~/.bashrc
echo "export GUROBI_HOME=/opt/gurobi810/linux64" >> ~/.bashrc
echo "export PATH=\"\${PATH}:\${GUROBI_HOME}/bin\"" >> ~/.bashrc
echo "export LD_LIBRARY_PATH=\"\${LD_LIBRARY_PATH}:\${GUROBI_HOME}/lib\""␣
↪>> ~/.bashrc
```

    3. Use your license to activate Gurobi:

```
/opt/gurobi810/linux64/bin/grbgetkey "<license>"
```

    4. Install the Python binding:

```
cd /opt/gurobi810/linux64
python setup.py install
```

13. Optionally, install the MINOS optimization package and the MINOS Python binding:
    1. Request an academic license from Michael Saunders
    2. Use the following commands to compile MINOS:

```
apt-get install csh gfortran zip
cd /path/to/parent of quadLP.zip
unzip quadLP.zip

cd quadLP/minos56
sed -i 's/FC        = gfortran/FC        = gfortran -fPIC/g' Makefile.defs
make clean
make
cd /tmp/quadLP/minos56/test
make minos
./run minos t1diet

../../../quadLP/qminos56
sed -i 's/FC        = gfortran/FC        = gfortran -fPIC/g' Makefile.defs
make clean
make
cd /tmp/quadLP/qminos56/test
make minos
./run minos t1diet
```

    3. Use the following commands to install the MINOS Python binding:

```
git clone https://github.com/SBRG/solvemepy.git
cd solvemepy
cp /path/to/quadLP/minos56/lib/libminos.a ./
cp /path/to/quadLP/qminos56/lib/libquadminos.a ./
pip install .
```

14. Optionally, install the MOSEK optimization package and the Mosek Python binding:
    1. Request an academic license at https://license.mosek.com/academic
    2. Recieve a license by email
    3. Save the license to *${HOME}/mosek/mosek.lic*
    4. Install Mosek:

```
sudo apt-get install wget
cd /tmp
wget --no-check-certificate https://d2i6rjz61faulo.cloudfront.net/
↪stable/8.1.0.78/mosektoolslinux64x86.tar.bz2
tar -xvvf mosektoolslinux64x86.tar.bz2
mv /tmp/mosek /opt/

echo "" >> ~/.bashrc
echo "# Mosek" >> ~/.bashrc
echo "export PATH=\"\${PATH}:/opt/mosek/8/tools/platform/linux64x86/
↪bin\"" >> ~/.bashrc
echo "export LD_LIBRARY_PATH=\"\${LD_LIBRARY_PATH}:/opt/mosek/8/
↪tools/platform/linux64x86/bin\"" >> ~/.bashrc
```

5. Install the Python binding:

```
# Python 3.6
cd /opt/mosek/8/tools/platform/linux64x86/python/3/
python3.6 setup.py install
```

15. Optionally, install the SoPlex optimization package and the SoPlex Python binding:
    1. Download SoPlex 3.1.1 from http://soplex.zib.de/#download
    2. Use the following commands to install SoPlex:

```
cd /path/to/parent of soplex-3.1.1.tgz
tar -xvvf soplex-3.1.1.tgz
cd soplex-3.1.1
mkdir build
cd build
cmake ..
make
make test
make install
```

    3. Use the following commands to install the SoPlex Python binding:

```
apt-get install libgmp-dev
pip install cython
git clone https://github.com/SBRG/soplex_cython.git
cd soplex_cython
cp /path/to/soplex-3.1.1.tgz .
pip install .
```

16. Optionally, install the XPRESS optimization package and the XPRESS Python binding
    1. Download and unpack XPRESS:

```
cd /tmp
wget --no-check-certificate https://clientarea.xpress.fico.com/
↪downloads/8.5.6/xp8.5.6_linux_x86_64_setup.tar
mkdir xp8.5.6_linux_x86_64_setup
tar -xvvf xp8.5.6_linux_x86_64_setup.tar -C xp8.5.6_linux_x86_64_
↪setup
```

    2. Get your host id:

```
cd /tmp/xp8.5.6_linux_x86_64_setup
utils/xphostid | grep -m 1 "<id>" | cut -d ">" -f 2 | cut -d "<" -f␣
↪1
```

    3. Use your host id to create a license at https://app.xpress.fico.com
    4. Save the license to */tmp/xpauth.xpr*
    5. Install XPRESS. Note, the standard library directory needs to be added to the library path to prevent the OS from using the versions of libcrypto and libssl provided by XPRESS.:

```
cd /tmp/xp8.5.6_linux_x86_64_setup
./install.sh

echo "" >> ~/.bashrc
```

(continues on next page)

(continued from previous page)

```
echo "# XPRESS" >> ~/.bashrc
echo "export XPRESSDIR=/opt/xpressmp" >> ~/.bashrc
echo "export PATH=\"\${PATH}:\${XPRESSDIR}/bin\"" >> ~/.bashrc
echo "export LD_LIBRARY_PATH=\"\${LD_LIBRARY_PATH}:/lib/x86_64-
↪linux-gnu:\${XPRESSDIR}/lib\"" >> ~/.bashrc
echo "export CLASSPATH=\"\${CLASSPATH}:\${XPRESSDIR}/lib/xprs.jar:\$
↪{XPRESSDIR}/lib/xprb.jar:\${XPRESSDIR}/lib/xprm.jar\"" >> ~/.
↪bashrc
echo "export XPRESS=\"\${XPRESSDIR}/bin\"" >> ~/.bashrc
```

6. Setup the XPRESS Python binding:

   - Add XPRESS to your Python path:

   ```
   # Python 3.6
   echo "/opt/xpressmp/lib" | tee /usr/local/lib/python3.6/site-
   ↪packages/xpress.pth
   ```

   - Save the following package meta data to */usr/local/lib/python3.6/site-packages/xpress-8.5.6.egg-info* for Python 3.6:

   ```
   Metadata-Version: 1.0
   Name: xpress
   Version: UNKNOWN
   Summary: FICO Xpress-Optimizer Python interface
   Home-page: http://www.fico.com/en/products/fico-xpress-
   ↪optimization
   Author: Fair Isaac Corporation
   Author-email: UNKNOWN
   License: UNKNOWN
   Description:
       Xpress-Python interface
       Copyright (C) Fair Isaac 2016
       Create, modify, and solve optimization problems in Python␣
   ↪using the Xpress Optimization suit
   Platform: UNKNOWN
   ```

   Note: If you want to install XPRESS onto a cluster, virtual machine, or docker image, you should first install a XPRESS license server on a static host and then install XPRESS using a floating license. See the XPRESS documentation for more information.

17. Install the SUNDIALS ODE solver and the scikits.odes Python interface:

    1. Install the Fortran and BLAS:

    ```
    sudo apt-get install \
        build-essential \
        cmake \
        gfortran \
        libopenblas-base \
        libopenblas-dev \
        wget
    ```

    2. Download, compile, and install SUNDIALS 3.2.1:

    ```
    export sundials_version=3.2.1
    cd /tmp
    ```

(continues on next page)

(continued from previous page)

```
wget https://computation.llnl.gov/projects/sundials/download/sundials-$
↪{sundials_version}.tar.gz
tar xzf sundials-${sundials_version}.tar.gz
cd sundials-${sundials_version}
mkdir build
cd build
cmake \
    -DEXAMPLES_ENABLE=OFF \
    -DLAPACK_ENABLE=ON \
    -DSUNDIALS_INDEX_TYPE=int32_t \
    ..
make
sudo make install
```

3. Install scikits.odes:

```
sudo pip install "scikits.odes < 2.5"
```

4. Remove SUNDIALS source files:

```
cd /tmp
rm sundials-${sundials_version}.tar.gz
rm -r sundials-${sundials_version}
```

18. Install the Sublime text editor:

```
wget -qO - https://download.sublimetext.com/sublimehq-pub.gpg | sudo apt-key␣
↪add -
echo "deb https://download.sublimetext.com/ apt/stable/" | sudo tee /etc/apt/
↪sources.list.d/sublime-text.list
sudo apt-get update
sudo apt-get install sublime-text
```

19. Install the PyCharm IDE:

```
sudo mv ~/Downloads/pycharm-community-2019.3.tar.gz /opt/
cd /opt/
sudo tar -xzf pycharm-community-2019.3.tar.gz
sudo rm -r pycharm-community-2019.3.tar.gz

# Run PyCharm
# pycharm-community-2019.3/bin/pycharm.sh &
```

20. Install the CircleCI command line tool:

```
sudo curl -o /usr/local/bin/circleci https://circle-downloads.s3.amazonaws.
↪com/releases/build_agent_wrapper/circleci
sudo chmod +x /usr/local/bin/circleci
```

21. Purchase and install Illustrator

3. Configure the packages

1. Open Sublime and edit the following settings

- Tools >> Install Package Control
- Preferences >> Package control >> Install package >> AutoPEP8

- Preferences >> Key Bindings:

```
[
    {"keys": ["ctrl+shift+r"], "command": "auto_pep8", "args": {"preview
↪": false}}
]
```

2. Open PyCharm and set the following settings to configure PyCharm
   - File >> Settings >> Tools >> Python Integrated Tools >> Default test runner: set to py.test
   - Run >> Edit configurations >> Defaults >> Python tests >> py.test: add additional arguments "–capture=no"
   - Run >> Edit configurations >> Defaults >> Python tests >> Nosetests: add additional arguments "–nocapture"
3. Optional, setup IDEs such as PyCharm to run code using a Docker image, such as, an image created with *wc_env_manager*.
   - Jupyter Notebook
   - PyCharm Professional Edition
   - Other IDEs:
     1. Install the IDE in a Docker image
     2. Use X11 forwarding to render graphical output from a Docker container to your host. See Using GUI's with Docker for more information.
4. Install additional software for tutorials:

```
sudo apt-get install \
    gimp \
    inkscape \
    mysql-server \
    texlive
```

CHAPTER 7

Appendix: Acronyms

CI continuous integration

CPU central processing unit

DES discrete event simulation

FBA flux balance analysis

GPU graphical processing unit

hESC human embryonic stem cell

InChI IUPAC International Chemical Identifier

MIASE Minimum Information About a Simulation Experiment

ODE ordinary differential equations

PDE partial differential equations

PDES parallel discrete event simulation

PGDB pathway genome database

ROSS Rensselaer's Optimistic Simulation System

SBML Systems Biology Markup Language

SED-ML Simulation Experiment Description Markup Language

SESSL Simulation Experiment Specification via a Scala Layer

SI International System of Units

SMILES simplified molecular-input line-entry system

SSA Stochastic Simulation Algorithm

SVG Scalable Vector Graphics

VCS version control system

WC whole-cell

CHAPTER 8

Appendix: Glossary

combinatorial complexity The large number of species and interactions that can occur in biological systems due to the noisy interfaces between biomolecules. Examples of combinatorial complexity include the large number of possible phosphorylation states of each protein; the large number of possible subunit compositions of each protein complex; the large number of RNA transcripts that can result from multiple transcription start and stop sites, splicing, RNA editing, and RNA degradation. To capture the combinatorial complexity of cell biology, WC models should be represented using rules and simulating using network-free simulation.

continuous integration (CI) A method for finding errors quickly by executing unit tests each time a system is revised, typically each time a revised system is pushed to a version control system such as Git.

curse of dimensionality The phenomenon that it is challenging to model high-dimensional systems due to sparsity of high-dimensional data and the combinatorial complexity of high-dimensional systems [1].

data model A description of the types of entities and the attributes of each type of entity, including attributes which describe relationships among types of entities.

See also: schema

discrete event simulation (DES) A dynamical simulation framework in which the simulated system evolves in discrete steps that represented as event messages.

flux balance analysis (FBA) A constraint-based framework that is frequently used to predict the steady state reaction fluxes of large metabolic networks.

Gillespie's algorithm An algorithm for exactly simulating biochemical networks.

See also: Stochastic Simulation Algorithm (SSA)

identifiability The ability of the value of a parameter to be uniquely, accurately, and precisely determined from experimental observations. For example, the values of parameters that only appear in combination with other parameters cannot be estimated by calibrated model predictions to experimental observations.

See also: model calibration, parameter estimation

IUPAC International Chemical Identifier (InChI) A textual format for describing the structure of a chemical compound including its chemical formula, bond connectivity, protonation, charge, stereochemistry, and isotope composition.

See also: simplified molecular-input line-entry system (SMILES)

Minimum Information About a Simulation Experiment (MIASE) Standard for the minimum metadata that should be recorded about a simulation experiment [2].

model calibration The process of determining the values of the parameters of a model typically by numerically minimizing the distance between model predictions and experimental observations.

See also: identifiability, parameter estimation

model organism database A database that contains integrated experimental information about a single species.

See also: pathway/genome database (PGDB)

model reduction The process of reducing a model to reduced model.

See also: reduced model

multi-algorithmic simulation A simulation in which multiple submodels are simultaneously simulated using different simulation algorithms such as ODE integration, SSA, and FBA [3]. Multi-algorithmic simulations are frequently used to simultaneously simulate both well-characterized pathways with fine-grained simulation algorithms and poorly-characterized pathways with coarse-grained simulation algorithms.

network-free simulation A methodology for efficient stochastic simulation of models that are described using rules. In contrast to conventional simulation methods which enumerate the entire reaction network (each species and each reaction) prior to simulation, network-free simulations dynamically discover the reaction network during simulation as states become occupied and reactions gain non-zero propensities. Network-free simulation is particularly effective for simulating combinatorially large state spaces that are sparsely occupied by small numbers of particles.

ontology A controlled vocabulary of terms, as well as the relationships among the terms. For example, the Unit of Measurement Ontology (UO) defines standard names for several common units and their relationship to the SI units and prefixes.

parallel discrete event simulation (PDES) A parallel implementation of discrete event simulation.

parameter estimation The process of estimating the values of the parameters of a model, typically by numerically minimizing the distance between model predictions and experimental observations.

See also: identifiability, model calibration

pathway/genome database (PGDB) A model organism database that contains integrated experimental information about the molecular biology of a single species such as its genome sequence, genes, protein complexes, and metabolic reactions.

See also: model organism database

provenance Metadata about the origin of data or models such as how a model was developed including who developed the model; when the model was developed; and the data source, assumptions, and design decisions that were used to build the model.

reaction network modeling The conventional, low-level representation of biochemical models which enumerates each individual species and each individual reaction. In contrast, rule-based modeling is an abstraction for representing reaction networks in terms of species and reaction patterns that can generate all of the individual species and reactions.

reconstruction The process of determining the molecular species and reactions of a biological process.

reduced model A smaller, less complex, and/or computationally cheaper model that approximates the behavior of the original model. Reduced models can be created either by lumping species, reactions, and/or parameters to create a second smaller, mechanistic model or by fitting model predictions to a smaller data-driven model.

See also: model reduction

rule-based modeling An abstraction for representing models in terms of species and reaction patterns which describe multiple individual species and reaction instances. Rule-based modeling is particularly effective for describing models with large numbers of species and reactions that emerge from the combinatorial interactions among species. Rule-based models can be simulated using conventional methods by statistically enumerating the reaction network or using network-free simulation which dynamically discovers the reaction network during simulation.

schema A description of the types of entities and the attributes of each type of entity, including attributes which describe relationships among types of entities.

See also: data model

simplified molecular-input line-entry system (SMILES) A textual format for describing the structure of a chemical compound. However, we recommend using InChI rather than SMILES because InChI is an open standard.

See also: IUPAC International Chemical Identifier (InChI)

Stochastic Simulation Algorithm (SSA) An algorithm for exactly simulating biochemical networks.

See also: Gillespie's algorithm

surrogate model A, typically computationally cheaper, model which approximates the behavior of another model.

See also: function approximation, metamodeling, model emulation, *model reduction*, *reduced model*, response surface modeling

Systems Biology Markup Language (SBML) An extensible format for describing cell models in terms of species and reactions.

test coverage The fraction of a system which is tested by a set of unit tests. To verify that a system is implemented correctly, 100% of the system should be tested. For example, computer code should be verified by testing every line of code and ranch. Similarly, models should be verified by testing the behavior of each species, reaction, and submodel at the edge cases of each rate law.

unit testing A methodology for organizing multiple tests to verify that a system is implemented correctly. These tests typically consist of tests of individual components, groups of components, and the entire system. For example, tests of model can test individual species, reactions, and submodels, as well as groups of submodels and entire models.

validation The process of checking that a system fulfills its intended purpose. For example, models can be validated by checking that they recapitulate the true biology (i.e., independent experimental data that was not used for model construction).

verification The process of checking that a system is implemented correctly. For example, models can be verified by checking that they recapitulate the known biology (i.e., the data that was used for model construction).

version control A methodology for tracking and merging changes to one or more documents which facilitates collaboration development of large systems such as models. One of the most popular version control systems for computer code is Git.

CHAPTER 9

Appendix: References

CHAPTER 10

Appendix: About this primer

10.1 License

This primer is released under the MIT license

10.2 Authors

The primer was written by Jonathan Karr and Arthur Goldberg at the Icahn School of Medicine at Mount Sinai in New York, USA.

10.3 Acknowledgements

We thank Yin Hoon Chew, Saahith Pochiraju, Yosef Roth, Andrew Sundstrom, and Bal'azs Szigeti for valuable input. This work was supported by a National Institute of Health MIRA award [grant number 1 R35 GM 119771-01]; a National Science Foundation INSPIRE award [grant number 1649014]; and the National Science Foundation / ERASynBio [grant numbers 1548123, 335672].

10.4 Questions and comments

Please contact the Karr Lab with any questions or comments.

Bibliography

[1] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In *Encyclopedia of Machine Learning*, pages 257–258. Springer, 2011.

[2] Dagmar Waltemath, Richard Adams, Daniel A Beard, Frank T Bergmann, Upinder S Bhalla, Randall Britten, Vijayalakshmi Chelliah, Michael T Cooling, Jonathan Cooper, Edmund J Crampin, and others. Minimum Information About a Simulation Experiment (MIASE). *PLoS Comput Biol*, 7(4):e1001122, 2011.

[3] Kouichi Takahashi, Kazunari Kaizu, Bin Hu, and Masaru Tomita. A multi-algorithm, multi-timescale method for cell simulation. *Bioinformatics*, 20(4):538–546, 2004.

[4] E L Haseltine and F H Arnold. Synthetic gene circuits: design with directed evolution. *Annu Rev Biophys Biomol Struct*, 36:1–19, 2007.

[5] R E Cobb, T Si, and H Zhao. Directed evolution: an evolving and enabling synthetic biology tool. *Curr Opin Chem Biol*, 16(3-4):285–291, 2012.

[6] Dean C Karnopp, Donald L Margolis, and Ronald C Rosenberg. *System dynamics: modeling, simulation, and control of mechatronic systems*. John Wiley & Sons, 2012.

[7] Joe A Clarke. *Energy simulation in building design*. Routledge, 2001.

[8] Ennio Cascetta. *Transportation systems analysis: models and applications*. Volume 29. Springer Science & Business Media, 2009.

[9] J R Karr, K Takahashi, and A Funahashi. The principles of whole-cell modeling. *Curr Opin Microbiol*, 27:18–24, 2015.

[10] D N Macklin, N A Ruggero, and M W Covert. The future of whole-cell modeling. *Curr Opin Biotechnol*, 28:111–115, 2014.

[11] M Tomita. Whole-cell simulation: a grand challenge of the 21st century. *Trends Biotechnol*, 19(6):205–210, 2001.

[12] Javier Carrera and Markus W Covert. Why build whole-cell models? *Trends Cell Biol*, 25(12):719–722, 2015.

[13] Jeffrey D Orth, Ines Thiele, and Bernhard Ø Palsson. What is flux balance analysis? *Nat Biotechnol*, 28(3):245–248, 2010.

[14] Aarash Bordbar, Jonathan M Monk, Zachary A King, and Bernhard O Palsson. Constraint-based models predict metabolic and associated cellular functions. *Nat Rev Genet*, 15(2):107, 2014.

[15] Adam M Feist and Bernhard Ø Palsson. The growing scope of applications of genome-scale metabolic reconstructions: the case of E. coli. *Nat Biotechnol*, 26(6):659, 2008.

[16] B Szigeti, Y D Roth, J A P Sekar, A P Goldberg, S C Pochiraju, and J R Karr. A blueprint for human whole-cell modeling. *Curr Opin Syst Biol*, In submission.

[17] Masaru Tomita, Kenta Hashimoto, Koichi Takahashi, Thomas Simon Shimizu, Yuri Matsuzaki, Fumihiko Miyoshi, Kanako Saito, Sakura Tanida, Katsuyuki Yugi, J Craig Venter, and others. E-CELL: software environment for whole-cell simulation. *Bioinformatics*, 15(1):72–84, 1999.

[18] Markus W Covert, Eric M Knight, Jennifer L Reed, Markus J Herrgard, and Bernhard O Palsson. Integrating high-throughput and computational data elucidates bacterial networks. *Nature*, 429(6987):92, 2004.

[19] Sriram Chandrasekaran and Nathan D Price. Probabilistic integrative modeling of genome-scale metabolic and regulatory networks in Escherichia coli and Mycobacterium tuberculosis. *Proc Natl Acad Sci U S A*, 107(41):17845–17850, 2010.

[20] Markus W Covert, Nan Xiao, Tiffany J Chen, and Jonathan R Karr. Integrating metabolic, transcriptional regulatory and signal transduction models in Escherichia coli. *Bioinformatics*, 24(18):2044–2050, 2008.

[21] Jong Min Lee, Erwin P Gianchandani, James A Eddy, and Jason A Papin. Dynamic analysis of integrated signaling, metabolic, and regulatory networks. *PLoS Comput Biol*, 4(5):e1000086, 2008.

[22] Javier Carrera, Raissa Estrela, Jing Luo, Navneet Rai, Athanasios Tsoukalas, and Ilias Tagkopoulos. An integrative, multi-scale, genome-wide model reveals the phenotypic landscape of Escherichia coli. *Mol Syst Biol*, 10(7):735, 2014.

[23] Ines Thiele, Neema Jamshidi, Ronan MT Fleming, and Bernhard Ø Palsson. Genome-scale reconstruction of Escherichia coli's transcriptional and translational machinery: a knowledge base, its mathematical formulation, and its functional characterization. *PLoS Comput Biol*, 5(3):e1000312, 2009.

[24] Emanuel Gonçalves, Joachim Bucher, Anke Ryll, Jens Niklas, Klaus Mauch, Steffen Klamt, Miguel Rocha, and Julio Saez-Rodriguez. Bridging the layers: towards integration of signal transduction, regulation and metabolism into mathematical models. *Mol Biosyst*, 9(7):1576–1583, 2013.

[25] JC Atlas, EV Nikolaev, ST Browning, and ML Shuler. Incorporating genome-wide DNA sequence information into a dynamic whole-cell model of Escherichia coli: application to DNA replication. *IET Syst Biol*, 2(5):369–382, 2008.

[26] Elijah Roberts, John E Stone, Leonardo Sepúlveda, Wen-Mei W Hwu, and Zaida Luthey-Schulten. Long time-scale simulations of in vivo diffusion using GPU hardware. In *IEEE Intl Symposium Parallel Distributed Processing*, 1–8. IEEE, 2009.

[27] 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.

[28] Aarash Bordbar, Douglas McCloskey, Daniel C Zielinski, Nikolaus Sonnenschein, Neema Jamshidi, and Bernhard O Palsson. Personalized whole-cell kinetic models of metabolism for discovery in genomics and pharmacodynamics. *Cell Syst*, 1(4):283–292, 2015.

[29] Daniel G Gibson, John I Glass, Carole Lartigue, Vladimir N Noskov, Ray-Yuan Chuang, Mikkel A Algire, Gwynedd A Benders, Michael G Montague, Li Ma, Monzia M Moodie, and others. Creation of a bacterial cell controlled by a chemically synthesized genome. *Science*, 329(5987):52–56, 2010.

[30] J R Karr, J C Sanghvi, D N Macklin, A Arora, and M W Covert. WholeCellKB: model organism databases for comprehensive whole-cell models. *Nucleic Acids Res*, 41(Database issue):D787–D792, 2013.

[31] Nikolay Kolesnikov, Emma Hastings, Maria Keays, Olga Melnichuk, Y Amy Tang, Eleanor Williams, Miroslaw Dylag, Natalja Kurbatova, Marco Brandizi, Tony Burdett, and others. ArrayExpress update–simplifying data submissions. *Nucleic Acids Res*, 43(D1):D1113–D1116, 2015.

[32] Emily Clough and Tanya Barrett. The Gene Expression Omnibus database. *Statistical Genomics: Methods and Protocols*, pages 93–110, 2016.

[33] Mingcong Wang, Christina J Herrmann, Milan Simonovic, Damian Szklarczyk, and Christian Mering. Version 4.0 of PaxDb: protein abundance data, integrated across model organisms, tissues, and cell-lines. *Proteomics*, 15(18):3163–3168, 2015.

[34] Ulrike Wittig, Renate Kania, Martin Golebiewski, Maja Rey, Lei Shi, Lenneke Jong, Enkhjargal Algaa, Andreas Weidemann, Heidrun Sauer-Danzwith, Saqib Mir, and others. SABIO-RK–database for biochemical reaction kinetics. *Nucleic Acids Res*, 40(D1):D790–D796, 2012.

[35] Iain C Macaulay, Chris P Ponting, and Thierry Voet. Single-cell multiomics: multiple measurements from single cells. *Trends Genet*, 33(2):155–168, 2017.

[36] AF Maarten Altelaar, Javier Munoz, and Albert JR Heck. Next-generation proteomics: towards an integrative view of proteome dynamics. *Nat Rev Genet*, 14(1):35, 2013.

[37] Tobias Fuhrer and Nicola Zamboni. High-throughput discovery metabolomics. *Curr Opinion Biotechnol*, 31:73–78, 2015.

[38] Peter W Laird. Principles and challenges of genome-wide DNA methylation analysis. *Nat Rev Genetics*, 11(3):191, 2010.

[39] Job Dekker, Marc A Marti-Renom, and Leonid A Mirny. Exploring the three-dimensional organization of genomes: interpreting chromatin interaction data. *Nat Rev Genet*, 14(6):390, 2013.

[40] Peter J Park. ChIP-seq: advantages and challenges of a maturing technology. *Nat Rev Genet*, 10(10):669, 2009.

[41] Antoine-Emmanuel Saliba, Alexander J Westermann, Stanislaw A Gorski, and Jörg Vogel. Single-cell RNA-seq: advances and future challenges. *Nucleic Acids Res*, 42(14):8845–8860, 2014.

[42] Aleksandra A Kolodziejczyk, Jong Kyoung Kim, Valentine Svensson, John C Marioni, and Sarah A Teichmann. The technology and biology of single-cell RNA sequencing. *Mol Cell*, 58(4):610–620, 2015.

[43] Je Hyuk Lee, Evan R Daugharthy, Jonathan Scheiman, Reza Kalhor, Joyce L Yang, Thomas C Ferrante, Richard Terry, Sauveur SF Jeanty, Chao Li, Ryoji Amamoto, and others. Highly multiplexed subcellular RNA sequencing in situ. *Science*, 343(6177):1360–1363, 2014.

[44] Katja Dettmer, Pavel A Aronov, and Bruce D Hammock. Mass spectrometry-based metabolomics. *Mass Spectrom Rev*, 26(1):51–78, 2007.

[45] Marcus Bantscheff, Simone Lemeer, Mikhail M Savitski, and Bernhard Kuster. Quantitative mass spectrometry in proteomics: critical review update from 2007 to the present. *Anal Bioanal Chem*, 404(4):939–965, 2012.

[46] Sean C Bendall, Garry P Nolan, Mario Roederer, and Pratip K Chattopadhyay. A deep profiler's guide to cytometry. *Trends Immunol*, 33(7):323–332, 2012.

[47] Tanvir Sajed, Ana Marcu, Miguel Ramirez, Allison Pon, An Chi Guo, Craig Knox, Michael Wilson, Jason R Grant, Yannick Djoumbou, and David S Wishart. ECMDB 2.0: a richer resource for understanding the biochemistry of E. coli. *Nucleic Acids Res*, 44(D1):D495–D501, 2016.

[48] Miguel Ramirez-Gaona, Ana Marcu, Allison Pon, An Chi Guo, Tanvir Sajed, Noah A Wishart, Naama Karu, Yannick Djoumbou Feunang, David Arndt, and David S Wishart. YMDB 2.0: a significantly expanded version of the yeast metabolome database. *Nucleic Acids Res*, 45(D1):D440–D445, 2017.

[49] Zachary A King, Justin Lu, Andreas Dräger, Philip Miller, Stephen Federowicz, Joshua A Lerman, Ali Ebrahim, Bernhard O Palsson, and Nathan E Lewis. BiGG models: a platform for integrating, standardizing and sharing genome-scale models. *Nucleic Acids Res*, 44(D1):D515–D522, 2015.

[50] figshare LLP. Figshare. https://figshare.com, 2017.

[51] SimTK Team. Simtk. https://simtk.org, 2017.

[52] CERN. Zenodo. https://zenodo.org, 2017.

[53] Damian Smedley, Syed Haider, Steffen Durinck, Luca Pandini, Paolo Provero, James Allen, Olivier Arnaiz, Mohammad Hamza Awedh, Richard Baldock, Giulia Barbiera, and others. The BioMart community portal: an innovative alternative to large, centralized data repositories. *Nucleic Acids Res*, 43(W1):W589–W598, 2015.

[54] Thomas Cokelaer, Dennis Pultz, Lea M Harder, Jordi Serra-Musach, and Julio Saez-Rodriguez. BioServices: a common Python package to access biological web services programmatically. *Bioinformatics*, 29(24):3241–3242, 2013.

[55] Alex Kalderimis, Rachel Lyne, Daniela Butano, Sergio Contrino, Mike Lyne, Joshua Heimbach, Fengyuan Hu, Richard Smith, Radek Štěpán, Julie Sullivan, and others. InterMine: extensive web services for modern biology. *Nucleic Acids Res*, 42(W1):W468–W472, 2014.

[56] Minoru Kanehisa, Miho Furumichi, Mao Tanabe, Yoko Sato, and Kanae Morishima. KEGG: new perspectives on genomes, pathways, diseases and drugs. *Nucleic Acids Res*, 45(D1):D353–D361, 2017.

[57] Ethan G Cerami, Benjamin E Gross, Emek Demir, Igor Rodchenkov, Özgün Babur, Nadia Anwar, Nikolaus Schultz, Gary D Bader, and Chris Sander. Pathway Commons, a web resource for biological pathway data. *Nucleic Acids Res*, 39(suppl_1):D685–D690, 2010.

[58] UniProt Consortium and others. UniProt: the universal protein knowledgebase. *Nucleic Acids Res*, 45(D1):D158–D169, 2017.

[59] Ron Caspi, Richard Billington, Luciana Ferrer, Hartmut Foerster, Carol A Fulcher, Ingrid M Keseler, Anamika Kothari, Markus Krummenacker, Mario Latendresse, Lukas A Mueller, and others. The MetaCyc database of metabolic pathways and enzymes and the BioCyc collection of pathway/genome databases. *Nucleic Acids Res*, 44(D1):D471–D480, 2016.

[60] Ingrid M Keseler, Amanda Mackie, Alberto Santos-Zavaleta, Richard Billington, César Bonavides-Martínez, Ron Caspi, Carol Fulcher, Socorro Gama-Castro, Anamika Kothari, Markus Krummenacker, and others. The EcoCyc database: reflecting new knowledge about Escherichia coli K-12. *Nucleic Acids Res*, 45(D1):D543–D550, 2017.

[61] Mario Latendresse, Markus Krummenacker, Miles Trupp, and Peter D Karp. Construction and completion of flux balance models from pathway databases. *Bioinformatics*, 28(3):388–396, 2012.

[62] Michael Y Galperin, Xosé M Fernández-Suárez, and Daniel J Rigden. The 24th annual Nucleic Acids Research database issue: a look back and upcoming changes. *Nucleic Acids Res*, 45(D1):D1–D11, 2017.

[63] Heinz Pampel, Paul Vierkant, Frank Scholze, Roland Bertelmann, Maxi Kindling, Jens Klump, Hans-Jürgen Goebelbecker, Jens Gundlach, Peter Schirmbacher, and Uwe Dierolf. Making research data repositories visible: the re3data.org Registry. *PloS One*, 8(11):e78080, 2013.

[64] Paul R Cohen. DARPA's Big Mechanism program. *Phys Biol*, 12(4):045008, 2015.

[65] Nancy Y Yu, James R Wagner, Matthew R Laird, Gabor Melli, Sébastien Rey, Raymond Lo, Phuong Dao, S Cenk Sahinalp, Martin Ester, Leonard J Foster, and others. PSORTb 3.0: improved protein subcellular localization prediction with refined localization subcategories and predictive capabilities for all prokaryotes. *Bioinformatics*, 26(13):1608–1615, 2010.

[66] Vikram Agarwal, George W Bell, Jin-Wu Nam, and David P Bartel. Predicting effective microRNA target sites in mammalian mRNAs. *eLife*, 4:e05005, 2015.

[67] Fedor A Kolpakov, Nikita I Tolstykh, Tagir F Valeev, Ilya N Kiselev, Elena O Kutumova, Anna Ryabova, Ivan S Yevshin, and Alexander E Kel. BioUML–open source plug-in based platform for bioinformatics: invitation to collaboration. In *Moscow Conference on Computational Molecular Biology*, 172–173. Department of Bioengineering and Bioinformatics of MV Lomonosov Moscow State University, 2011.

[68] Yukiko Matsuoka, Akira Funahashi, Samik Ghosh, and Hiroaki Kitano. Modeling and simulation using CellDesigner. *Transcription Factor Regulatory Networks: Methods and Protocols*, pages 121–145, 2014.

[69] Frank T Bergmann, Stefan Hoops, Brian Klahn, Ursula Kummer, Pedro Mendes, Jürgen Pahle, and Sven Sahle. COPASI and its applications in biotechnology. *J Biotechnol*, 261:215–220, 2017.

[70] Herbert M Sauro, Michael Hucka, Andrew Finney, Cameron Wellock, Hamid Bolouri, John Doyle, and Hiroaki Kitano. Next generation simulation tools: the Systems Biology Workbench and BioSPICE integration. *Omics*, 7(4):355–372, 2003.

[71] Diana C Resasco, Fei Gao, Frank Morgan, Igor L Novak, James C Schaff, and Boris M Slepchenko. Virtual Cell: computational tools for modeling in cell biology. *Wiley Interdiscip Rev Syst Biol Med*, 4(2):129–140, 2012.

[72] Adam M Smith, Wen Xu, Yao Sun, James R Faeder, and G Elisabeta Marai. RuleBender: integrated modeling, simulation and visualization for rule-based intracellular biochemistry. *BMC Bioinformatics*, 13(8):S3, 2012.

[73] Ali Ebrahim, Joshua A Lerman, Bernhard O Palsson, and Daniel R Hyduke. COBRApy: constraints-based reconstruction and analysis for Python. *BMC Syst Biol*, 7(1):74, 2013.

[74] Joost Boele, Brett G Olivier, and Bas Teusink. FAME, the flux analysis and modeling environment. *BMC Syst Biol*, 6(1):8, 2012.

[75] Rasmus Agren, Liming Liu, Saeed Shoaie, Wanwipa Vongsangnak, Intawat Nookaew, and Jens Nielsen. The RAVEN toolbox and its use for generating a genome-scale metabolic model for Penicillium chrysogenum. *PLoS Comput Biol*, 9(3):e1002980, 2013.

[76] Katherine Wolstencroft, Stuart Owen, Olga Krebs, Quyen Nguyen, Natalie J Stanford, Martin Golebiewski, Andreas Weidemann, Meik Bittkowski, Lihua An, David Shockley, and others. SEEK: a systems biology data and model management platform. *BMC Syst Biol*, 9(1):33, 2015.

[77] T Helikar, B Kowal, and JA Rogers. A cell simulator platform: the Cell Collective. *Clin Pharmacol Ther*, 93(5):393–395, 2013.

[78] Franco du Preez. JWS Online. *Encyclopedia of Systems Biology*, pages 1063–1066, 2013.

[79] Carlos F Lopez, Jeremy L Muhlich, John A Bachman, and Peter K Sorger. Programming biological models in Python using PySB. *Mol Syst Biol*, 9(1):646, 2013.

[80] Falko Krause, Jannis Uhlendorf, Timo Lubitz, Marvin Schulz, Edda Klipp, and Wolfram Liebermeister. Annotation and merging of SBML models with semanticSBML. *Bioinformatics*, 26(3):421–422, 2009.

[81] Maxwell L Neal, Michael T Cooling, Lucian P Smith, Christopher T Thompson, Herbert M Sauro, Brian E Carlson, Daniel L Cook, and John H Gennari. A reappraisal of how to build modular, reusable models of biological systems. *PLoS Comput Biol*, 10(10):e1003849, 2014.

[82] Paul Kirk, Thomas Thorne, and Michael PH Stumpf. Model selection in systems and synthetic biology. *Curr Opin Biotechnol*, 24(4):767–774, 2013.

[83] Juliane Liepe, Paul Kirk, Sarah Filippi, Tina Toni, Chris P Barnes, and Michael PH Stumpf. A framework for parameter estimation and model selection from experimental data in systems biology using approximate bayesian computation. *Nat Protoc*, 9(2):439, 2014.

[84] Tina Toni, David Welch, Natalja Strelkowa, Andreas Ipsen, and Michael PH Stumpf. Approximate bayesian computation scheme for parameter inference and model selection in dynamical systems. *J R Soc Interface*, 6(31):187–202, 2009.

[85] Max Flöttmann, Jörg Schaber, Stephan Hoops, Edda Klipp, and Pedro Mendes. ModelMage: a tool for automatic model generation, selection and management. *Genome Inform*, 20:52–63, 2008.

[86] Rob Johnson, Paul Kirk, and Michael PH Stumpf. SYSBIONS: nested sampling for systems biology. *Bioinformatics*, 31(4):604–605, 2014.

[87] Jeffrey D Orth and Bernhard Ø Palsson. Systematizing the generation of missing metabolic knowledge. *Biotechnol Bioeng*, 107(3):403–412, 2010.

[88] Edik M Blais, Arvind K Chavali, and Jason A Papin. Linking genome-scale metabolic modeling and genome annotation. *Methods Mol Biol*, pages 61–83, 2013.

[89] Vinay Satish Kumar, Madhukar S Dasika, and Costas D Maranas. Optimization based automated curation of metabolic reconstructions. *BMC Bioinformatics*, 8(1):212, 2007.

[90] Markus J Herrgård, Stephen S Fong, and Bernhard Ø Palsson. Identification of genome-scale metabolic network models using experimentally measured flux profiles. *PLoS Comput Biol*, 2(7):e72, 2006.

[91] Jennifer L Reed, Trina R Patel, Keri H Chen, Andrew R Joyce, Margaret K Applebee, Christopher D Herring, Olivia T Bui, Eric M Knight, Stephen S Fong, and Bernhard O Palsson. Systems approach to refining genome annotation. *Proc Natl Acad Sci U S A*, 103(46):17480–17484, 2006.

[92] Mario Latendresse. Efficiently gap-filling reaction networks. *BMC Bioinformatics*, 15(1):225, 2014.

[93] Vinay Satish Kumar and Costas D Maranas. GrowMatch: an automated method for reconciling in silico/in vivo growth predictions. *PLoS Comput Biol*, 5(3):e1000308, 2009.

[94] Peter Kharchenko, Lifeng Chen, Yoav Freund, Dennis Vitkup, and George M Church. Identifying metabolic enzymes with multiple types of association evidence. *BMC Bioinformatics*, 7(1):177, 2006.

[95] Zhaleh Hosseini and Sayed-Amir Marashi. Discovering missing reactions of metabolic networks by using gene co-expression data. *Sci Rep*, 2017.

[96] Matthew N Benedict, Michael B Mundy, Christopher S Henry, Nicholas Chia, and Nathan D Price. Likelihood-based gene annotations for gap filling and quality assessment in genome-scale metabolic models. *PLoS Comput Biol*, 10(10):e1003882, 2014.

[97] Edward Vitkin and Tomer Shlomi. MIRAGE: a functional genomics-based approach for metabolic network model reconstruction and its application to cyanobacteria networks. *Genome Biol*, 13(11):R111, 2012.

[98] Michelle L Green and Peter D Karp. A Bayesian method for identifying missing enzymes in predicted metabolic pathway databases. *BMC Bioinformatics*, 5(1):76, 2004.

[99] Andrei Osterman. A hidden metabolic pathway exposed. *Proc Natl Acad Sci U S A*, 103(15):5637–5638, 2006.

[100] Alan Garny, David P Nickerson, Jonathan Cooper, Rodrigo Weber dos Santos, Andrew K Miller, Steve McKeever, Poul MF Nielsen, and Peter J Hunter. CellML and associated tools and techniques. *Philos Trans A Math Phys Eng Sci*, 366(1878):3017–3043, 2008.

[101] 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, and others. The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[102] Leonard A Harris, Justin S Hogg, Jose-Juan Tapia, John AP Sekar, Sanjana Gupta, Ilya Korsunsky, Arshi Arora, Dipak Barua, Robert P Sheehan, and James R Faeder. BioNetGen 2.2: advances in rule-based modeling. *Bioinformatics*, 32(21):3366–3368, 2016.

[103] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theor Comput Sci*, 325(1):69–110, 2004.

[104] Carsten Maus, Stefan Rybacki, and Adelinde M Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Syst Biol*, 5(1):166, 2011.

[105] Vijayalakshmi Chelliah, Nick Juty, Ishan Ajmera, Raza Ali, Marine Dumousseau, Mihai Glont, Michael Hucka, Gaël Jalowicki, Sarah Keating, Vincent Knight-Schrijver, and others. BioModels: ten-year anniversary. *Nucleic Acids Res*, 43(D1):D542–D548, 2015.

[106] Dagmar Waltemath, Jonathan R Karr, Frank T Bergmann, Vijayalakshmi Chelliah, Michael Hucka, Marcus Krantz, Wolfram Liebermeister, Pedro Mendes, Chris J Myers, Pinar Pir, and others. Toward community standards and software for whole-cell modeling. *IEEE Trans Biomed Eng*, 63(10):2007–2014, 2016.

[107] Michael W Sneddon, James R Faeder, and Thierry Emonet. Efficient modeling, simulation and coarse-graining of biological complexity with NFsim. *Nat Methods*, 8(2):177–183, 2011.

[108] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *J Phys Chem*, 81(25):2340–2361, 1977.

[109] Vo Hong Thanh, Roberto Zunino, and Corrado Priami. Efficient constant-time complexity algorithm for stochastic simulation of large reaction networks. *IEEE/ACM Trans Comput Biol Bioinform*, 14(3):657–667, 2017.

[110] Dagmar Waltemath, Richard Adams, Frank T Bergmann, Michael Hucka, Fedor Kolpakov, Andrew K Miller, Ion I Moraru, David Nickerson, Sven Sahle, Jacky L Snoep, and others. Reproducible computational biology experiments with SED-ML-the Simulation Experiment Description Markup Language. *BMC Syst Biol*, 5(1):198, 2011.

[111] Roland Ewald and Adelinde M Uhrmacher. SESSL: a domain-specific language for simulation experiments. *ACM Trans Modeling Comput Simul*, 24(2):11, 2014.

[112] Pawan K Dhar, Kouichi Takahashi, Yoichi Nakayama, and Masaru Tomita. E-Cell: computer simulation of the cell. *Rev Cell Biol Mol Med*, 2006.

[113] 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.

[114] 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.

[115] Marco S Nobile, Daniela Besozzi, Paolo Cazzaniga, Giancarlo Mauri, and Dario Pescini. cupSODA: a CUDA-powered simulator of mass-action kinetics. In *International Conference on Parallel Computing Technologies*, 344–357. Springer, 2013.

[116] Marco S Nobile, Paolo Cazzaniga, Daniela Besozzi, Dario Pescini, and Giancarlo Mauri. cuTauLeaping: a GPU-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. *PLoS One*, 9(3):e91963, 2014.

[117] Christopher D Carothers, David Bauer, and Shawn Pearce. ROSS: a high-performance, low-memory, modular Time Warp system. *J Parallel Distrib Comput*, 62(11):1648–1669, 2002.

[118] Julio R Banga and Eva Balsa-Canto. Parameter estimation and optimal experimental design. *Essays in biochemistry*, 45:195–210, 2008.

[119] Alexander IJ Forrester and Andy J Keane. Recent advances in surrogate-based optimization. *Progress Aerospace Sci*, 45(1):50–79, 2009.

[120] Chen Wang, Qingyun Duan, Wei Gong, Aizhong Ye, Zhenhua Di, and Chiyuan Miao. An evaluation of adaptive surrogate modeling based optimization with two benchmark problems. *Environmental Modelling Softw*, 60:167–179, 2014.

[121] Jason P Halloran and Ahmet Erdemir. Adaptive surrogate modeling for expedited estimation of nonlinear tissue properties through inverse finite element analysis. *Annal Biomed Eng*, 39(9):2388–2397, 2011.

[122] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *J Global Optim*, 21(4):345–383, 2001.

[123] Yew S Ong, Prasanth B Nair, and Andrew J Keane. Evolutionary optimization of computationally expensive problems via surrogate modeling. *AIAA journal*, 41(4):687–696, 2003.

[124] Saman Razavi, Bryan A Tolson, and Donald H Burn. Numerical assessment of metamodelling strategies in computationally intensive optimization. *Environmental Modelling & Software*, 34:67–86, 2012.

[125] Nestor V Queipo, Salvador Pintos, Néstor Rincón, Nemrod Contreras, and Juan Colmenares. Surrogate modeling-based optimization for the integration of static and dynamic data into a reservoir description. *Journal of Petroleum Science and Engineering*, 35(3):167–181, 2002.

[126] Liviu Panait and Sean Luke. Cooperative multi-agent learning: the state of the art. *Autonomous Agents Multi-agent Syst*, 11(3):387–434, 2005.

[127] Daniel P Palomar and Yonina C Eldar. *Convex optimization in signal processing and communications*. Cambridge university press, 2010.

[128] Robin L Raffard, Claire J Tomlin, and Stephen P Boyd. Distributed optimization for cooperative agents: application to formation flight. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 3, 2453–2459. IEEE, 2004.

[129] Michael Rabbat and Robert Nowak. Distributed optimization in sensor networks. In *Proceedings of the 3rd international symposium on Information processing in sensor networks*, 20–27. ACM, 2004.

[130] Brian Y Chen, Viacheslav Y Fofanov, Drew H Bryant, Bradley D Dodson, David M Kristensen, Andreas M Lisewski, Marek Kimmel, Olivier Lichtarge, and Lydia E Kavraki. Geometric sieving: automated distributed optimization of 3d motifs for protein function prediction. *Lecture Notes in Computer Science*, 3909:500–515, 2006.

[131] Louis B Rall. *Automatic differentiation: Techniques and applications*. Springer, 1981.

[132] Rohit Ramachandran and Paul I Barton. Effective parameter estimation within a multi-dimensional population balance model framework. *Chemical Engineering Science*, 65(16):4884–4893, 2010.

[133] H Martin Bücker, George Corliss, Paul Hovland, Uwe Naumann, and Boyana Norris. *Automatic differentiation: applications, theory, and implementations*. Volume 50. Springer Science & Business Media, 2006.

[134] Jan Schumann-Bischoff, Stefan Luther, and Ulrich Parlitz. Nonlinear system identification employing automatic differentiation. *Communications in Nonlinear Science and Numerical Simulation*, 18(10):2733–2742, 2013.

[135] Oana-Teodora Chis, Julio R Banga, and Eva Balsa-Canto. Structural identifiability of systems biology models: a critical comparison of methods. *PloS One*, 6(11):e27755, 2011.

[136] Maksat Ashyraliyev, Yves Fomekong-Nanfack, Jaap A Kaandorp, and Joke G Blom. Systems biology: parameter estimation for biochemical models. *FEBS J*, 276(4):886–902, 2009.

[137] I-Chun Chou and Eberhard O Voit. Recent developments in parameter estimation and structure identification of biochemical and genomic systems. *Math Biosci*, 219(2):57–83, 2009.

[138] Jianyong Sun, Jonathan M Garibaldi, and Charlie Hodgman. Parameter estimation using metaheuristics in systems biology: a comprehensive review. *IEEE/ACM Trans Comput Biol Bioinfor*, 9(1):185–202, 2012.

[139] Carmen G Moles, Pedro Mendes, and Julio R Banga. Parameter estimation in biochemical pathways: a comparison of global optimization methods. *Genome Res*, 13(11):2467–2474, 2003.

[140] Giuseppina Bellu, Maria Pia Saccomani, Stefania Audoly, and Leontina D'Angiò. DAISY: a new software tool to test global identifiability of biological and physiological systems. *Comput Meth Program Biomed*, 88(1):52–61, 2007.

[141] David R Penas, Patricia González, Jose A Egea, Ramón Doallo, and Julio R Banga. Parameter estimation in large-scale systems biology models: a parallel and self-adaptive cooperative strategy. *BMC Bioinformatics*, 18(1):52, 2017.

[142] Richard Adams, Allan Clark, Azusa Yamaguchi, Neil Hanlon, Nikos Tsorman, Shakir Ali, Galina Lebedeva, Alexey Goltsov, Anatoly Sorokin, Ozgur E Akman, and others. SBSI: an extensible distributed software infrastructure for parameter estimation in systems biology. *Bioinformatics*, 29(5):664–665, 2013.

[143] Edmund M Clarke, James R Faeder, Christopher J Langmead, Leonard A Harris, Sumit Kumar Jha, and Axel Legay. Statistical model checking in BioLab: applications to the automated analysis of T-cell receptor signaling pathway. In *Int Conf Comput Meth Syst Biol*, 231–250. Springer, 2008.

[144] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: verification of probabilistic real-time systems. In *Comput Aided Verification*, 585–591. Springer, 2011.

[145] Christian Lieven, Moritz Beber, and Nikolaus Sonnenschein. Memote – a testing suite for constraint-based metabolic models. http://easychair.org/smart-program/ICSB2017/2017-08-08.html#talk:51929, 2017.

[146] Cyrus Omar, Jonathan Aldrich, and Richard C Gerkin. Collaborative infrastructure for test-driven scientific model validation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, 524–527. ACM, 2014.

[147] Circle Internet Services Inc. Circleci. https://circleci.com, 2017.

[148] Kohsuke Kawaguchi. Jenkins. https://jenkins.io, 2017.

[149] Software Freedom Conservancy. Git. https://git-scm.com/, 2017.

[150] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the HDF5 technology suite and its applications. In *Proc EDBT/ICDT 2011 Workshop Array Databases*, 36–47. ACM, 2011.

[151] Shabana Vohra, Benjamin A Hall, Daniel A Holdbrook, Syma Khalid, and Philip C Biggin. Bookshelf: a simple curation system for the storage of biomolecular simulation data. *Database*, 2010:baq033, 2010.

[152] Giacomo Finocchiaro, Ting Wang, Rene Hoffmann, Aitor Gonzalez, and Rebecca C Wade. DSMM: a database of simulated molecular motions. *Nucleic Acids Res*, 31(1):456–457, 2003.

[153] Marc W van der Kamp, R Dustin Schaeffer, Amanda L Jonsson, Alexander D Scouras, Andrew M Simms, Rudesh D Toofanny, Noah C Benson, Peter C Anderson, Eric D Merkley, Steven Rysavy, and others. Dynameomics: a comprehensive database of protein dynamics. *Structure*, 18(4):423–435, 2010.

[154] Tim Meyer, Marco D'Abramo, Adam Hospital, Manuel Rueda, Carles Ferrer-Costa, Alberto Pérez, Oliver Carrillo, Jordi Camps, Carles Fenollosa, Dmitry Repchevsky, and others. MoDEL (Molecular Dynamics Extended Library): a database of atomistic molecular dynamics trajectories. *Structure*, 18(11):1399–1409, 2010.

[155] Gerard Lemson and others. Halo and galaxy formation histories from the millennium simulation: public release of a vo-oriented and sql-queryable database for studying the evolution of galaxies in the lambdacdm cosmogony. *arXiv preprint astro-ph/0608019*, 2006.

[156] Kristin Riebe, Adrian M Partl, Harrya Enke, Jaime Forero-Romero, Stefan Gottloeber, Anatolyb Klypin, Gerardc Lemson, Franciscod Prada, Joel R Primack, Matthiasa Steinmetz, and others. The MultiDark database: release of the Bolshoi and MultiDark cosmological simulations. *Astronomische Nachrichten*, 334(7):691–708, 2013.

[157] Katy Wolstencroft, Stuart Owen, Franco du Preez, Olga Krebs, Wolfgang Mueller, Carole Goble, and Jacky L Snoep. The SEEK: a platform for sharing data and models in systems biology. *Meth Enzymol*, 500:629–655, 2011.

[158] J R Karr, N C Phillips, and M W Covert. WholeCellSimDB: a hybrid relational/HDF database for whole-cell model predictions. *Database*, 2014:bau095, 2014.

[159] Zachary A King, Andreas Dräger, Ali Ebrahim, Nikolaus Sonnenschein, Nathan E Lewis, and Bernhard O Palsson. Escher: a web application for building, sharing, and embedding data-rich visualizations of biological pathways. *PLoS Comput Biol*, 11(8):e1004321, 2015.

[160] Suzanne M Paley and Peter D Karp. The Pathway Tools cellular overview diagram and Omics Viewer. *Nucleic Acids Res*, 34(13):3771–3778, 2006.

[161] R Lee, J R Karr, and M W Covert. WholeCellViz: data visualization for whole-cell models. *BMC Bioinformatics*, 14:253, 2013.

[162] Jill C Sible and John J Tyson. Mathematical modeling as a tool for investigating cell cycle control networks. *Methods*, 41(2):238–247, 2007.

[163] Albert Goldbeter. Computational approaches to cellular rhythms. *Nature*, 420(6912):238, 2002.

[164] Andreas VM Herz, Tim Gollisch, Christian K Machens, and Dieter Jaeger. Modeling single-neuron dynamics and computations: a balance of detail and abstraction. *Science*, 314(5796):80–85, 2006.

[165] Neil Swainston, Kieran Smallbone, Hooman Hefzi, Paul D Dobson, Judy Brewer, Michael Hanscho, Daniel C Zielinski, Kok Siong Ang, Natalie J Gardiner, Jahir M Gutierrez, and others. Recon 2.2: from reconstruction to model of human metabolism. *Metabolomics*, 12(7):1–7, 2016.

[166] Rasmus Agren, Sergio Bordel, Adil Mardinoglu, Natapol Pornputtapong, Intawat Nookaew, and Jens Nielsen. Reconstruction of genome-scale active metabolic networks for 69 human cell types and 16 cancer types using INIT. *PLoS Comput Biol*, 8(5):e1002518, 2012.

[167] Mathias Uhlen, Cheng Zhang, Sunjae Lee, Evelina Sjöstedt, Linn Fagerberg, Gholamreza Bidkhori, Rui Benfeitas, Muhammad Arif, Zhengtao Liu, Fredrik Edfors, and others. A pathology atlas of the human cancer transcriptome. *Science*, 357(6352):eaan2507, 2017.

[168] Jacob J Hughey, Timothy K Lee, and Markus W Covert. Computational modeling of mammalian signaling networks. *Wiley Interdiscip Rev Syst Biol Med*, 2(2):194–209, 2010.

[169] Mark B Gerstein, Anshul Kundaje, Manoj Hariharan, Stephen G Landt, Koon-Kiu Yan, Chao Cheng, Xinmeng Jasmine Mu, Ekta Khurana, Joel Rozowsky, Roger Alexander, and others. Architecture of the human regulatory network derived from ENCODE data. *Nature*, 489(7414):91, 2012.

[170] Shigeru Kondo and Takashi Miura. Reaction-diffusion model as a framework for understanding biological pattern formation. *Science*, 329(5999):1616–1620, 2010.

[171] Anja Geitmann and Joseph KE Ortega. Mechanics and modeling of plant cell growth. *Trend Plant Sci*, 14(9):467–478, 2009.

[172] Kerwyn Casey Huang, Yigal Meir, and Ned S Wingreen. Dynamic structures in Escherichia coli: spontaneous formation of MinE rings and MinD polar zones. *Proc Natl Acad Sci U S A*, 100(22):12724–12728, 2003.

[173] Harold P Erickson. Modeling the physics of ftsz assembly and force generation. *Proc Natl Acad Sci U S A*, 106(23):9238–9243, 2009.

[174] Guy Karlebach and Ron Shamir. Modelling and analysis of gene regulatory networks. *Nat Rev Mol Cell Biol*, 9(10):770, 2008.

[175] Tommy Yu, Catherine M Lloyd, David P Nickerson, Michael T Cooling, Andrew K Miller, Alan Garny, Jonna R Terkildsen, James Lawson, Randall D Britten, Peter J Hunter, and others. The Physiome Model Repository 2. *Bioinformatics*, 27(5):743–744, 2011.

[176] Stephen Hilgartner. Constituting large-scale biology: building a regime of governance in the early years of the Human Genome Project. *BioSocieties*, 8(4):397–416, 2013.

[177] Francis S Collins, Michael Morgan, and Aristides Patrinos. The Human Genome Project: lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

[178] Stephen Heller, Alan McNaught, Stephen Stein, Dmitrii Tchekhovskoi, and Igor Pletnev. InChI-the worldwide chemical structure identifier standard. *J Cheminform*, 5(1):7, 2013.

[179] J R Karr, A H Williams, J D Zucker, A Raue, B Steiert, J Timmer, C Kreutz, DREAM8 Parameter Estimation Challenge Consortium, S Wilkinson, B A Allgood, B M Bot, B R Hoff, M R Kellen, M W Covert, G A Stolovitzky, and P Meyer. Summary of the DREAM8 parameter estimation challenge: toward parameter identification for whole-cell models. *PLoS Comput Biol*, 2015.

[180] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: a grammar of interactive graphics. *IEEE Trans Vis Comput Graphics*, 23(1):341–350, 2017.

[181] ML Shuler, S Leung, and CC Dick. A mathematical model for the growth of a single bacterial cell. *Annals of the New York Academy of Sciences*, 326(1):35–52, 1979.

[182] Elijah Roberts. Cellular and molecular structure as a unifying framework for whole-cell modeling. *Curr Opin Structural Biol*, 25:86–91, 2014.

[183] Michael J Hallock, John E Stone, Elijah Roberts, Corey Fry, and Zaida Luthey-Schulten. Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations. *Parallel Comput*, 40(5):86–99, 2014.

[184] John A Cole, Lars Kohler, Jamila Hedhli, and Zaida Luthey-Schulten. Spatially-resolved metabolic cooperativity within dense bacterial colonies. *BMC Syst Biol*, 9(1):15, 2015.

[185] Oliver Purcell, Bonny Jain, Jonathan R Karr, Markus W Covert, and Timothy K Lu. Towards a whole-cell modeling approach for synthetic biology. *Chaos*, 23(2):025112, 2013.

[186] Denis Kazakiewicz, Jonathan R Karr, Karol M Langner, and Dariusz Plewczynski. A combined systems and structural modeling approach repositions antibiotics for Mycoplasma genitalium. *Comput Biol Chem*, 59:91–97, 2015.

[187] Doug Howe, Maria Costanzo, Petra Fey, Takashi Gojobori, Linda Hannick, Winston Hide, David P Hill, Renate Kania, Mary Schaeffer, Susan St Pierre, and others. Big data: the future of biocuration. *Nature*, 455(7209):47–50, 2008.

[188] Jacky L Snoep, Frank Bruggeman, Brett G Olivier, and Hans V Westerhoff. Towards building the silicon cell: a modular approach. *Biosystems*, 83(2):207–216, 2006.

[189] J Kyle Medley, Arthur P Goldberg, and Jonathan R Karr. Guidelines for reproducibly building and simulating systems biology models. *IEEE Trans Biomed Eng*, 63(10):2015–2020, 2016.

[190] Arthur P Goldberg, Yin Hoon Chew, and Jonathan R Karr. Toward scalable whole-cell modeling of human cells. In *Proc 2016 Annu ACM Conf SIGSIM Princip Adv Discret Simul*, 259–262. ACM, 2016.

[191] Ken Martin, Will Schroeder, and Bill Lorensen. Vtk: the visualization toolkit. https://www.vtk.org, 2017.

[192] ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57–74, 2012.

[193] James A Thomson, Joseph Itskovitz-Eldor, Sander S Shapiro, Michelle A Waknitz, Jennifer J Swiergiel, Vivienne S Marshall, and Jeffrey M Jones. Embryonic stem cell lines derived from human blastocysts. *Science*, 282(5391):1145–1147, 1998.

[194] Peter Löser, Jacqueline Schirm, Anke Guhr, Anna M Wobus, and Andreas Kurtz. Human embryonic stem cell lines and their use in international research. *Stem Cells*, 28(2):240–246, 2010.

[195] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software Systems Modeling*, 15(3):609–629, 2016.

[196] Jonathan R Karr, Maria Lluch-Senar, Luis Serrano, and Javier Carrera. The 2016 Whole-Cell Modeling Summer School. 2017. doi:10.5281/zenodo.1004027.

[197] Jonathan R Karr, Maria Lluch-Senar, Luis Serrano, and Damjana Kastelic. The 2017 Whole-Cell Modeling Summer School. 2017. doi:10.5281/zenodo.1004135.

Index

U

V

W