

Rule-Based Modeling of Biological Systems

Theory and practice with BioNetGen

James R. Faeder

2026-10-13

Table of contents

Preface	6
Who this book is for	6
How the book is organized	6
Tools	7
Conventions	7
Acknowledgments	8
License	9
 I Foundations	 10
1 Reaction networks as dynamical systems	11
1.1 Why model reaction networks?	11
1.2 A cytokine signaling circuit	12
1.3 Step 1: Ligand–receptor binding	13
1.3.1 Reactions and mass action	13
1.3.2 From reactions to differential equations	13
1.3.3 The stoichiometry matrix	14
1.3.4 Conservation laws and equilibrium	14
1.3.5 Solving the equations numerically	15
1.4 Writing the model in BioNetGen	16
1.5 Building the circuit step by step	18
1.5.1 Step 2: STAT binds the ligand-bound receptor	18
1.5.2 Step 3: Receptor-bound STAT is activated	19
1.5.3 Step 4: Activated STAT is released and deactivated	20
1.5.4 Step 5: pSTAT induces SOCS expression	21
1.5.5 Step 6: SOCS closes the feedback loop	23
1.6 Negative feedback turns a sustained signal into a pulse	24
1.7 The complete network and its limits	25
1.8 Summary	26
1.9 Exercises	26
1.10 Further reading	27
 2 Molecules, patterns, and rules	 28
2.1 Signaling proteins have many interaction sites	28
2.2 Molecules with structure	29
2.3 Complexes are graphs	30
2.4 Patterns	31
2.4.1 Observables are patterns	32

2.5	Rules	32
2.5.1	Rule diagrams	33
2.6	The Chapter 1 model as rules	34
2.6.1	Network generation	37
2.7	Relaxing the assumptions	38
2.7.1	Ligand can leave any receptor complex	39
2.7.2	SOCS binds its own site and blocks kinase activity	39
2.7.3	Cytokines bring two receptor chains together	39
2.7.4	STATs act as dimers	41
2.7.5	A second phosphorylation site	41
2.8	Rules grow linearly; networks grow combinatorially	41
2.9	Mechanistic details change the predictions	44
2.10	When networks are too large to generate	46
2.11	Summary	47
2.12	Exercises	47
2.13	Further reading	47
3	Stochastic simulation and noise	48
3.1	Why molecule numbers matter	48
3.2	From concentrations to molecule numbers	49
3.3	The stochastic description of a reaction network	50
3.4	Gillespie's algorithm	50
3.5	Noise and system size	52
3.6	Bursts in gene expression	54
3.7	A noisy feedback loop	56
3.8	Faster approximate methods	59
3.9	Summary	59
3.10	Exercises	59
3.11	Further reading	60
4	A field guide to rules	61
4.1	Reading the generated network	61
4.2	Binding	62
4.2.1	Identical partners	62
4.2.2	Multiple identical sites	62
4.2.3	Cooperativity through context	63
4.3	Enzymatic modification	63
4.4	Making and removing molecules	66
4.5	Rate laws from functions	66
4.6	Chains, rings, and the size of the network	67
4.7	Common mistakes	68
4.8	Summary	69
4.9	Exercises	69
4.10	Further reading	70

II	Building realistic models	71
5	Compartments and transport	72
5.1	Declaring compartments	72
5.2	Where molecules are	73
5.3	How compartment size sets the rate	74
5.4	Transport	77
5.5	STAT in a cell with a nucleus	77
5.6	Endocytosis and receptor trafficking	80
5.7	Common mistakes	84
5.8	Summary	84
5.9	Exercises	84
5.10	Further reading	85
6	Network-free simulation	86
6.1	Rules without a network	86
6.2	Running a network-free simulation	87
6.3	Receptor aggregation and the sol–gel transition	89
6.4	Cost and scaling	94
6.5	Common mistakes	96
6.6	Summary	96
6.7	Exercises	97
6.8	Further reading	97
7	Analyzing models	98
7.1	Conservation laws	98
7.2	Steady states and their stability	99
7.3	Dose–response curves	102
7.4	Sensitivity analysis	104
7.5	Switches and memory	107
7.6	Common mistakes	111
7.7	Summary	111
7.8	Exercises	112
7.9	Further reading	112
8	Energy-based rule-based modeling	113
8.1	A steady state with a circulating flux	113
8.2	Free energies of molecular structures	115
8.2.1	Concentration references and affinity	116
8.3	From energy differences to rates	117
8.3.1	Checking the generated dynamics	117
8.4	Cooperativity and a binding polynomial	120
8.4.1	A reciprocal interaction between RAF dimerization and drug binding	122
8.5	Equilibrium and chemically driven signaling	124
8.6	Choosing parameters for calibration	124
8.7	Summary	125
8.8	Exercises	125
8.9	Further reading	126

References	127
Appendices	131
A Installing the tools	131
A.1 No installation: BNG Playground	131
A.2 Visual Studio Code	131
A.3 Python	131
A.4 The book’s models and code	132
A.5 Reproducing the book	132
B BNGL language reference	134
B.1 File structure and lexical conventions	134
B.2 Parameters, expressions, and units	136
B.3 Molecule types and molecular species	137
B.4 Patterns and matching	138
B.5 Observables	139
B.6 Reaction rules and graph changes	139
B.6.1 Deletion and rule modifiers	140
B.7 Functions and rate-law conventions	141
B.7.1 Local functions and tags	142
B.7.2 Specialized network rate laws	142
B.8 Compartments and transport	143
B.9 Energy patterns	144
B.10 Network generation	145
B.11 Simulation actions	145
B.12 Parameter changes, protocols, and scans	146
B.13 Exports, visualization, and output files	148
B.14 Diagnosing syntax and execution errors	149
B.15 Additional language constructs and source locations	150

Preface

Cells process information with networks of interacting proteins, genes, and small molecules. To understand how these networks behave, we write down what we know about their parts and interactions and ask what that knowledge implies. Mathematical models make that question precise. This book teaches **rule-based modeling**, an approach to building such models that suits networks of proteins with many interaction sites, using the [BioNetGen](#) language and tools (Blinov et al. 2004; Faeder et al. 2009; Harris et al. 2016).

Who this book is for

The book has three audiences in mind:

- **Students** in courses on computational and systems biology. The early chapters assume a little calculus and some Python, but no prior modeling experience.
- **New members of a modeling lab** who need to become productive with rule-based models: building them, simulating them, fitting them to data, and sharing them.
- **Experimental and computational biologists** who want a practical introduction to the BioNetGen tools.

How the book is organized

Each chapter introduces one set of ideas and applies them to a running example, a cytokine signaling circuit in which STAT transcription factors induce their own inhibitor, SOCS. The chapters are grouped into parts.

Part I. Foundations

- Chapter [1](#): Builds the STAT–SOCS circuit one reaction at a time and shows how a reaction network becomes a set of differential equations.
- Chapter [2](#): Shows why naming every complex by hand breaks down, and how rules that act on structured molecules solve the problem.
- Chapter [3](#): Describes reactions among small numbers of molecules with the chemical master equation and Gillespie’s algorithm, and follows the noise from a single SOCS gene through the feedback loop.
- Chapter [4](#): Writes rules for binding, enzymatic modification, synthesis and degradation, and aggregation, and checks each one against the reactions BioNetGen generates.

Part II. Building realistic models

- Chapter 5: Places the circuit in a cell with a nucleus and endosomes, and shows how compartment sizes scale the rates of reactions.
- Chapter 6: Simulates models whose networks are too large to generate, using receptor aggregation as the example.
- Chapter 7: Computes conservation laws, steady states and their stability, dose-response curves, sensitivities, and bistability.
- Chapter 8: Assigns free energies to molecular structures so that cooperative binding satisfies detailed balance, and distinguishes equilibrium from signaling driven by ATP hydrolysis.

In preparation. Part III, *Models meet data*, will cover parameter estimation, uncertainty and identifiability, building and sharing models, and AI-assisted modeling. Part IV will cover advanced topics, including spatial simulation, rare events, and visualization, and Part V will present case studies. The plan for the full book is in [BOOK_PLAN.md](#).

Every model in the book is in the [models/](#) folder of the book’s repository, organized by chapter, and every figure is produced by running those models. Each chapter ends with exercises. Lecture slides and a printable worksheet for a course based on Chapters 1 and 2 are in the [course repository](#).

Tools

You can run the models in this book in three ways:

1. In a web browser, with no installation, using [BNG Playground](#).
2. In Visual Studio Code with the BioNetGen extension, which provides syntax highlighting, one-click simulation, and plotting.
3. From Python, using [bngsim](#) for fast ODE and stochastic simulation. This is how the figures in the book are made, it runs every model in the book, and it is the route to the analysis and fitting covered in later chapters.

Appendix [A](#) explains how to install them, and Appendix [B](#) summarizes the syntax of BNGL.

Conventions

BNGL code is shown in shaded blocks:

```
begin reaction rules
  LRbind: L(r) + R(l) <-> L(r!1).R(l!1) kon, koff
end reaction rules
```

Python code that produces a figure is shown above the figure in the web version of the book. Unless stated otherwise, concentrations are in micromolar (μM) and time is in minutes. The parameter values in the running example are illustrative, chosen to show the behavior clearly rather than fitted to a particular cell type.

Acknowledgments

BioNetGen began in the Cell Signaling Team at Los Alamos National Laboratory, which Byron Goldstein founded. The team included his late wife, Carla Wofsy, together with Tony Redondo, Bill Hlavacek, Michael Blinov, and me. The problem that led to BioNetGen, the combinatorial complexity of the complexes that form in receptor signaling, came out of that team's work (Hlavacek et al. 2003), and the first version of the software was written there (Blinov et al. 2004).

I thank Byron and the rest of the Cell Signaling Team at that time, Tony, Carla, and Bill, for inviting me to join them in their odyssey to map and model the world of biological complexes.

Many people have contributed to BioNetGen and the tools built around it since then. Development moved to my group in the Department of Computational and Systems Biology at the University of Pittsburgh, where Leonard Harris, Justin Hogg, José-Juan Tapia, John Sekar, Sanjana Gupta, Ilya Korsunsky, Arshi Arora, Dipak Barua, and Robert Sheehan contributed to BioNetGen 2 (Harris et al. 2016). Leonard Harris and Justin Hogg led the development of compartmental BNGL (Harris et al. 2009). John Sekar and Justin Hogg led the development of energy-based modeling (Sekar et al. 2016), and John Sekar and José-Juan Tapia led the development of the automated visualization of rule-based models used in Chapter 2. José-Juan Tapia also led the development of Atomizer, which converts reaction-network models into rule-based models. Michael Sneddon led the development of NFsim, in a collaboration with Thierry Emonet (Sneddon et al. 2011). Arvind Rasi Subramaniam, an investigator at the Fred Hutchinson Cancer Center in Seattle, pioneered the use of rule-based models to describe translation and also contributed to NFsim. Wen Xu and Adam Smith led the development of RuleBender, a graphical interface for BioNetGen, in a collaboration with G. Elisabeta Marai (Xu et al. 2011), and Yao Sun and Robert Clark also contributed to it. Ali Sinan Saglam created PyBioNetGen and the BioNetGen extension for Visual Studio Code, with contributions from Alex DiBiasi, Nika Zaslavsky, Caleb Armstrong, and Achyudhan Kutuva. Achyudhan Kutuva developed BNG Playground. Adam Husar led the development of MCell4, which integrates particle-based stochastic simulation with rule-based modeling (Husar et al. 2024). Integration of rule-based modeling into MCell4 was developed as part of my longtime collaboration with Tom Bartol and Terry Sejnowski at the Salk Institute, and it built on earlier work by José-Juan Tapia, Ali Sinan Saglam, and Jacob Czech.

At Los Alamos and at Northern Arizona University, Bill Hlavacek, Richard Posner, and their colleagues have extended BioNetGen with tools for simulation and parameter estimation. Jin Yang, Michael Monine, Joshua Colvin, and Ryan Gutenkunst developed network-free simulation methods and RuleMonkey (Yang et al. 2008; Colvin et al. 2010); Brandon Thomas, Lily Chylek, and Joshua Colvin developed BioNetFit (Thomas et al. 2016); Eshan Mitra and Ryan Suderman developed PyBioNetFit (Mitra et al. 2019); and Yen Ting Lin and Song Feng developed the partial scaling algorithm (Lin et al. 2019). Bill Hlavacek also wrote bngsim, which produced every simulation in this book. The commit histories of the [RuleWorld](#) repositories record many further contributions. Alan Perelson, Walter Fontana, and Michael Hucka did not contribute directly to BioNetGen, but they encouraged its development in a variety of ways, some intentional and some not. Vincent Danos independently envisioned rule-based modeling as a solution to the problem of combinatorial complexity, and his work inspired, directly and indirectly, many of the subsequent developments in BioNetGen. He and his collaborators in developing the Kappa language, including Jérôme Feret, Russell Harmer, and Jean Krivine, made major contributions to the theory and applications of rule-based modeling (Danos, Feret, Fontana, Harmer, et al. 2007; Danos, Feret, Fontana, and Krivine 2007; Feret et al. 2009).

Development of BioNetGen has been supported by the National Institutes of Health, through the National Center for Multiscale Modeling of Biological Systems (MMBioS, P41GM103712), which ended in 2022, and grants GM076570, GM085273, AI35997, and CA109552; by the National Science Foundation (grant 0829788); by the Arizona Biomedical Research Commission; and by the Department of Computational Biology at the University of Pittsburgh School of Medicine. Work at Los Alamos National Laboratory was performed under Department of Energy contract DE-AC52-06NA25396.

Finally, I thank the students of all ages and career stages who have taken the time to learn BioNetGen and to develop models with it. Their enthusiasm has been the main encouragement for me to assemble this book.

License

The text and figures are licensed under [CC BY 4.0](#), and the code and models under the MIT License. You are free to reuse and adapt them with attribution.

Part I

Foundations

1 Reaction networks as dynamical systems

A cell senses its environment through receptors, relays the signal through chains of protein interactions, and responds by changing which genes it expresses. Each of these steps is a chemical reaction: molecules bind, unbind, are modified, are made, and are destroyed. In this chapter we turn a small signaling circuit into a set of reactions, show how the reactions define differential equations for the amount of each molecule, and solve those equations with BioNetGen. Throughout the book we treat a model as a precise statement of a mechanism, and simulation as the way to find out what that mechanism implies.

Learning goals

After this chapter you should be able to:

- describe a biochemical process as a set of species and reactions;
- write the rate of each reaction using mass-action kinetics;
- turn a reaction network into ordinary differential equations (ODEs);
- write, run, and interpret a reaction network model in the BioNetGen language (BNGL);
- explain how a negative feedback loop shapes a signaling response.

1.1 Why model reaction networks?

A **reaction network model** has four ingredients:

- **species**: the molecules and complexes whose amounts we track;
- **reactions**: the ways species are converted into one another;
- **rate laws** with **rate constants**: how fast each reaction occurs;
- **initial conditions**: the amount of each species at the start.

The same framework describes atmospheric chemistry, the spread of an epidemic, predator–prey ecology, and the signaling and gene regulatory networks that control cell behavior. In each case the model is built from simple elementary steps, and its behavior emerges from how the steps are connected.

Cell biology needs this kind of reasoning because cellular networks are large and contain many feedback loops. Cell fate decisions such as proliferation, differentiation, and death depend on interacting modules for growth-factor signaling, the cell cycle, DNA damage, and apoptosis (Tyson and Novák 2010). Integrated models of whole cells link many such modules (Bouhaddou et al. 2018), but they are assembled from the same kind of elementary reactions we write in this chapter. A model gives us:

- a precise statement of a proposed mechanism, which can be checked for consistency;

- predictions of dynamics that are hard to measure directly;
- a way to test whether a mechanism can explain experimental data;
- hypotheses that suggest the next experiment.

1.2 A cytokine signaling circuit

Our running example is a simplified version of how cells respond to **cytokines**, secreted proteins such as interleukins and interferons that coordinate immune responses (Figure 1.1).

- A cytokine (the ligand, L) binds its receptor (R) at the cell surface. The ligand-bound receptor activates kinases of the JAK family that are associated with it.
- STAT transcription factors (S) dock on the active receptor and are phosphorylated by JAK. Phosphorylated STAT (pSTAT) leaves the receptor and moves to the nucleus.
- In the nucleus pSTAT binds the promoters of target genes. One of these encodes SOCS (suppressor of cytokine signaling).
- SOCS protein binds the receptor and blocks further STAT activation. Because the output of the pathway inhibits its own input, the circuit is a **negative feedback loop**.

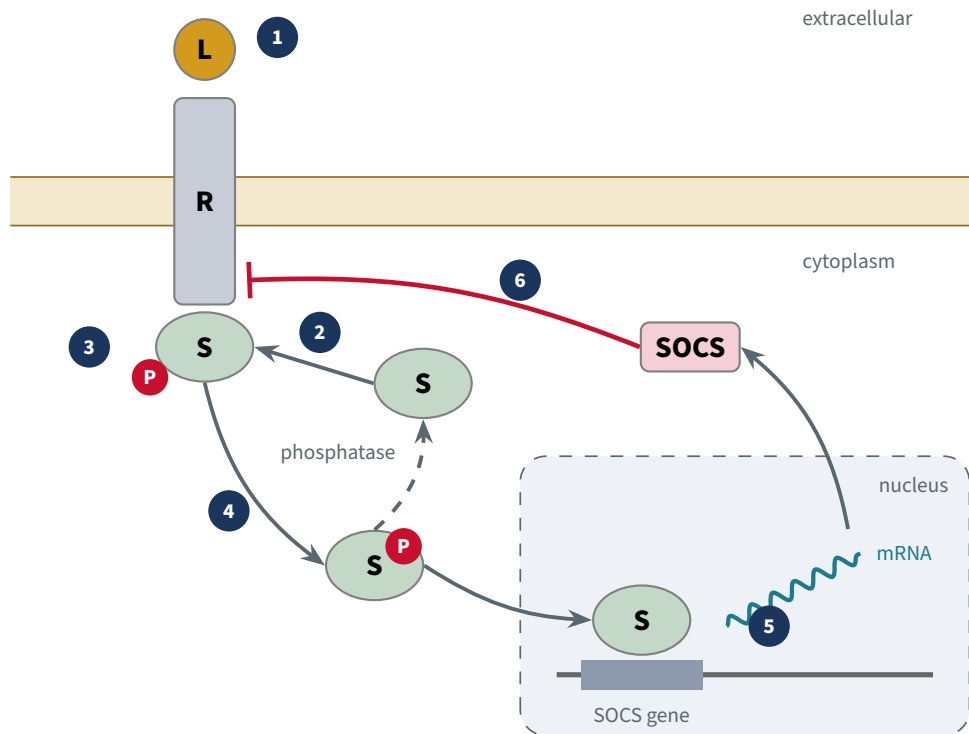


Figure 1.1: The cytokine–STAT–SOCS circuit. Numbers mark the six steps we model in this chapter: (1) ligand binding, (2) STAT recruitment to the ligand-bound receptor, (3) STAT activation, (4) release of pSTAT and its dephosphorylation, (5) induction of SOCS, and (6) SOCS feedback on the receptor.

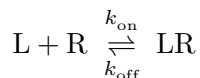
We will build a model of this circuit one step at a time, adding the reactions for each step to the

previous model and simulating as we go. The model leaves out many known details, such as STAT dimerization, the separate receptor chains, and transport into and out of the nucleus, some of which return in later chapters. Starting from a simple model makes each mechanism easier to follow, and the simple model already shows how feedback shapes the response.

1.3 Step 1: Ligand–receptor binding

1.3.1 Reactions and mass action

Binding of a ligand to its receptor is reversible:



We treat this as two **elementary reactions**: binding, which consumes one L and one R and produces one LR, and unbinding, which does the reverse. The rate of each is given by the **law of mass action**, first stated by the Norwegian chemists Cato Guldberg and Peter Waage in the 1860s, which says that the rate of an elementary reaction is proportional to the product of the concentrations of its reactants. The proportionality constant is called the **rate constant**. Writing $[X]$ for the concentration of species X, the two rates are

$$v_1 = k_{\text{on}} [L] [R], \quad v_2 = k_{\text{off}} [LR].$$

The units follow from the rate law. Rates have units of concentration per time, so the second-order constant k_{on} has units of $\mu\text{M}^{-1} \text{min}^{-1}$ and the first-order constant k_{off} has units of min^{-1} . Mass action applies to elementary reactions, and it is a good description when molecules are well mixed and each reaction event requires a single encounter of its reactants. Later chapters consider what happens when these assumptions fail.

1.3.2 From reactions to differential equations

Each reaction changes the concentration of each species it involves: reactants are consumed and products are created. Binding lowers $[L]$ and $[R]$ and raises $[LR]$; unbinding does the reverse. Adding up the contributions of all reactions to each species gives one ordinary differential equation per species:

$$\begin{aligned} \frac{d[L]}{dt} &= -k_{\text{on}}[L][R] + k_{\text{off}}[LR], \\ \frac{d[R]}{dt} &= -k_{\text{on}}[L][R] + k_{\text{off}}[LR], \\ \frac{d[LR]}{dt} &= +k_{\text{on}}[L][R] - k_{\text{off}}[LR]. \end{aligned} \tag{1.1}$$

Once the reactions and rate laws are written down, the differential equations follow mechanically, which is why software such as BioNetGen can generate them for us.

1.3.3 The stoichiometry matrix

The bookkeeping in Equation 1.1 has a compact general form. Collect the concentrations in a vector $\mathbf{X}$ and the reaction rates in a vector $\mathbf{v}$. Then

$$\frac{d\mathbf{X}}{dt} = \mathbf{S} \mathbf{v}(\mathbf{X}), \quad \frac{d}{dt} \begin{pmatrix} [\text{L}] \\ [\text{R}] \\ [\text{LR}] \end{pmatrix} = \begin{pmatrix} -1 & +1 \\ -1 & +1 \\ +1 & -1 \end{pmatrix} \begin{pmatrix} k_{\text{on}}[\text{L}][\text{R}] \\ k_{\text{off}}[\text{LR}] \end{pmatrix}, \quad (1.2)$$

where the **stoichiometry matrix** $\mathbf{S}$ has one row per species and one column per reaction. Its entries are

$$S_{ij} = (\text{times species } i \text{ is a product of reaction } j) \\ - (\text{times species } i \text{ is a reactant of reaction } j).$$

Equation 1.2 holds for any reaction network; only $\mathbf{S}$ and the rate laws in $\mathbf{v}$ change from one network to another.

1.3.4 Conservation laws and equilibrium

Binding only moves molecules between free and bound forms. The total amounts of ligand and receptor are therefore constant:

$$[\text{L}] + [\text{LR}] = L_0, \quad [\text{R}] + [\text{LR}] = R_0,$$

where L_0 and R_0 are the total concentrations of ligand and receptor, free and bound. When the system starts with no complex, as here, they are the initial concentrations of L and R. These **conservation laws** appear as linear dependencies among the rows of $\mathbf{S}$. Here the rows for L and R are both equal to minus the row for LR, so only one of the three equations is independent.

Consider what happens after a long time, when the system has reached equilibrium. Binding and unbinding then occur at the same rate ($v_1 = v_2$), so

$$\frac{[\text{L}][\text{R}]}{[\text{LR}]} = \frac{k_{\text{off}}}{k_{\text{on}}} \equiv K_D,$$

which defines the **dissociation constant** K_D . Substituting the conservation laws gives a quadratic equation for the equilibrium amount of complex, whose physically meaningful root is

$$[\text{LR}]_{\text{eq}} = \frac{1}{2} \left[b - \sqrt{b^2 - 4L_0R_0} \right], \quad b = L_0 + R_0 + K_D. \quad (1.3)$$

1.3.5 Solving the equations numerically

A model this small can be solved by hand, but the equations for realistic signaling networks are nonlinear and usually have no closed-form solution, so we integrate them numerically. The simplest scheme, forward Euler, takes small time steps,

$$\mathbf{X}(t + \Delta t) \approx \mathbf{X}(t) + \Delta t \mathbf{S} \mathbf{v}(\mathbf{X}(t)).$$

In practice, reactions in a network often occur on very different time scales. When the fastest processes relax much more quickly than the dynamics of interest, the equations are **stiff**, meaning that simple methods like Euler's need impractically small steps to remain stable. Production solvers such as CVODE adapt the step size and use implicit methods designed for stiff systems. BioNetGen and bngsim use CVODE for ODE simulation.

Figure 1.2 shows binding over time at three ligand concentrations, computed with bngsim from the BNGL model introduced in the next section. The code that produced it is shown above the figure. It loads the model, changes the initial ligand concentration L0, and runs an ODE simulation for each value.

```
model = bngsim.Model.from_bngl(model_path("ch01", "stat_step1_binding"))
sim = bngsim.Simulator(model, method="ode")

R0, KD = 1.0, 0.1
for L0 in [10, 1, 0.1]:
    model.set_param("L0", L0)
    model.reset() # back to initial conditions
    result = sim.run(t_span=(0, 5), n_points=501)
    line, = plt.plot(result.time, result.observables["LR"],
                     label=f"$L_0$ = {L0} μM")
    b = L0 + R0 + KD # equilibrium, from the formula
    LR_eq = (b - np.sqrt(b**2 - 4 * L0 * R0)) / 2
    plt.axhline(LR_eq, ls="--", lw=1, color=line.get_color())
model.set_param("L0", 10) # restore default
plt.xlabel("time (min)")
plt.ylabel("[LR] (μM)")
plt.legend(loc="center right");
```

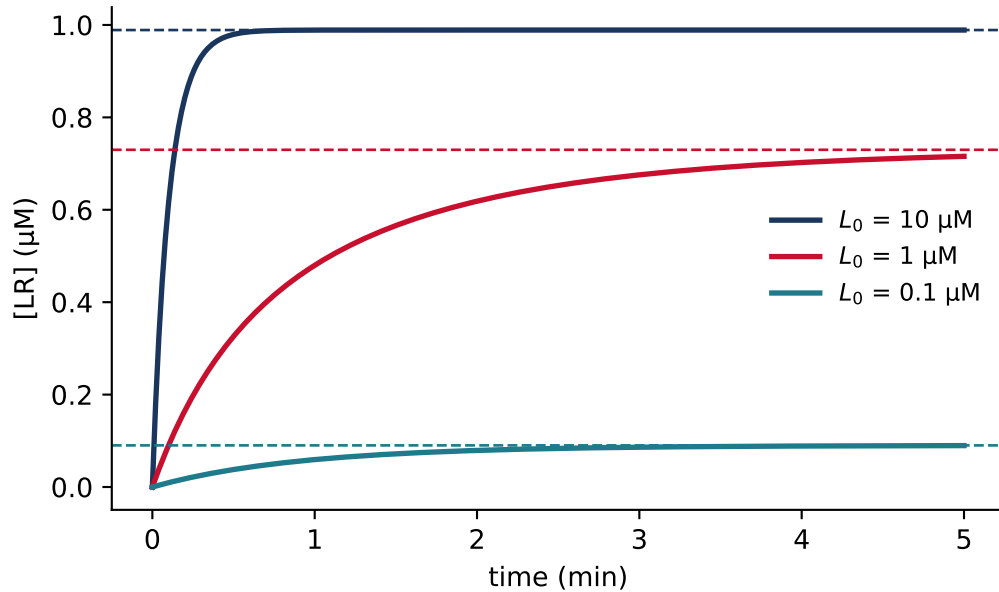


Figure 1.2: Ligand–receptor binding approaches equilibrium faster, and binds more receptors, at higher ligand concentration. Solid lines are numerical solutions; dashed lines are the equilibrium values from Equation 1.3. $R_0 = 1 \mu\text{M}$, $k_{\text{on}} = 1 \mu\text{M}^{-1}\text{min}^{-1}$, $k_{\text{off}} = 0.1 \text{ min}^{-1}$ ($K_D = 0.1 \mu\text{M}$).

Given that at $L_0 = 10 \mu\text{M}$ the ligand is in large excess over both K_D and R_0 , one expects almost every receptor to be bound, and the simulation confirms this. Adding more ligand can increase the response only slightly, since the receptors are already nearly **saturated**. Binding is also faster at higher ligand concentration. Near equilibrium the complex relaxes at rate $k_{\text{on}}([L] + [R]) + k_{\text{off}}$, which grows with the amount of free ligand.

1.4 Writing the model in BioNetGen

Here is the complete model file for ligand–receptor binding, `models/ch01/stat_step1_binding.bngl`:

```
begin model

begin parameters
  L0      10    # ligand (uM)
  R0      1     # receptor (uM)
  kon     1     # binding (1/uM/min)
  koff    0.1   # unbinding (1/min)
end parameters

begin molecule types
  L
  R
```

```

    LR
end molecule types

begin species
    L    L0
    R    R0
end species

begin observables
    Molecules Lfree  L
    Molecules Rfree  R
    Molecules LR     LR
end observables

begin reaction rules
    # Ligand binds receptor
    LRbind: L + R <-> LR kon, koff
end reaction rules

end model

generate_network({overwrite=>1})
simulate({method=>"ode",t_end=>2,n_steps=>200})

```

A BNGL file has a **model** section, between **begin model** and **end model**, and an **actions** section after it.

- **parameters** names every quantity that appears in the model, with its value. Lines starting with # are comments.
- **molecule types** declares every kind of species. In this chapter species have no internal structure, so each is just a name. Chapter 2 gives molecules binding sites and states.
- **species** lists the species present at the start and their initial amounts. Here these are given by the parameters L0 and R0. Species not listed start at zero.
- **observables** defines the quantities to report. **Molecules LR LR** reports the amount of species LR under the name LR.
- **reaction rules** lists the reactions. <-> marks a reversible reaction, followed by the forward and reverse rate constants. The optional label (LRbind:) names the rule.
- The **actions** tell BioNetGen what to do with the model: **generate_network** builds the list of species and reactions, and **simulate** integrates the ODEs and writes the observables to a .gdat file.

You can run the model in three ways. In VS Code with the BioNetGen extension, open the file and click the run button, and the results open in a plot viewer. In BNG Playground, paste the file into the editor and click **Run**. From Python, load the model with bngsim, as in the code for Figure 1.2:

```

model = bngsim.Model.from_bngl("models/ch01/stat_step1_binding.bngl")
sim = bngsim.Simulator(model, method="ode")
model.reset()
result = sim.run(t_span=(0, 20), n_points=101)
result.observables["LR"]          # time course of LR

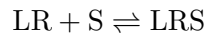
```

`Model.from_bngl` uses BioNetGen to generate the network but does not run the file's actions, so the simulation is controlled entirely from Python. Parameters are changed on the model with `model.set_param`, and `model.reset()` returns the species to their initial amounts. It is important to note that a run starts from the model's current state, which is wherever the previous run ended, so `model.reset()` should be called before each run.

1.5 Building the circuit step by step

Each of the remaining steps adds a few reactions to the model. The model files `stat_step2_recruit.bngl` through `stat_step6_feedback.bngl` are cumulative, and lines added at each step are marked # new. Below we show only the new lines.

1.5.1 Step 2: STAT binds the ligand-bound receptor



STAT binds the ligand-bound receptor LR but never free R. Since the model contains no reaction in which free receptor recruits STAT, ligand binding is required for STAT recruitment. Every new species needs a molecule type, and species present at the start also need an initial amount.

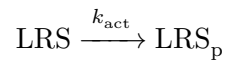
```

begin parameters
  S0          10      # STAT (uM)
  kSon        1       # STAT binding (1/uM/min)
  kSoff       1       # STAT unbinding (1/min)
end parameters
begin molecule types
  S
  LRS
end molecule types
begin species
  S      S0
end species
begin observables
  Molecules Sfree  S
  Molecules LRS    LRS
end observables
begin reaction rules
  # STAT binds ligand-bound receptor
  Sbind: LR + S <-> LRS kSon, kSoff

```

```
end reaction rules
```

1.5.2 Step 3: Receptor-bound STAT is activated



The receptor-associated kinase phosphorylates the docked STAT. This is a first-order, irreversible reaction with rate $v = k_{\text{act}}[\text{LRS}]$.

```
begin parameters
  kact      1      # STAT activation (1/min)
end parameters
begin molecule types
  LRSp
end molecule types
begin observables
  Molecules LRSp      LRSp
  Molecules pSTAT    LRSp
end observables
begin reaction rules
  # Receptor-associated kinase activates STAT
  Sact: LRS -> LRSp kact
end reaction rules
```

Consider what this model implies. Nothing in it releases activated STAT from the receptor, so one would expect every receptor to end up trapped as LRSp, having activated a single STAT. The simulation confirms this expectation (Figure 1.3).

```
model = bngsim.Model.from_bngl(model_path("ch01", "stat_step3_activate"))
sim = bngsim.Simulator(model, method="ode")
model.reset()
r = sim.run(t_span=(0, 15), n_points=301)
for name, label in [("LR", "LR"), ("LRS", "LRS"), ("LRSp", "LRSp_p$")]:
    plt.plot(r.time, r.observables[name], label=label)
plt.xlabel("time (min)")
plt.ylabel("concentration (μM)")
plt.legend();
```

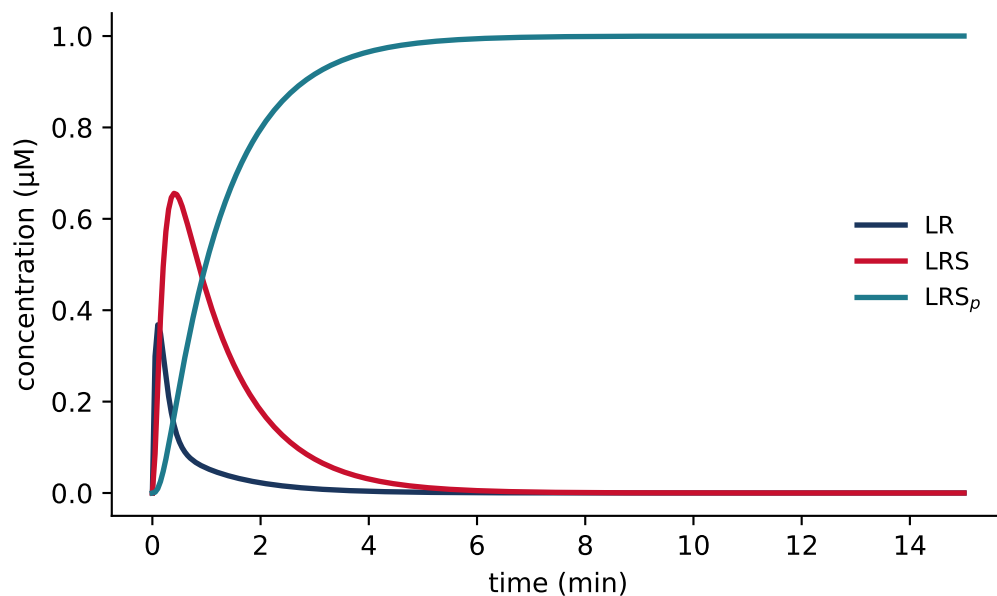
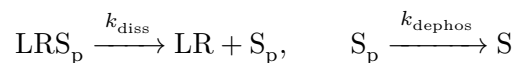


Figure 1.3: With activation but no release (step 3), every receptor ends up as LRS_p .

1.5.3 Step 4: Activated STAT is released and deactivated



Releasing pSTAT frees the receptor to activate another STAT, so one receptor can now activate many. A phosphatase returns free pSTAT to the inactive pool.

```
begin parameters
  kdiss      1      # pSTAT release (1/min)
  kdephos    0.1    # pSTAT dephosphorylation (1/min)
end parameters
begin molecule types
  Sp
end molecule types
begin observables
  Molecules Sp      Sp
  Molecules pSTAT   LRSp Sp
end observables
begin reaction rules
  # Activated STAT dissociates
  Srelease: LRSp -> LR + Sp kdiss
  # Phosphatase deactivates free STAT
  Sdephos: Sp -> S kdephos
end reaction rules
```

Total pSTAT now rises to a **steady state** set by the balance between activation and dephosphorylation (Figure 1.4). A steady state differs from the binding equilibrium above. At equilibrium every

reaction is balanced by its reverse, whereas here STAT cycles continuously through phosphorylation, release, and dephosphorylation, and the amounts stay constant only because these irreversible steps run at equal rates. Maintaining such a cycle requires a source of energy, which in cells is the ATP consumed by the kinase. The observable `pSTAT` sums all species that contain phosphorylated STAT. Its definition lists them by name, `LRSp Sp`, a chore we return to in Chapter 2.

```
model = bngsim.Model.from_bngl(model_path("ch01", "stat_step4_release"))
sim = bngsim.Simulator(model, method="ode")
model.reset()
r = sim.run(t_span=(0, 30), n_points=301)
plt.plot(r.time, r.observables["pSTAT"], label="total pSTAT")
plt.plot(r.time, r.observables["Sfree"], label="inactive STAT")
plt.xlabel("time (min)")
plt.ylabel("concentration (μM)")
plt.legend();
```

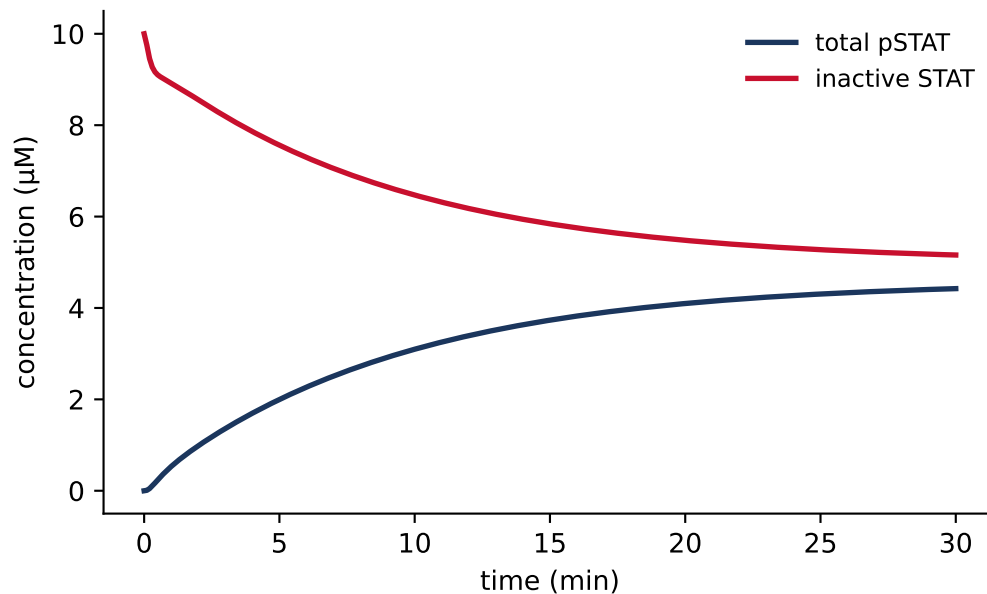


Figure 1.4: With release and dephosphorylation (step 4), pSTAT reaches a steady state.

1.5.4 Step 5: pSTAT induces SOCS expression

In step 5, pSTAT binds the SOCS promoter (G), the bound promoter is transcribed into SOCS mRNA, and the mRNA is translated into SOCS protein. Both mRNA and protein are degraded.

```
begin parameters
  G0      1      # SOCS gene promoter (uM)
  kGon    1      # pSTAT-promoter binding (1/uM/min)
  kGoff   1      # pSTAT-promoter unbinding (1/min)
```

```

    ktx      1      # transcription (1/min)
    kmdeg    0.1    # mRNA degradation (1/min)
    ktl      0.1    # translation (1/min)
    kSOCSdeg 0.05   # SOCS degradation (1/min)
end parameters
begin molecule types
    G
    GSp
    mSOCS
    SOCS
end molecule types
begin species
    G    GO
end species
begin observables
    Molecules mSOCS    mSOCS
    Molecules SOCS     SOCS
    Molecules pSTAT    LRSp Sp GSp
end observables
begin reaction rules
    # pSTAT binds the SOCS promoter
    Gbind: Sp + G <-> GSp kGon, kGoff
    # Bound promoter is transcribed
    Transcribe: GSp -> GSp + mSOCS ktx
    # mRNA degradation
    mRNAdeg: mSOCS -> 0 kmdeg
    # Translation
    Translate: mSOCS -> mSOCS + SOCS ktl
    # SOCS protein degradation
    SOCSdeg: SOCS -> 0 kSOCSdeg
end reaction rules

```

This step introduces synthesis and degradation reactions. In a **synthesis** reaction such as $GSp \rightarrow GSp + mSOCS$, a species appears on both sides; it acts as a template and is not consumed, so the rate is proportional to its amount. In a **degradation** reaction such as $SOCS \rightarrow 0$, the product 0 indicates that the molecule is removed. The rate constants of these reactions are first order.

SOCS now accumulates, but it does not yet act on anything, so pSTAT is essentially unchanged (Figure 1.5).

```

model = bngsim.Model.from_bngl(model_path("ch01", "stat_step5_transcribe"))
sim = bngsim.Simulator(model, method="ode")
model.reset()
r = sim.run(t_span=(0, 120), n_points=241)
for name, label in [("pSTAT", "pSTAT"), ("mSOCS", "SOCS mRNA"), ("SOCS",
    ↪ "SOCS protein")]:
    plt.plot(r.time, r.observables[name], label=label)

```

```
plt.xlabel("time (min)")
plt.ylabel("concentration (μM)")
plt.legend();
```

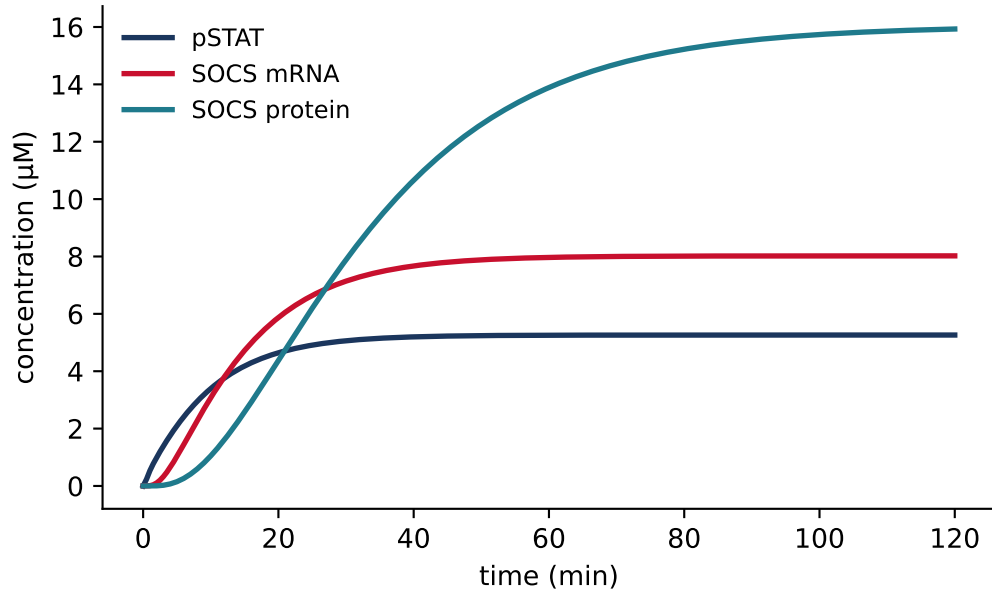
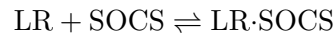
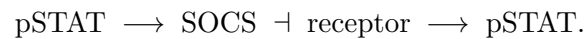


Figure 1.5: With SOCS expression but no feedback (step 5), SOCS mRNA and protein accumulate while pSTAT is unaffected.

1.5.5 Step 6: SOCS closes the feedback loop



SOCS competes with STAT for the ligand-bound receptor. The complex $\text{LR} \cdot \text{SOCS}$ cannot recruit STAT, so as SOCS accumulates, pSTAT production falls. This reaction completes the feedback loop:



```
begin parameters
  kSOCSon 1 # SOCS-receptor binding (1/uM/min)
  kSOCSoff 0.01 # SOCS-receptor unbinding (1/min)
end parameters
begin molecule types
  LRSOCS
end molecule types
begin observables
  Molecules LRSOCS LRSOCS
```

```

end observables
begin reaction rules
  # SOCS binds ligand-bound receptor, blocking STAT binding
  SOCSbind: LR + SOCS <-> LRSOCS kSOCSon, kSOCSoff
end reaction rules

```

1.6 Negative feedback turns a sustained signal into a pulse

Figure 1.6 compares the full model with the same model when SOCS cannot bind the receptor ($k_{\text{SOCSon}} = 0$). The model file runs both simulations itself, using the action `setParameter` between them. Here we do the same from Python.

```

model = bngsim.Model.from_bngl(model_path("ch01", "stat_step6_feedback"))
sim = bngsim.Simulator(model, method="ode")

for kSOCSon, label in [(0, "without feedback ( $k_{\text{SOCSon}} = 0$ )"),
                       (1, "with SOCS feedback")]:
    model.set_param("kSOCSon", kSOCSon)
    model.reset()
    r = sim.run(t_span=(0, 120), n_points=1201)
    plt.plot(r.time, r.observables["pSTAT"], label=label)
plt.xlabel("time (min)")
plt.ylabel("total pSTAT ( $\mu\text{M}$ )")
plt.ylim(0, 6)
plt.legend(loc="lower right");

```

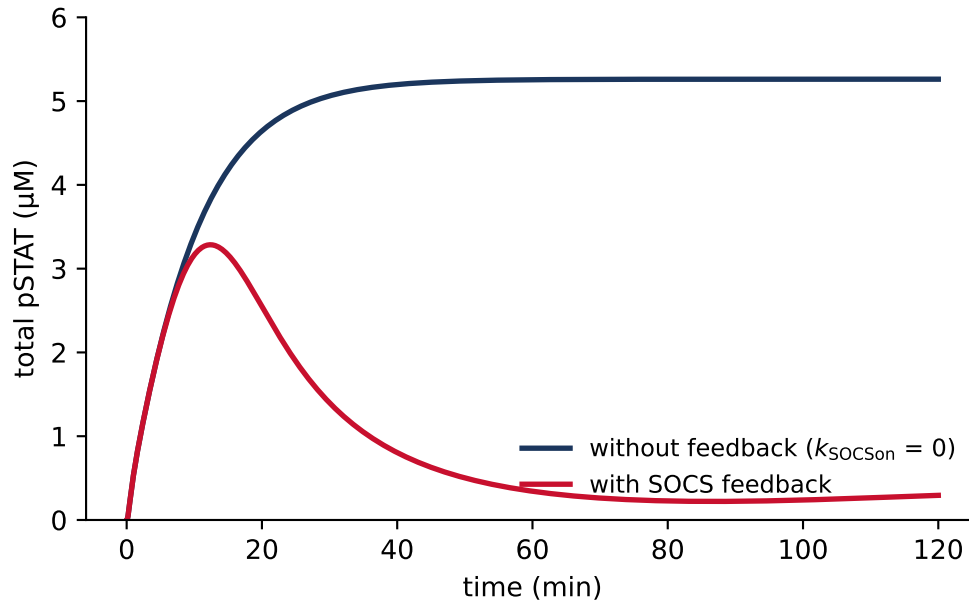


Figure 1.6: Negative feedback turns a sustained response into a pulse. Without feedback, pSTAT rises to a high plateau. With feedback, it peaks and then falls to a much lower adapted level, even though ligand is still present.

Following the addition of ligand, receptors bind ligand within seconds and begin to activate STAT, and pSTAT rises along the same curve whether or not SOCS can act, since no SOCS has yet been made. In the full model, SOCS accumulates over the next 10 to 20 minutes and binds the active receptors, and pSTAT falls to a low level even though ligand is still present, a behavior called **adaptation**. The delay before the feedback acts comes from the intervening steps, in which pSTAT must bind the promoter, mRNA must accumulate, and SOCS protein must build up to levels comparable to the receptor.

The pulse can be described by three properties: the **time of the peak**, the **height of the peak**, and the **adapted level** that pSTAT settles to. Each depends on the parameters in a different way. The exercises at the end of the chapter ask you to predict these dependencies, and a later chapter on model analysis develops tools to study them systematically.

1.7 The complete network and its limits

Table 1.1 lists the complete model. Eleven reaction lines in BNGL define 15 reactions among 12 species, since four of the reactions are reversible. BioNetGen generates the stoichiometry matrix and the 12 ODEs from this list, and writes the network to a `.net` file that you can inspect.

Table 1.1: The cytokine–STAT–SOCS reaction network.

Step	Reaction	Rate constants
1	$L + R \rightleftharpoons LR$	$k_{\text{on}}, k_{\text{off}}$
2	$LR + S \rightleftharpoons LRS$	$k_{\text{Son}}, k_{\text{Soff}}$

Step	Reaction	Rate constants
3	$\text{LRS} \rightarrow \text{LRS}_p$	k_{act}
4	$\text{LRS}_p \rightarrow \text{LR} + \text{S}_p$	k_{diss}
	$\text{S}_p \rightarrow \text{S}$	k_{dephos}
5	$\text{S}_p + \text{G} \rightleftharpoons \text{GS}_p$	$k_{\text{Gon}}, k_{\text{Goff}}$
	$\text{GS}_p \rightarrow \text{GS}_p + \text{mSOCS}$	k_{tx}
	$\text{mSOCS} \rightarrow 0$	k_{mdeg}
	$\text{mSOCS} \rightarrow \text{mSOCS} + \text{SOCS}$	k_{tl}
	$\text{SOCS} \rightarrow 0$	k_{SOCSdeg}
6	$\text{LR} + \text{SOCS} \rightleftharpoons \text{LR} \cdot \text{SOCS}$	$k_{\text{SOCSon}}, k_{\text{SOCSoff}}$

The model makes strong simplifying assumptions, and relaxing them exposes two limitations of the approach in this chapter.

First, every species had to be named by hand: L, LR, LRS, LRS_p , $\text{LR} \cdot \text{SOCS}$, and so on. Suppose SOCS could also bind receptors that already carry STAT, or ligand could dissociate from any receptor complex. Each such change creates new complexes, and each complex needs its own name and its own copy of every reaction it takes part in. For real signaling proteins, with many binding sites and modification sites, the number of possible complexes grows combinatorially. Chapter 2 introduces **rules** that act on structured molecules, so that the network is generated automatically.

Second, ODEs treat concentrations as continuous and deterministic. A cell may contain only one or two copies of the SOCS gene and a handful of mRNA molecules. At such small numbers the random timing of individual reactions matters, and genetically identical cells respond differently. The same reaction network can be simulated stochastically, which is the subject of Chapter 3.

1.8 Summary

- A reaction network model consists of species, reactions, rate laws, and initial conditions.
- Mass-action rate laws turn the network into ODEs, which have the general form $d\mathbf{X}/dt = \mathbf{S}\mathbf{v}(\mathbf{X})$.
- Conservation laws reduce the number of independent equations. Simple models can be solved analytically; realistic ones are integrated numerically with stiff solvers.
- BNGL describes the network directly. BioNetGen and bngsim generate and integrate the equations.
- Building the cytokine–STAT–SOCS circuit one step at a time shows what each mechanism contributes. Negative feedback through SOCS turns a sustained response into an adaptive pulse.

1.9 Exercises

These exercises are also available as a printable worksheet in the [course repository](#).

1. **Binding equilibrium.** Using `stat_step1_binding.bngl`, compute $[LR]_{eq}$ from Equation 1.3 for $L_0 = 0.1, 1, \text{ and } 10 \mu\text{M}$, and compare with simulations run to $t = 20 \text{ min}$. Above roughly what ligand concentration are almost all receptors bound?
2. **Tuning the pulse.** For each of `L0`, `kact`, `kdis`, `kdephos`, `ktl`, and `kSOCSdeg`, predict whether doubling the parameter raises, lowers, or leaves unchanged the time of the peak, the peak height, and the adapted level of pSTAT. Then test your predictions with `stat_step6_feedback.bngl`, simulating to $t = 600 \text{ min}$ so that pSTAT settles. Which parameter changes only the adapted level, and why? Why does doubling `L0` have almost no effect, and does it matter at $L_0 = 0.1 \mu\text{M}$?
3. **No phosphatase.** In `stat_step4_release.bngl`, set `kdephos` to 0. Predict, then simulate, the long-time value of total pSTAT. What sets this value?
4. **Toward rules.** Suppose SOCS can also bind receptors that carry unphosphorylated STAT (LRS). Write the new reaction in the notation of this chapter, and list the other reactions that the new complex needs. How many new species would you need if SOCS could also bind LRS_p ?

1.10 Further reading

Faeder et al. (2009) is a thorough introduction to BNGL and the BioNetGen software. Sekar and Faeder (2012) is a self-contained tutorial on rule-based modeling of signal transduction. Harris et al. (2016) describes features added in BioNetGen 2.2.

2 Molecules, patterns, and rules

In Chapter 1 every species in the cytokine–STAT–SOCS model had a name, such as L, LR, LRS, LRS_p , and $\text{LR} \cdot \text{SOCS}$. Naming species by hand works for twelve species but fails once the model includes what is known about the molecules involved, because signaling proteins have many binding and modification sites and the number of ways those sites can be combined grows combinatorially. Rule-based modeling, developed in the early 2000s to address this problem (Blinov et al. 2004; Hlavacek et al. 2006), represents molecules as structured objects with sites and states, and writes each interaction as a **rule** that mentions only the parts of the molecules that matter. BioNetGen then generates the reaction network from the rules.

Learning goals

After this chapter you should be able to:

- explain why multi-site proteins lead to combinatorial complexity;
- represent molecules with components, states, and bonds in BNGL;
- read and write patterns, and use them to define observables;
- write reaction rules and predict which reactions a rule generates;
- read a contact map and a rule diagram, and distinguish what a rule changes from its context;
- describe how BioNetGen generates a network from rules, and when network-free simulation is needed instead.

2.1 Signaling proteins have many interaction sites

Figure 2.1 shows the domain structure of STAT3. Its SH2 domain binds phosphotyrosines on activated receptors. Tyrosine 705, near the C terminus, is phosphorylated by JAK, and the phosphorylated tyrosine of one STAT binds the SH2 domain of another to form an active dimer. Serine 727, in the transactivation domain, is phosphorylated separately and modulates transcriptional activity. The N-terminal and coiled-coil regions mediate further interactions and nuclear transport, and the DNA-binding domain binds promoters.

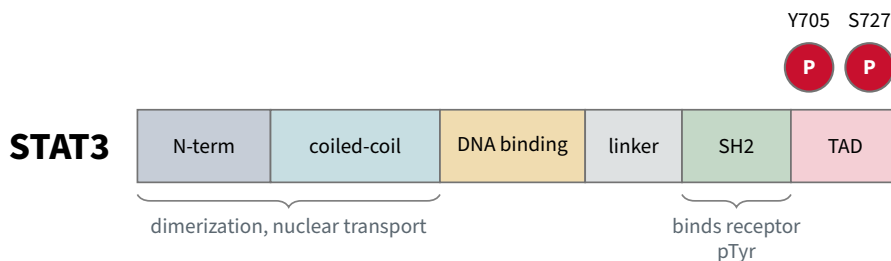


Figure 2.1: Domain structure of STAT3. Each domain or modification site is a separate place where the protein can interact or change state.

The receptor and JAK have their own sets of sites, as does nearly every signaling protein, and these sites can be occupied or modified in combination. Consider a protein with n sites, each of which can be modified independently. Such a protein has 2^n possible states, and since a complex combines the states of all its parts, the number of possible complexes grows faster still. This growth is called **combinatorial complexity**. In a model of the ErbB receptor family with its known binding partners, a single ErbB1 homodimer can take more than 5×10^{10} distinct states, and an ErbB1–ErbB3 heterodimer more than 3.8×10^9 , far more than could be named individually (Creamer et al. 2012).

However, each interaction is often thought to depend on only a few sites. In our model, for example, we assume that whether SOCS binds the receptor does not depend on whether a STAT molecule elsewhere in the complex is phosphorylated on serine 727. Rule-based modeling takes advantage of such locality by writing each interaction once, mentioning only the sites it depends on, so that the interaction applies to every complex that has those sites in the required states. The ability to represent sites separately does not mean that they behave independently, however. Locality is an assumption of the model, which can be tested, and where one site affects another, for example through an allosteric change, the rule must include that site in its context.

2.2 Molecules with structure

In BNGL a **molecule** is a named object with a list of **components** in parentheses. Components represent binding sites or modification sites. A component can have a set of allowed **states**, written after ~. The STAT molecule in our model is declared as

```
begin molecule types
  S(r,Y~0~P,g)
end molecule types
```

It has a site **r** that binds the receptor, a tyrosine **Y** that is either unphosphorylated (0) or phosphorylated (P), and a site **g** that binds DNA (Figure 2.2). A component can have both a state and a bond. A molecule with no components, such as **mSOCS()**, is just a species name, as in Chapter 1.

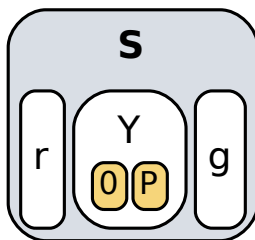


Figure 2.2: The STAT molecule type $S(r, Y \sim O \sim P, g)$, drawn from the model file. The molecule is gray, its components white, and the allowed states of Y yellow.

2.3 Complexes are graphs

Molecules join into complexes through **bonds** between components. The species LRS_p from Chapter 1, a ligand-bound receptor carrying phosphorylated STAT, is written

```
L(r!1).R(l!1,s!2).S(r!2,Y~P,g)
```

The dot (.) joins molecules in the same complex. A bond is written as ! followed by a label on each of its two ends: !1 on the ligand's r and on the receptor's l says those two components are bound to each other. Bond labels only need to be unique within a species. A component with no !, like g here, is **free**.

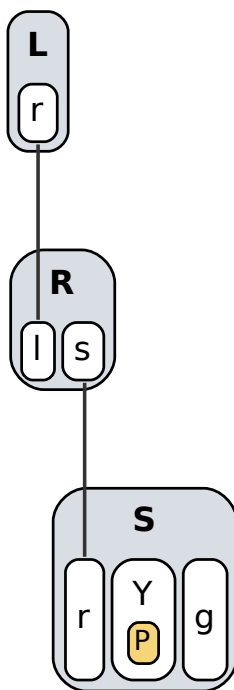


Figure 2.3: The complex $L(r!1).R(l!1,s!2).S(r!2,Y\sim P,g)$. Each molecule contains its components, the yellow box gives the state of Y , and lines are bonds between components.

A species is therefore a **graph** (Figure 2.3). Many strings describe the same graph: molecules can be listed in any order and bonds numbered in any way. BioNetGen recognizes when two strings describe the same species by comparing their graphs.

2.4 Patterns

A **pattern** describes a set of species by listing only the parts that matter. The guiding principle is **don't write, don't care**: any component not mentioned in a pattern may be in any state and bound or not. Within a pattern:

- a component with no bond notation must be **free**;
- $!+$ means bound to any partner;
- $!?$ means bound or not;
- a state such as $\sim P$ must match exactly.

A pattern matches a species if it can be found anywhere in the species' graph. Table 2.1 shows which species from Chapter 1 match several patterns.

Table 2.1: Patterns and the species they match.

Pattern	LR	LRS	LRS _p	S _p	GS _p	LR · SOCS	Meaning
S(Y~P)			✓	✓	✓		any phosphorylated STAT
S(r,Y~P)				✓	✓		phosphorylated, not on a receptor
S(r!+)		✓	✓				STAT bound through r
R(l!+)	✓	✓	✓			✓	ligand-bound receptor
R(l!+,s)	✓						... with its docking site free
R(s!+)		✓	✓			✓	docking site occupied

2.4.1 Observables are patterns

Observables are defined by patterns, which solves the bookkeeping problem we met in Chapter 1. There, total pSTAT had to be defined by listing every species that contains it:

```
Molecules pSTAT LRSp Sp GSp
```

With structured molecules we describe the property instead:

```
Molecules pSTAT S(Y~P)
```

The pattern continues to match every pSTAT-containing species as new complexes are added to the model, so the observable never needs to be rewritten.

A **Molecules** observable counts every match of its pattern, so a STAT dimer with two phosphotyrosines contributes 2 to S(Y~P), whereas a **Species** observable counts each matching complex once. Anticipating the STAT dimers introduced later in this chapter, in which STAT gains a dimerization site d, the observable

```
Species pSTATdimer S(Y~P,d!1).S(Y~P,d!1)
```

counts complexes that contain a dimer of two phosphorylated STATs. A complex that contained two such dimers would still count once.

2.5 Rules

A **reaction rule** has the same form as a reaction, but its reactants and products are patterns. Here is the rule for STAT recruitment:

```
Sbind: R(l!+,s) + S(r,Y~0) <-> R(l!+,s!1).S(r!1,Y~0) kSon, kSoff
```

- The **reactant patterns** on the left say what must be present: a ligand-bound receptor whose docking site s is free, and an unphosphorylated STAT whose site r is free.

- The **product pattern** on the right says what changes: a bond forms between **s** and **r**.
- Everything not mentioned is carried along unchanged. If the receptor were part of a larger complex, the whole complex would take part.
- Each reaction generated by the rule gets the rule's rate constant, multiplied by any statistical factor that arises from identical molecules or sites (Chapter 4), and its rate follows mass action.

The parts of a reactant pattern that are not changed by the rule, here **l!+** and the state **Y~0**, are its **context**, which restricts where the rule applies. A rule can perform five kinds of operation:

Operation	Example
form a bond	$A(b) + B(a) \rightarrow A(b!1).B(a!1)$
break a bond	$A(b!1).B(a!1) \rightarrow A(b) + B(a)$
change a state	$S(Y\sim 0) \rightarrow S(Y\sim P)$
create a molecule	$G(s!+) \rightarrow G(s!+) + mSOCS()$
delete a molecule	$SOCS(r) \rightarrow 0$

2.5.1 Rule diagrams

A rule can also be drawn. In a **rule diagram**, the reactant patterns are drawn as graphs, and each operation the rule performs is a separate node with arrows to the parts of the patterns that it changes (Sekar et al. 2017). Everything else in the diagram is context. Figure 2.4 shows the diagrams that BioNetGen generates for **Sbind** and for the rule that phosphorylates STAT on the receptor,

Sact: $R(s!1).S(r!1,Y\sim 0) \rightarrow R(s!1).S(r!1,Y\sim P)$ **kact**

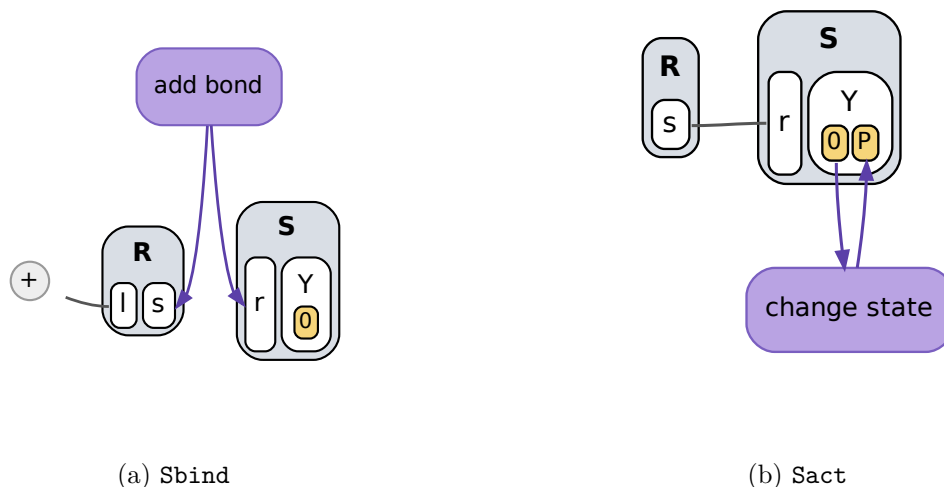


Figure 2.4: Rule diagrams generated from `stat_rbm1_rules.bngl`. Molecules are gray, components white, and internal states yellow. Purple nodes are the operations each rule performs, with arrows to the components or states they change. Gray lines and the + node are context.

Molecules are drawn as gray boxes, their components as white boxes inside them, and internal states as yellow boxes inside the components. In **Sbind** the operation forms a bond between the docking site **s** of the receptor and the site **r** of STAT. The rest is context. The + attached to **l** is a bond to an unspecified partner, written **l!+** in the rule, and the yellow **0** requires unphosphorylated STAT. In **Sact** the operation changes the state of **Y** from **0** to **P**, and the gray line is a bond between receptor and STAT that must already exist. A component drawn without a line must be free, whereas a component that the rule does not mention, such as the site **g** of STAT, is not drawn at all, since it does not affect whether the rule applies. Reading a rule as a diagram thus separates the question of what a rule does from the question of where it applies.

2.6 The Chapter 1 model as rules

We now rewrite the model of Chapter 1 with structured molecules. Six molecule types replace twelve species names, and **seed species** list the complexes present at the start. Every other species will be generated from these by the rules.

```
begin molecule types
  L(r)                                # ligand with one receptor-binding site
  R(l,s)                              # receptor chain: ligand site l, docking site s
  ⇐ (shared by STAT and SOCS)
  S(r,Y~0~P,g)                       # STAT: receptor site r, tyrosine Y
  ⇐ (unphosphorylated/phosphorylated), DNA site g
  G(s)                                # SOCS gene promoter with a STAT-binding site
```

```

mSOCS()                # SOCS mRNA (no internal structure)
SOCS(r)                 # SOCS protein with a receptor-binding site
end molecule types
begin seed species
  L(r)                  L0
  R(l,s)                R0
  S(r,Y~0,g)           S0
  G(s)                  G0
end seed species
begin observables
  Molecules pSTAT      S(Y~P)
  Molecules LRtot      L(r!1).R(l!1)
  Molecules mSOCS      mSOCS()
  Molecules SOCS       SOCS()
end observables

```

A **contact map** summarizes the molecule types of a model and the bonds that its rules can form (Figure 2.5). BioNetGen draws one from a model with the action `visualize({type=>"contactmap"})`. Each molecule type appears once, with its components and their internal states, and each line joins two components that some rule can bind. The SOCS mRNA, which has no components, appears as a box with no lines. A contact map shows which interactions are possible, but not the conditions under which they occur. For example, it shows that STAT and SOCS both bind the docking site `s`, but not that they bind only receptors that carry ligand. That information is the context of the rules.

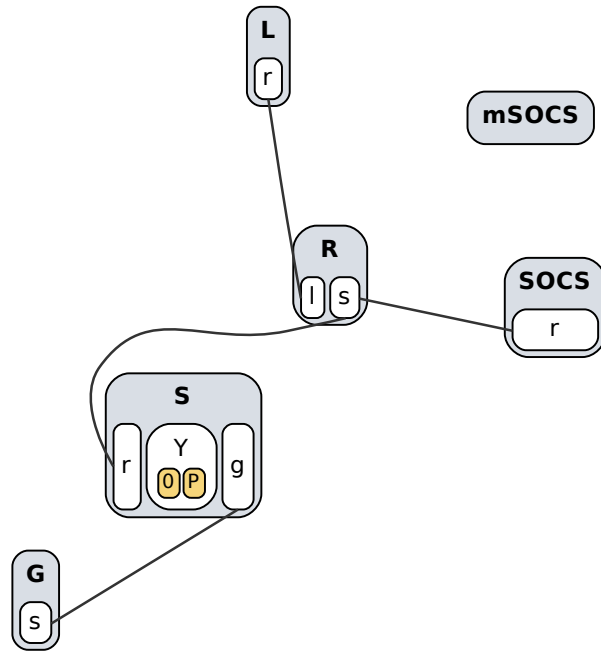


Figure 2.5: Contact map of `stat_rbm1_rules.bngl`. Each line joins two components that a rule of the model can bind.

The receptor has a single docking site `s` that STAT and SOCS compete for, which is how the model of Chapter 1 treats SOCS. The rules are:

```
begin reaction rules
# Ligand binds a free receptor
LRbind:    L(r) + R(l,s) <-> L(r!1).R(l!1,s)           kon, koff
# STAT docks on a ligand-bound receptor
Sbind:     R(l!+,s) + S(r,Y~0) <-> R(l!+,s!1).S(r!1,Y~0) kSon, kSoff
# Receptor-associated kinase activates docked STAT
Sact:      R(s!1).S(r!1,Y~0) -> R(s!1).S(r!1,Y~P)      kact
# Activated STAT dissociates
Srelease:  R(s!1).S(r!1,Y~P) -> R(s) + S(r,Y~P)        kdiss
# Phosphatase deactivates free monomeric pSTAT
Sdephos:   S(r,g,Y~P) -> S(r,g,Y~0)                    kdephos
# pSTAT binds the SOCS promoter
Gbind:     S(r,Y~P,g) + G(s) <-> S(r,Y~P,g!1).G(s!1)   kGon, kGoff
# Bound promoter is transcribed
Transcribe: G(s!+) -> G(s!+) + mSOCS()                 ktx
# mRNA degradation
mRNAddeg:  mSOCS() -> 0                                kmdeg
# Translation
```

```

Translate:  mSOCS() -> mSOCS() + SOCS(r)          ktl
# Free SOCS degradation
SOCSdeg:    SOCS(r) -> 0                          kSOCSdeg
# SOCS binds a ligand-bound receptor
SOCSbind:   R(l!+,s) + SOCS(r) <-> R(l!+,s!1).SOCS(r!1)  kSOCSon,
↪ kSOCSoff
end reaction rules

```

Context encodes the assumptions of the earlier model. $R(l,s)$ in `LRbind` says that ligand binds and leaves only when the docking site is empty, so ligand can leave LR but not LRS. $R(l!+,s)$ in `Sbind` and `SOCSbind` says that only ligand-bound receptors recruit STAT or SOCS.

2.6.1 Network generation

BioNetGen turns rules into a reaction network by **network generation**:

1. Start with the seed species.
2. Apply every rule to every species, or pair of species, that matches its reactant patterns.
3. Add any new species and reactions produced.
4. Repeat until nothing new appears.

The `generate_network` action does this and writes the result to a `.net` file. ODE and stochastic simulations then use the generated network. `bngsim`'s `Model.from_bngl` also generates the network, so we can check the result directly:

```

named = bngsim.Model.from_bngl(model_path("ch01", "stat_step6_feedback"))
rules = bngsim.Model.from_bngl(model_path("ch02", "stat_rbm1_rules"))
print("named species:", named.n_species, "species,", named.n_reactions,
↪ "reactions")
print("rule-based:   ", rules.n_species, "species,", rules.n_reactions,
↪ "reactions")
for name in rules.species_names:
    print("  ", name)

```

```

named species: 12 species, 15 reactions
rule-based:   12 species, 15 reactions
L(r)
R(l,s)
S(Y~0,g,r)
G(s)
L(r!1).R(l!1,s)
L(r!1).R(l!1,s!2).S(Y~0,g,r!2)
L(r!1).R(l!1,s!2).S(Y~P,g,r!2)
S(Y~P,g,r)
G(s!1).S(Y~P,g!1,r)
mSOCS()

```

```
SOCS(r)
L(r!1).R(l!1,s!2).SOCS(r!2)
```

The rules generate the same 12 species and 15 reactions as the hand-written network, now with structured names. Figure 2.6 confirms that the two models give identical dynamics.

```
for model, style, label in [(named, dict(color="#D39B1E", lw=6), "named
↪ species (Chapter 1)",
                                (rules, dict(color="#1B365D", lw=1.5), "rules
↪ (this chapter)"]):
    sim = bngsim.Simulator(model, method="ode")
    model.reset()
    r = sim.run(t_span=(0, 120), n_points=1201)
    plt.plot(r.time, r.observables["pSTAT"], label=label, **style)
plt.xlabel("time (min)")
plt.ylabel("total pSTAT (μM)")
plt.legend();
```

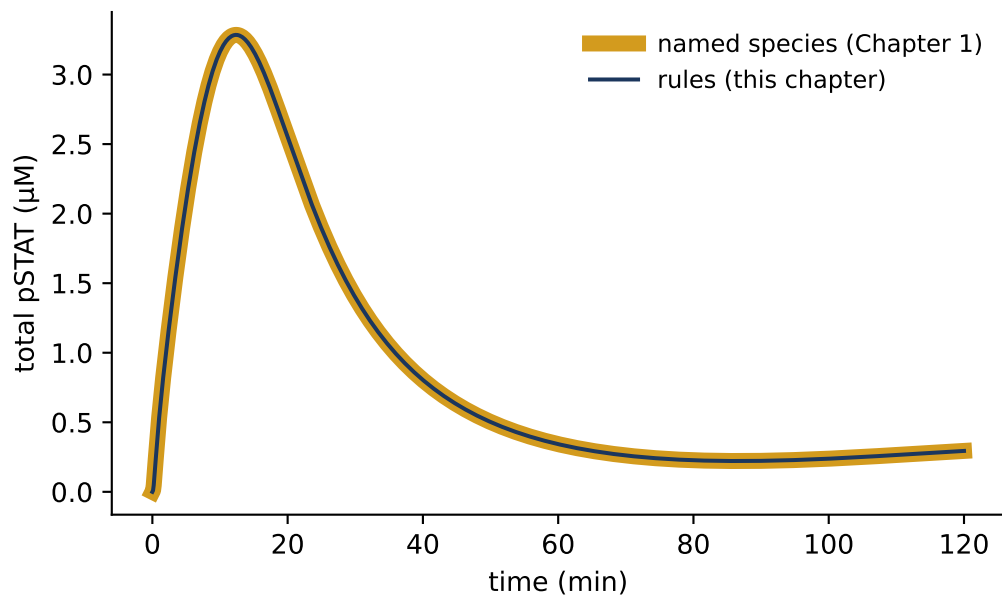


Figure 2.6: The rule-based model generates the same network as the model of Chapter 1, so their predictions are identical.

2.7 Relaxing the assumptions

We now make the model more realistic by relaxing its simplifying assumptions one at a time. Each version is a separate file in `models/ch02/`, and each changes only a few lines.

2.7.1 Ligand can leave any receptor complex

Removing the context `s` from the ligand rule lets ligand bind and unbind whatever the receptor's docking site is doing:

```
# Ligand binds a free receptor
LRbind:    L(r) + R(l) <-> L(r!1).R(l!1)          kon, koff
```

STAT docking and activation still require bound ligand, but STAT can undock from a receptor that has lost its ligand. A reversible rule applies the same context in both directions, so the binding and unbinding of STAT (and of SOCS) are now written as separate rules:

```
# STAT docks on a ligand-bound receptor
Sbind:     R(l!+,s) + S(r,Y~0) -> R(l!+,s!1).S(r!1,Y~0)    kSon
# Unphosphorylated STAT undocks
Sunbind:   R(s!1).S(r!1,Y~0) -> R(s) + S(r,Y~0)          kSoff
# Receptor-associated kinase activates docked STAT
Sact:      R(l!+,s!1).S(r!1,Y~0) -> R(l!+,s!1).S(r!1,Y~P)  kact
```

As a result, `LRbind` alone now generates four binding reactions, one each for ligand binding to `R` alone, to `R · S`, to `R · Sp`, and to `R · SOCS`, together with four matching unbinding reactions, none of which had to be written by hand.

2.7.2 SOCS binds its own site and blocks kinase activity

In the first model SOCS blocks STAT docking by occupying the same site. SOCS proteins act differently in cells, where SOCS1 inhibits JAK directly and SOCS3 inhibits JAK while bound to the receptor. We give the receptor a separate SOCS site and require it to be free for STAT activation:

```
R(l,s,socs)          # receptor chain: ligand site l, STAT docking
  ⇨ site s, SOCS site socs
R(l,s,socs)    R0
# Receptor-associated kinase activates docked STAT
Sact:          R(l!+,socs,s!1).S(r!1,Y~0) -> R(l!+,socs,s!1).S(r!1,Y~P)  kact
# SOCS binds a ligand-bound receptor
SOCSbind:      R(l!+,socs) + SOCS(r) -> R(l!+,socs!1).SOCS(r!1)    kSOCSon
# SOCS unbinds
SOCSunbind:    R(socs!1).SOCS(r!1) -> R(socs) + SOCS(r)          kSOCSoff
```

The free `socs` component in `Sact` means that activation happens only when SOCS is absent. A receptor can now carry STAT and SOCS at the same time.

2.7.3 Cytokines bring two receptor chains together

Many cytokine receptors signal when the cytokine brings together two receptor chains whose JAKs activate each other. The chains are often of different types, some receptors are assembled before the

cytokine binds, and some cytokines assemble larger complexes. We take the simplest case, in which a ligand with two identical sites bridges two identical chains, and require the dimer for activation (Figure 2.7):

```

L(r,r)                                # bivalent ligand: two identical receptor-binding
↔ sites
L(r,r)      L0
# Ligand binds a free receptor
LRbind:     L(r,r) + R(l) <-> L(r!1,r).R(l!1)           kon, koff
# Ligand already bound to one receptor captures a second (receptor dimer)
Crosslink:  L(r!+,r) + R(l) <-> L(r!+,r!1).R(l!1)       kx, koff
# Receptor-associated kinase activates docked STAT
Sact:       L(r!+,r!2).R(l!2,socs,s!1).S(r!1,Y~0) ->
↔ L(r!+,r!2).R(l!2,socs,s!1).S(r!1,Y~P)    kact

```

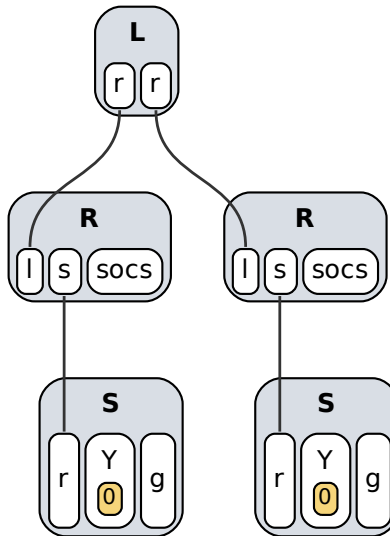


Figure 2.7: A cytokine bridging two receptor chains, each carrying an unphosphorylated STAT, one of the species generated from `stat_rbm4_receptor_dimer.bngl`.

BioNetGen accounts automatically for the two identical ligand sites. A free ligand can bind a receptor through either site, so the generated reaction has a statistical factor of 2. In addition, a bivalent ligand in large excess occupies receptor chains one at a time and prevents crosslinking, so the response falls at high dose. This version therefore lowers L_0 from 10 to 1 μM .

2.7.4 STATs act as dimers

Phosphorylated STATs form dimers, and it is the dimer that binds DNA. We add a dimerization site and let free pSTAT dimerize. Only dimers bind the SOCS promoter, and only monomers are dephosphorylated:

```
S(r,Y~0~P,d,g)          # STAT: receptor site r, tyrosine Y, dimerization
  ⇨ site d, DNA site g
# Phosphatase deactivates free monomeric pSTAT
Sdephos:    S(r,d,g,Y~P) -> S(r,d,g,Y~0)          kdephos
# Free pSTAT dimerizes
Sdimer:     S(r,Y~P,d) + S(r,Y~P,d) <-> S(r,Y~P,d!1).S(r,Y~P,d!1)    kdim,
  ⇨ kundim
# pSTAT binds the SOCS promoter
Gbind:      S(d!+,g) + G(s) <-> S(d!+,g!1).G(s!1)          kGon, kGoff
```

$S(d!+,g)$ matches any STAT that is in a dimer and not yet on DNA. We restricted dimerization to free pSTAT because, if receptor-bound STATs could also dimerize, dimers could bridge receptor complexes into chains of any length and the network would be infinite. Network generation would never finish unless limited, for example with the `max_agg` option, which caps the size of complexes.

2.7.5 A second phosphorylation site

STAT3 is also phosphorylated on serine 727. We treat this as a separate process that does not depend on the state of tyrosine 705, an assumption that could be changed by adding $Y\sim 0$ or $Y\sim P$ to the context of the rule. The change requires one new component and a single rule:

```
S(r,Y~0~P,S727~0~P,d,g)  # STAT: receptor site r, tyrosine Y, serine S727,
  ⇨ dimerization site d, DNA site g
# Ser727 is phosphorylated and dephosphorylated in any context
Sser:       S(S727~0) <-> S(S727~P)                kser,
  ⇨ kserdephos
```

The rule mentions only S727, so it applies to STAT in every complex: free, on the receptor, in a dimer, and on DNA. Every species containing one STAT now comes in two versions, and every species containing two STATs in three or four.

2.8 Rules grow linearly; networks grow combinatorially

Table 2.2 counts the rules in each version and the species and reactions that BioNetGen generates from them.

```
import pandas as pd
versions = [("stat_rbm1_rules", "Chapter 1 model as rules"),
```

```

        ("stat_rbm2_ligand_release", "ligand leaves any complex"),
        ("stat_rbm3_socs_site", "separate SOCS site"),
        ("stat_rbm4_receptor_dimer", "receptor dimers"),
        ("stat_rbm5_stat_dimer", "STAT dimers"),
        ("stat_rbm6_ser727", "Ser727 site")]
rows, models = [], {}
for i, (name, change) in enumerate(versions, start=1):
    m = bngsim.Model.from_bngl(model_path("ch02", name))
    models[i] = m
    rows.append({"version": i, "change": change, "rules": rule_count("ch02",
↪ name),
                "species": m.n_species, "reactions": m.n_reactions})
sizes = pd.DataFrame(rows).set_index("version")
sizes

```

Table 2.2: Rules in each model version and the size of the generated network.

	change	rules	species	reactions
version				
1	Chapter 1 model as rules	11	12	15
2	ligand leaves any complex	13	15	24
3	separate SOCS site	13	19	39
4	receptor dimers	14	40	188
5	STAT dimers	15	43	198
6	Ser727 site	16	95	663

```

for col, marker in [("reactions", "o"), ("species", "s"), ("rules", "^")]:
    plt.semilogy(sizes.index, sizes[col], marker=marker, label=col)
plt.xlabel("model version")
plt.ylabel("count")
plt.xticks(sizes.index)
plt.legend();

```

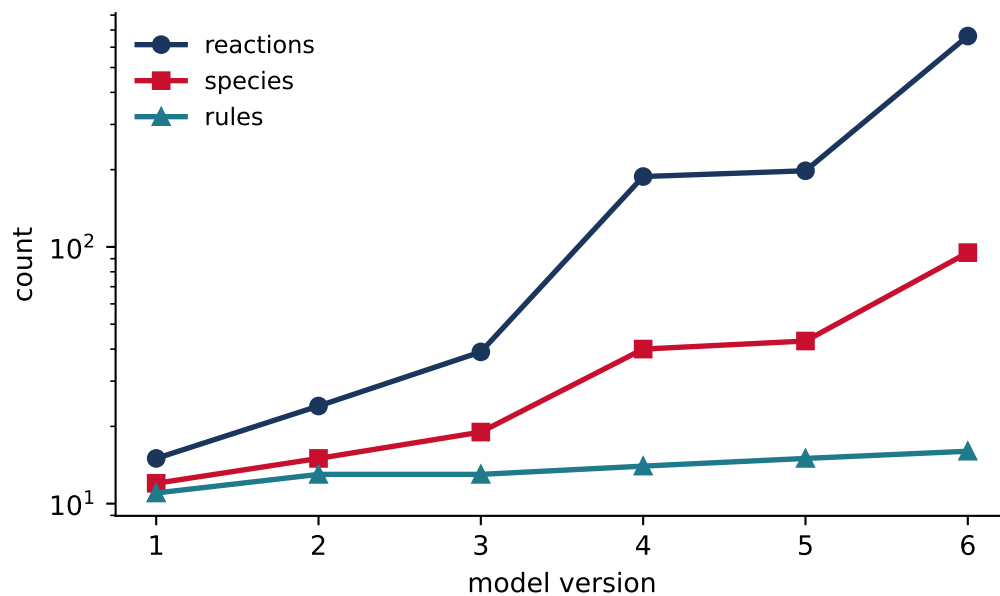


Figure 2.8: From 11 to 16 rules, the generated network grows from 15 to 663 reactions.

Five relaxed assumptions added five rules but grew the network more than fortyfold. Writing 663 reactions and the ODEs for 95 species by hand would be slow and error-prone. Rules let us write down the mechanism and leave the bookkeeping to the software.

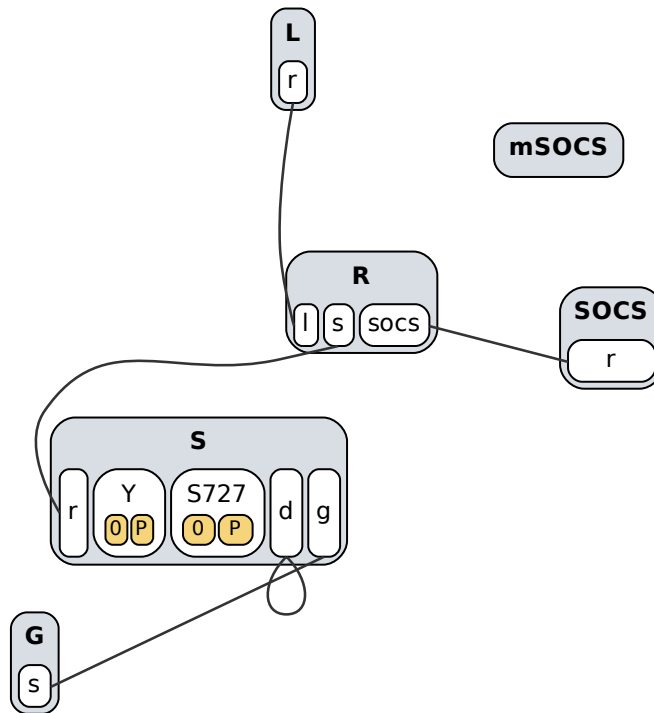


Figure 2.9: Contact map of `stat_rbm6_ser727.bng1`, the final version of the model.

The contact map grows in the same way as the rules. Figure 2.9 shows the contact map of version 6, which has 11 components and 5 possible bonds, compared with 8 and 4 in version 1 (Figure 2.5). The new components are the SOCS site on the receptor and the dimerization site and serine on STAT. The second site of the ligand adds nothing, since a contact map draws identical sites once. The loop on the site `d` of STAT is the bond between the two STATs of a dimer. The contact map remains small enough to read at a glance, while the network it implies does not.

2.9 Mechanistic details change the predictions

Each version embodies a different hypothesis about the mechanism, and the hypotheses make different predictions. Since the versions were built cumulatively, however, consecutive versions can differ in more than one way, and an effect can be attributed to a mechanism only by comparing models that differ in that mechanism alone. Versions 2 and 3 differ only in where SOCS binds. Versions 4 and 5 differ in two ways, since in version 5 dimerization protects pSTAT from the phosphatase and only dimers bind the SOCS promoter. To separate these, we construct a variant of version 5 in which pSTAT binds the promoter whether or not it is in a dimer, as in version 4 (Figure 2.10).

```

import tempfile, os
def run_pSTAT(m):
    sim = bngsim.Simulator(m, method="ode")
    m.reset()
    return sim.run(t_span=(0, 120), n_points=1201)

def half_decay(r):
    # time after the peak at which pSTAT first falls to half its peak value
    p, t = r.observables["pSTAT"], r.time
    i = p.argmax()
    return t[i + np.argmax(p[i:] < p[i] / 2)]

# variant of version 5: promoter binding as in version 4
src = model_path("ch02", "stat_rbm5_stat_dimer").read_text()
old = "Gbind:      S(d!+,g) + G(s) <-> S(d!+,g!1).G(s!1)"
assert old in src
with tempfile.TemporaryDirectory() as d:
    path = os.path.join(d, "v5_protection_only.bngl")
    open(path, "w").write(src.replace(old, "Gbind:      S(r,Y~P,g) + G(s)
        ↪ <-> S(r,Y~P,g!1).G(s!1)"))
    v5p = bngsim.Model.from_bngl(path, cache=False)

runs = {"2: SOCS blocks docking": run_pSTAT(models[2]),
        "3: SOCS blocks the kinase": run_pSTAT(models[3]),
        "4: receptor dimers": run_pSTAT(models[4]),
        "5: STAT dimers": run_pSTAT(models[5]),
        "5, protection only": run_pSTAT(v5p)}
tau = {k: half_decay(r) for k, r in runs.items()}
peak = {k: r.observables["pSTAT"].max() for k, r in runs.items()}

for keys in [list(runs)[:2], list(runs)[2:]]:
    fig, ax = plt.subplots(figsize=(4.2, 3.2))
    for k in keys:
        ax.plot(runs[k].time, runs[k].observables["pSTAT"], label=k)
    ax.set_xlabel("time (min)")
    ax.set_ylabel("total pSTAT (µM)")
    ax.legend()
    plt.show()

```

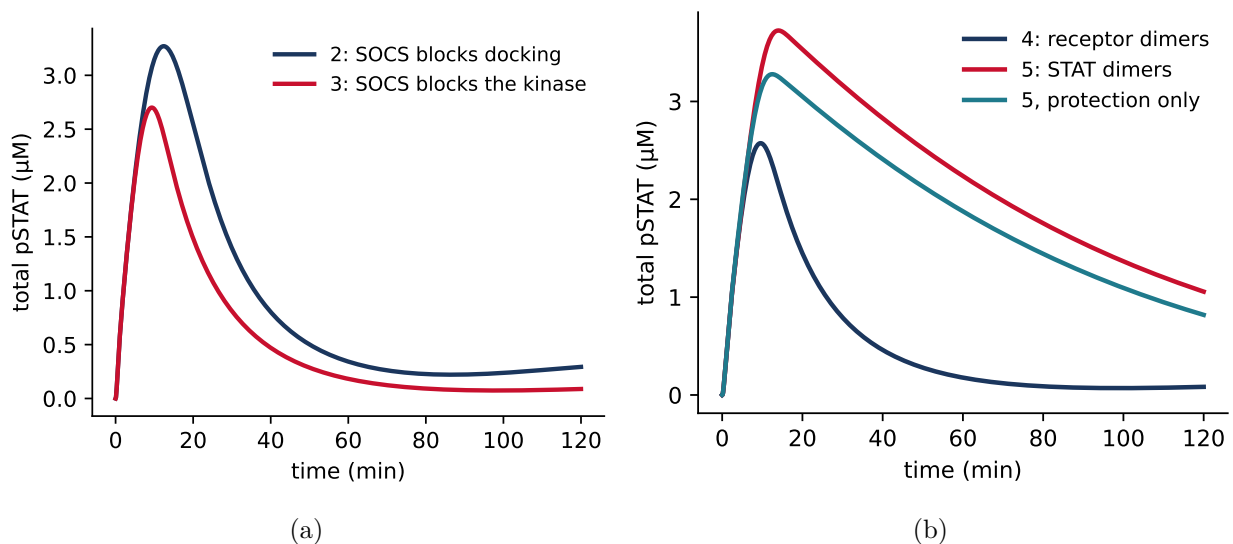


Figure 2.10: Controlled comparisons of mechanisms. (a) SOCS blocks STAT docking (version 2) or the kinase (version 3); the models are otherwise identical. (b) Version 4, version 5, and a variant of version 5 in which dimerization protects pSTAT from the phosphatase but monomers can still bind the SOCS promoter. All three have $L_0 = 1 \mu\text{M}$.

In the first comparison, blocking the kinase (version 3) gives a lower peak, 2.70 μM rather than 3.27 μM , and a faster shutoff than blocking docking (version 2). pSTAT falls to half its peak 22 min after stimulation rather than at 27 min. Since this is the only difference between the two models, the change can be attributed to the site at which SOCS acts. In the second comparison, pSTAT falls to half its peak at 22 min without dimers and at 75 min with them. Protection from the phosphatase alone delays this time to 70 min, so it accounts for most of the longer response, and the requirement that only dimers bind the promoter accounts for the rest. These comparisons demonstrate that mechanistic details that are easy to overlook can change the predicted response substantially. Deciding which mechanism operates in cells requires experiments, and rules make it easy to build the alternative models and compare their predictions, which is the first step in designing experiments that distinguish them.

2.10 When networks are too large to generate

Network generation works well when the network is small, but realistic models can imply far more species than can be listed. A rule-based model of T cell receptor signaling with 20 molecule types and 129 rules (Chylek et al. 2014), and the ErbB model mentioned above (Creamer et al. 2012), are too large for network generation. Some models, like the STAT model with dimerization on receptors, are infinite.

Network-free simulation avoids generating the network at all. It tracks individual molecules and their bonds, and applies rules to them as the simulation runs, so only the complexes that actually form are ever represented (Sneddon et al. 2011). The cost per simulated event depends on the number of rules and molecules rather than the size of the network. The trade-offs are that

network-free simulation is stochastic and works with molecule counts rather than concentrations. bngsim provides two network-free simulators, and a later chapter covers them in detail.

2.11 Summary

- Multi-site proteins create combinatorial complexity, in which a few interactions imply many species and reactions.
- BNGL represents molecules as objects with components, states, and bonds, so species are graphs.
- Patterns specify only what matters (don't write, don't care), and define observables that stay correct as a model grows.
- Rules transform patterns. One rule generates every reaction that fits its pattern, and context restricts where it applies.
- A contact map shows which components of a model can bind, and a rule diagram shows what a rule changes and the context in which it applies. BioNetGen generates both from the model.
- BioNetGen generates the network from rules for ODE and stochastic simulation. Network-free methods simulate models whose networks are too large to generate.

2.12 Exercises

1. In `stat_rbm1_rules.bngl`, change `R(1,s)` to `R(1)` in `LRbind`. Generate the network. Which new species appear, and why?
2. Write an observable for receptors that carry both STAT and SOCS in `stat_rbm3_socs_site.bngl`. Should it be a `Molecules` or a `Species` observable?
3. In `stat_rbm4_receptor_dimer.bngl`, vary `L0` from 0.01 to 100 μM and plot the peak of total pSTAT against `L0`. Explain the shape of the curve.
4. Remove the `r` context from `Sdimer` in `stat_rbm5_stat_dimer.bngl`, so that receptor-bound pSTAT can dimerize. What happens when you generate the network? Try adding `max_agg=>6` to `generate_network`.
5. Sketch by hand the rule diagrams for `SOCSbind` in `stat_rbm1_rules.bngl` and in `stat_rbm3_socs_site.bngl`. Then draw them with `rule_diagram` from `scripts/bookutils.py`, or with the action `visualize({type=>"ruleviz_operation"})`. Which parts of the diagrams differ, and what does the difference mean for the competition between STAT and SOCS?

2.13 Further reading

Hlavacek et al. (2006) reviews the motivation for rule-based modeling and its applications. Faeder et al. (2009) and Sekar and Faeder (2012) cover BNGL syntax and modeling practice in depth, including many examples of rules for common signaling mechanisms. Sneddon et al. (2011) introduces network-free simulation with NFsim. Sekar et al. (2017) describes the contact maps and rule diagrams that BioNetGen generates, and methods for visualizing larger models.

3 Stochastic simulation and noise

The differential equations of Chapter 1 treat the amount of each species as a continuous quantity that changes smoothly and predictably in time. This is a good description when every species is present in large numbers. A cell, however, typically contains one or two copies of a given gene, a few to a few hundred copies of each mRNA, and proteins in numbers ranging from tens to millions. At the low end of this range, the timing of individual reaction events matters, and genetically identical cells exposed to the same signal respond differently. In this chapter we describe a reaction network by the numbers of molecules it contains and by the probabilities of the reactions that change them, simulate it with the stochastic simulation algorithm, and use the simulations to ask how random fluctuations shape the response of the cytokine–STAT–SOCS circuit.

Learning goals

After this chapter you should be able to:

- convert a model written in concentrations into one written in molecule numbers;
- explain what a reaction propensity is and how it is related to a rate constant;
- describe Gillespie’s stochastic simulation algorithm and implement it for a simple reaction;
- explain how the size of fluctuations depends on the number of molecules;
- describe how bursts of transcription arise and how they propagate through a feedback loop;
- run stochastic simulations with `bngsim` and analyze ensembles of trajectories.

3.1 Why molecule numbers matter

The idea that biochemical noise can be measured, rather than merely supposed, was established by experiments in single cells. In 2002 Elowitz and coworkers placed two copies of the same promoter, driving two fluorescent proteins of different colors, into the same *Escherichia coli* cells (Elowitz et al. 2002). If expression were deterministic, each cell would contain equal amounts of the two proteins. Instead, the cells varied in two ways. The total amount of the two proteins differed from cell to cell, with the two colors rising and falling together, and within a single cell the two colors differed from each other. Elowitz and coworkers called the correlated part of the variation extrinsic noise, since it arises from differences among cells in factors that affect both genes, such as the amounts of polymerase and ribosomes. They called the uncorrelated part intrinsic noise, since it arises from the random timing of transcription and translation of each gene, and they found that both make substantial contributions to the variation among cells. Since then, cell-to-cell variation in gene expression has been found in organisms from bacteria to mammals, with consequences for stress responses, development, and cell fate decisions (Raj and Oudenaarden 2008).

Consider the SOCS gene in our model. In Chapter 1 the promoter G had a concentration of 1 μM , which in a cell of 1 pL volume corresponds to about 600,000 copies. The real number is one or two. A single promoter is either bound by pSTAT or not, and it produces mRNA only while it is bound, so the production of SOCS mRNA is a sequence of discrete events whose timing is random. The question we want to answer is whether this randomness matters for the signaling response, and if so, which features of the response it affects.

3.2 From concentrations to molecule numbers

To describe a network by molecule numbers we need the number of molecules that corresponds to a given concentration. For a volume V in liters, 1 μM corresponds to

$$N_A V \times 10^{-6} \text{ molecules per } \mu\text{M},$$

where $N_A = 6.022 \times 10^{23} \text{ mol}^{-1}$ is Avogadro's number. In a 1 pL mammalian cell this is about 6×10^5 molecules per μM , so 1 nM corresponds to about 600 molecules. In a bacterium of 1 fL volume, 1 nM corresponds to less than one molecule.

Rate constants must be converted as well. A first-order rate constant, with units of inverse time, is unchanged, since it describes what happens to each molecule independently. A second-order rate constant in $\mu\text{M}^{-1} \text{ min}^{-1}$ is divided by the number of molecules per μM to give a constant per pair of molecules. In BNGL we carry out these conversions in the `parameters` block, so that the model can still be written in terms of familiar concentrations. Here is the ligand-receptor model of Chapter 1 written this way:

```
begin parameters
  NAV      100          # molecules per uM in the reaction volume
  LO_uM    1            # ligand (uM)
  RO_uM    1            # receptor (uM)
  kon_uM   1            # binding (1/uM/min)
  koff     0.1          # unbinding (1/min)
  LO       rint(LO_uM*NAV) # ligand (molecules)
  RO       rint(RO_uM*NAV) # receptor (molecules)
  kon      kon_uM/NAV     # binding (1/molecule/min)
end parameters
```

The parameter NAV is the number of molecules per μM . With NAV = 100 the model contains 100 ligand and 100 receptor molecules, which corresponds to a reaction volume of about 0.17 fL. BioNetGen does not perform these conversions automatically for models without compartments. It treats the numbers in the `species` block as amounts in whatever units the rate constants assume, and it is up to the modeler to keep the two consistent. A later chapter on compartments shows how BioNetGen can handle volumes explicitly.

3.3 The stochastic description of a reaction network

In the stochastic description, the state of the system at time t is the vector $\mathbf{x}$ of molecule numbers, and each reaction j has a **propensity** $a_j(\mathbf{x})$, defined so that $a_j(\mathbf{x}) dt$ is the probability that one reaction event of type j occurs in the next small time interval dt . For an elementary reaction the propensity is the per-molecule rate constant multiplied by the number of distinct combinations of reactant molecules. For ligand–receptor binding and unbinding,

$$a_1 = k_{\text{on}} x_{\text{L}} x_{\text{R}}, \quad a_2 = k_{\text{off}} x_{\text{LR}},$$

where k_{on} is now the per-molecule constant. For a reaction between two molecules of the same type, the number of distinct pairs is $x(x-1)/2$, and BioNetGen accounts for the corresponding symmetry factors when it generates the network.

The probability $P(\mathbf{x}, t)$ of finding the system in state $\mathbf{x}$ obeys the **chemical master equation**,

$$\frac{dP(\mathbf{x}, t)}{dt} = \sum_j \left[a_j(\mathbf{x} - \nu_j) P(\mathbf{x} - \nu_j, t) - a_j(\mathbf{x}) P(\mathbf{x}, t) \right], \quad (3.1)$$

where ν_j is the column of the stoichiometry matrix for reaction j , the change in $\mathbf{x}$ when reaction j fires. The first term in the brackets is the flux of probability into state $\mathbf{x}$ from the states that lead to it, and the second is the flux out. Equation 3.1 is a system of linear equations with one equation for every possible state. For all but the smallest networks the number of states is far too large to solve it directly, so instead we sample trajectories of the process it describes. When the volume and the numbers of all molecules grow together, the fluctuations become small compared with the mean, and the trajectories approach the solution of the ODEs of Chapter 1. At finite numbers, the mean of the stochastic process obeys the ODEs only for networks of first-order reactions. For a bimolecular reaction the mean rate depends on the average of a product, such as $\langle LR \rangle$, which differs from the product of the averages, $\langle L \rangle \langle R \rangle$, whenever the two numbers fluctuate together. The difference is small when the numbers are large, but it need not be small in general.

3.4 Gillespie's algorithm

In 1977 Daniel Gillespie described a method for generating trajectories that are exact samples of the process described by Equation 3.1. The method, now called the **stochastic simulation algorithm** (SSA) or Gillespie's algorithm, follows directly from the definition of the propensity. Since the probability that each reaction fires in the next short interval depends only on the current state, the waiting time until the next event of any kind is exponentially distributed with rate $a_0 = \sum_j a_j(\mathbf{x})$, and the probability that the next event is reaction j is a_j/a_0 . One step of the algorithm is thus:

1. Compute the propensities $a_j(\mathbf{x})$ and their sum a_0 .
2. Draw a random number r_1 uniformly from $(0, 1)$ and advance time by $\tau = -\ln(r_1)/a_0$.
3. Draw a second random number r_2 and choose reaction j with probability a_j/a_0 .
4. Update the state, $\mathbf{x} \leftarrow \mathbf{x} + \nu_j$, and repeat.

The algorithm is short enough to write ourselves for ligand–receptor binding:

```

rng = np.random.default_rng(1)

def gillespie(L, R, LR, kon, koff, t_end):
    """Exact stochastic simulation of L + R <-> LR."""
    t, times, counts = 0.0, [0.0], [LR]
    while True:
        a1, a2 = kon * L * R, koff * LR          # propensities
        a0 = a1 + a2
        if a0 == 0:                               # no reaction can occur
            break
        t += -np.log(rng.random()) / a0          # time to the next event
        if t > t_end:                             # next event is too late
            break
        if rng.random() * a0 < a1:               # choose which event
            L, R, LR = L - 1, R - 1, LR + 1
        else:
            L, R, LR = L + 1, R + 1, LR - 1
        times.append(t)
        counts.append(LR)
    times.append(t_end)                          # hold the last state to
    ↪ t_end
    counts.append(LR)
    return np.array(times), np.array(counts)

t_hand, LR_hand = gillespie(L=100, R=100, LR=0, kon=0.01, koff=0.1,
    ↪ t_end=20)

```

In bngsim the same simulation is a change of method. The model file defines the conversion to molecule numbers, so nothing else needs to change. Figure 3.1 compares one trajectory from each implementation with the ODE solution.

```

model = bngsim.Model.from_bngl(model_path("ch03", "lr_counts"))
ssa = bngsim.Simulator(model, method="ssa")
ode = bngsim.Simulator(model, method="ode")

model.reset()
r_ssa = ssa.run(t_span=(0, 20), n_points=201, seed=2)
model.reset()
r_ode = ode.run(t_span=(0, 20), n_points=201)

plt.step(t_hand, LR_hand, where="post", lw=1, label="SSA, Python
    ↪ implementation")
plt.step(r_ssa.time, r_ssa.observables["LR"], where="post", lw=1,
    ↪ label="SSA, bngsim")
plt.plot(r_ode.time, r_ode.observables["LR"], "k--", lw=1.5, label="ODE")
plt.xlabel("time (min)")
plt.ylabel("LR (molecules)")

```

```
plt.legend(loc="lower right");
```

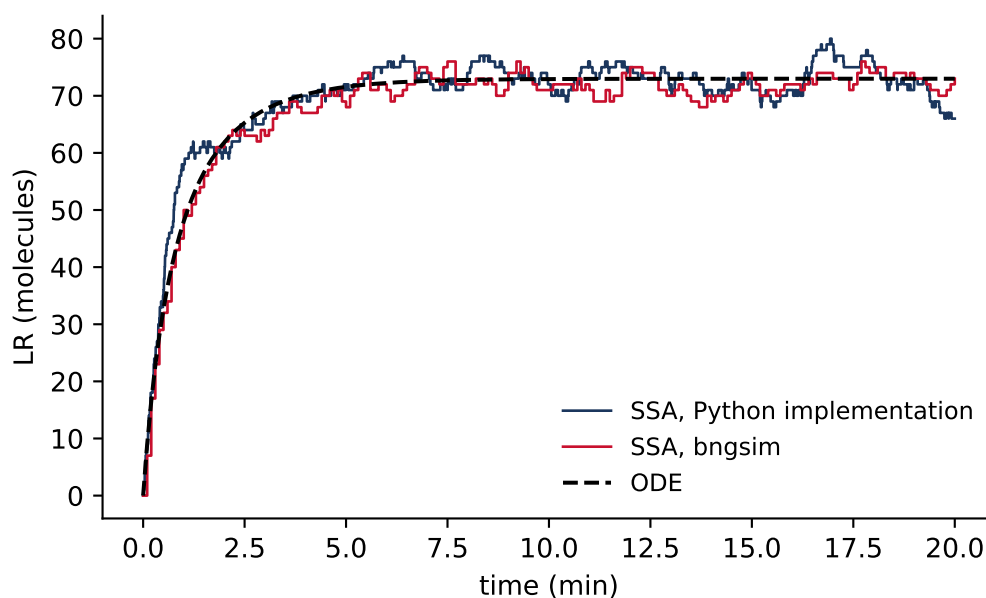


Figure 3.1: Stochastic simulations of ligand–receptor binding with 100 ligand and 100 receptor molecules, from the Python implementation above and from bngsim, compared with the ODE solution. Both trajectories fluctuate about the deterministic curve. The Python implementation records every event, so its trajectory moves in steps of one molecule. bngsim records the state at 201 evenly spaced times, so a single step in its trajectory can contain several events.

Every stochastic simulation with a different random seed gives a different trajectory. The `seed` argument makes a run reproducible, and `run_replicates` runs many trajectories with seeds `seed`, `seed + 1`, and so on, which is how we build up the statistics in the rest of this chapter.

3.5 Noise and system size

Consider the same binding reaction in volumes of different size, with the concentrations and rate constants held fixed. The ODE solution is identical in every case, since it depends only on concentrations, but the stochastic trajectories are not. Figure 3.2 shows that the relative size of the fluctuations shrinks as the number of molecules grows.

```
fig, ax = plt.subplots(figsize=(4.2, 3.2))
for NAV in [10, 100, 1000]:
    model.set_param("NAV", NAV)
    model.reset()
    r = ssa.run(t_span=(0, 20), n_points=401, seed=4)
    ax.plot(r.time, r.observables["LR"] / NAV, lw=1, label=f"NAV = {NAV}")
```

```

model.set_param("NAV", 100)
model.reset()
ax.plot(r_ode.time, r_ode.observables["LR"] / 100, "k--", lw=1.5)
ax.set_xlabel("time (min)")
ax.set_ylabel("[LR] ( $\mu\text{M}$ )")
ax.legend(loc="lower right")
plt.show()

navs = np.array([10, 30, 100, 300, 1000, 3000])
mean_lr, cv_lr = [], []
for NAV in navs:
    model.set_param("NAV", NAV)
    model.reset()
    reps = ssa.run_replicates(400, t_span=(0, 30), n_points=31, seed=5)
    x = np.array([rep.observables["LR"][-1] for rep in reps])
    mean_lr.append(x.mean())
    cv_lr.append(x.std() / x.mean())
model.set_param("NAV", 100)
mean_lr, cv_lr = np.array(mean_lr), np.array(cv_lr)

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.loglog(mean_lr, cv_lr, "o-")
ax.loglog(mean_lr, cv_lr[0] * np.sqrt(mean_lr[0] / mean_lr), "k--", lw=1)
ax.set_xlabel("mean number of LR complexes")
ax.set_ylabel("coefficient of variation")
plt.show()

```

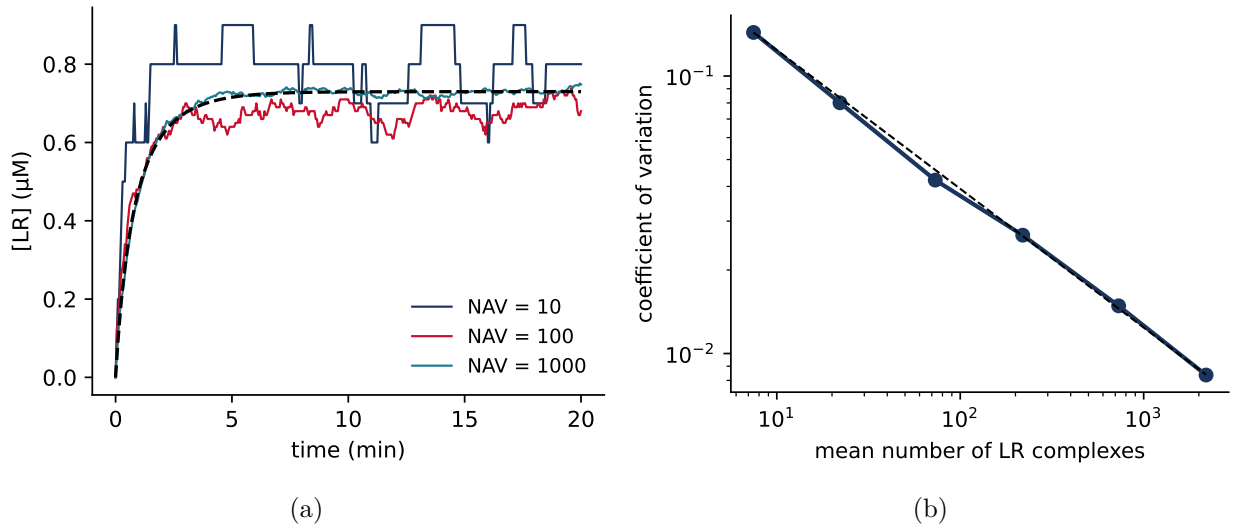


Figure 3.2: Fluctuations shrink as the number of molecules grows. (a) Single trajectories of $[\text{LR}]$ at three values of NAV, the number of molecules per μM , with the ODE solution (dashed). (b) Coefficient of variation of the number of complexes at equilibrium, from 400 trajectories at each volume. The dashed line has slope $-1/2$.

With about 7 complexes at equilibrium, the standard deviation of the number of complexes is 14% of the mean. With about 2189 complexes it is 0.8%. The coefficient of variation falls as $1/\sqrt{N}$, the dependence expected when a quantity is the sum of many independent random events. This scaling explains why the ODEs work well for species that are abundant and whose fluctuations arise from their own reactions. For a Poisson distribution, the coefficient of variation is $1/\sqrt{\langle N \rangle}$, so a protein present at 10,000 copies would fluctuate by 1%, which is smaller than the precision of most of the measurements we want to explain. More generally, the square of the coefficient of variation is the Fano factor, defined in the next section, divided by the mean. Processes that produce molecules in bursts, or that inherit fluctuations from a species present at small numbers, have large Fano factors, and an abundant protein made from a single gene can fluctuate far more than the Poisson value. Noise thus matters most in species present at small numbers and in the processes they drive.

3.6 Bursts in gene expression

Gene expression is the clearest example of a process driven by small numbers. Figure 3.3 shows the simplest model that captures it, in which a promoter switches between an inactive and an active state, and mRNA is made only while the promoter is active (Peccoud and Ycart 1995). The model uses a structured molecule, `Gene(p~off~on)`, whose component `p` records the state of the promoter.

```
begin parameters
  kact      0.05    # promoter activation (1/min)
  kinact    0.05    # promoter inactivation (1/min)
  ktx       4       # transcription from the active promoter (1/min)
  kdeg      0.1     # mRNA degradation (1/min)
end parameters

begin molecule types
  Gene(p~off~on)
  mRNA()
end molecule types

begin seed species
  Gene(p~off) 1
end seed species

begin observables
  Molecules GeneOn  Gene(p~on)
  Molecules mRNA    mRNA()
end observables

begin reaction rules
  Switch:      Gene(p~off) <-> Gene(p~on)          kact, kinact
  Transcribe:  Gene(p~on)  -> Gene(p~on) + mRNA()  ktx
  Degrade:     mRNA()     -> 0                     kdeg
end reaction rules
```

At steady state, the mean number of mRNA molecules is the rate of transcription from the active promoter, multiplied by the fraction of time the promoter is active, divided by the rate constant for degradation:

$$\langle m \rangle = \frac{k_{\text{tx}}}{k_{\text{deg}}} \frac{k_{\text{act}}}{k_{\text{act}} + k_{\text{inact}}},$$

which is 20 for the parameters above. The mean says nothing about how expression is distributed in time, however. A useful measure of noise is the **Fano factor**, the variance divided by the mean, which equals 1 for a Poisson distribution. For this model with a single promoter, at steady state,

$$\frac{\sigma_m^2}{\langle m \rangle} = 1 + \frac{k_{\text{tx}} k_{\text{inact}}}{(k_{\text{act}} + k_{\text{inact}})(k_{\text{act}} + k_{\text{inact}} + k_{\text{deg}})}. \quad (3.2)$$

When the promoter switches rapidly compared with mRNA degradation, the second term becomes small, and the distribution of mRNA approaches the Poisson distribution that transcription at a constant rate would give. When the promoter switches slowly compared with mRNA degradation, and many transcripts are made in each period of activity, each period produces a **burst** of transcripts, and the distribution becomes much broader. Figure 3.3 compares the two cases, which have the same mean.

```
gene = bngsim.Model.from_bngl(model_path("ch03", "gene_expression"))
gssa = bngsim.Simulator(gene, method="ssa")
cases = {"slow switching": 0.05, "fast switching": 50}

fig, ax = plt.subplots(figsize=(4.2, 3.2))
for label, k in cases.items():
    gene.set_param("kact", k)
    gene.set_param("kinact", k)
    gene.reset()
    r = gssa.run(t_span=(0, 500), n_points=1001, seed=8)
    ax.step(r.time, r.observables["mRNA"], where="post", lw=1, label=label)
ax.set_xlabel("time (min)")
ax.set_ylabel("mRNA (molecules)")
ax.legend(loc="upper left")
plt.show()

fano = {}
fig, ax = plt.subplots(figsize=(4.2, 3.2))
for label, k in cases.items():
    gene.set_param("kact", k)
    gene.set_param("kinact", k)
    gene.reset()
    reps = gssa.run_replicates(2000, t_span=(0, 200), n_points=3, seed=7)
    m = np.array([rep.observables["mRNA"][-1] for rep in reps])
    fano[label] = m.var() / m.mean()
    if label == "slow switching":
```

```

    zero_frac = np.mean(m == 0)
    high_frac = np.mean(m > 2 * m.mean())
    ax.hist(m, bins=np.arange(0, 80, 2), alpha=0.6, density=True,
    ↪ label=label)
ax.set_xlabel("mRNA (molecules)")
ax.set_ylabel("probability per molecule")
ax.legend()
plt.show()

```

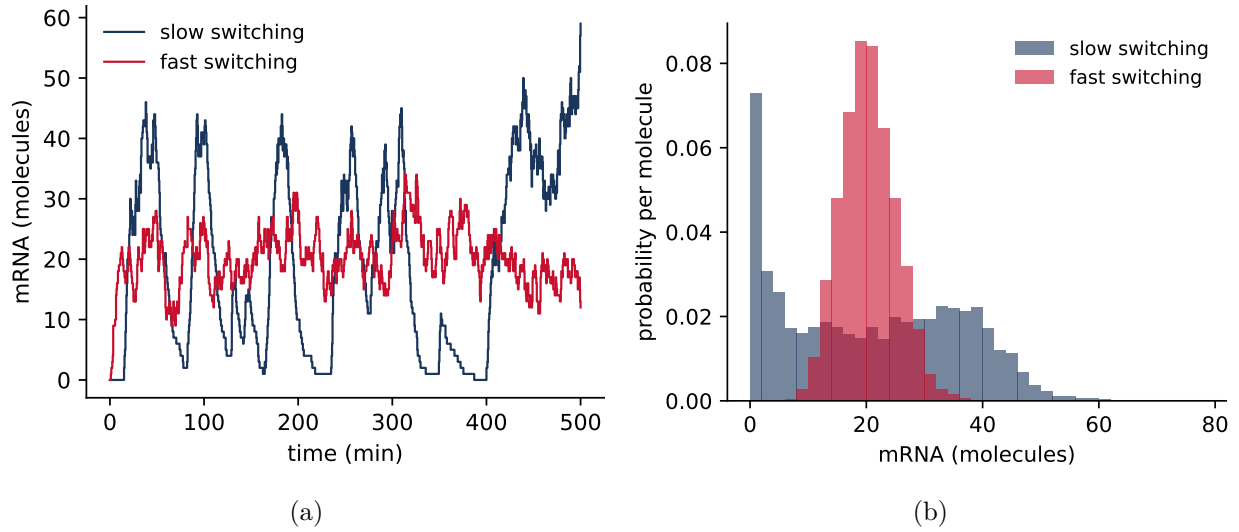


Figure 3.3: Transcriptional bursting. Slow promoter switching ($k_{\text{act}} = k_{\text{inact}} = 0.05 \text{ min}^{-1}$) and fast switching (50 min^{-1}) give the same mean of 20 mRNA molecules but very different fluctuations. (a) Single trajectories. (b) Distributions of mRNA number across 2000 cells at $t = 200 \text{ min}$.

The simulated Fano factors are 11.2 for slow switching and 1.09 for fast switching, compared with 11.0 and 1.02 from Equation 3.2. With slow switching, 10% of cells contain no mRNA at all at a given moment, while 9% contain more than twice the mean. A protein made from these transcripts averages over the bursts for a time set by its own lifetime, so bursting matters most when the protein is short-lived or when its effect depends on how much is present at a particular moment.

3.7 A noisy feedback loop

We now return to the cytokine–STAT–SOCS circuit and give it a single copy of the SOCS gene. The model `stat_ssa_feedback.bngl` is the rule-based model of Chapter 2 written in molecule numbers. To keep the simulations fast, it uses `NAV = 1000`, so that the receptor is present at 1000 copies, STAT at 10,000, and the gene at one. A real mammalian cell has a volume about 1000 times larger, with correspondingly more copies of the abundant proteins, but the same one or two copies of the gene. The gene-expression rate constants are chosen so that the fully induced gene gives about 20 SOCS mRNA molecules.

NAV	1000	# molecules per uM in the reaction volume
GO	1	# SOCS gene copies
kGon	1/NAV	# pSTAT-promoter binding (1/molecule/min)
kGoff	1	# pSTAT-promoter unbinding (1/min)
ktx	2.5	# transcription from the bound promoter (1/min)
ktl	12	# translation (1/min)

Figure 3.4 shows ten individual cells, the mean of 200 cells, and the ODE solution of the same model.

```

stat = bngsim.Model.from_bngl(model_path("ch03", "stat_ssa_feedback"))
sssa = bngsim.Simulator(stat, method="ssa")
sode = bngsim.Simulator(stat, method="ode")

stat.reset()
cells = sssa.run_replicates(200, t_span=(0, 240), n_points=481, seed=11)
P = np.array([c.observables["pSTAT"] for c in cells])
t = cells[0].time
stat.reset()
r_det = sode.run(t_span=(0, 240), n_points=481)

fig, ax = plt.subplots(figsize=(4.2, 3.2))
for p in P[:10]:
    ax.plot(t, p, color="0.6", lw=0.8)
ax.plot(t, P.mean(axis=0), color="#1B365D", lw=2, label="mean of 200 cells")
ax.plot(r_det.time, r_det.observables["pSTAT"], "k--", lw=1.5, label="ODE")
ax.set_xlim(0, 120)
ax.set_xlabel("time (min)")
ax.set_ylabel("total pSTAT (molecules)")
ax.legend()
plt.show()

peak_cv = P.max(axis=1).std() / P.max(axis=1).mean()
late = P[:, -1]
late_cv = late.std() / late.mean()

stat.set_param("kGoff", 0.05)          # slower switching, same occupancy
stat.set_param("kGon", 0.05 / 1000)
stat.reset()
slow = sssa.run_replicates(200, t_span=(0, 240), n_points=481, seed=11)
late_slow = np.array([c.observables["pSTAT"][-1] for c in slow])
late_slow_cv = late_slow.std() / late_slow.mean()
stat.set_param("kGoff", 1)
stat.set_param("kGon", 1 / 1000)

fig, ax = plt.subplots(figsize=(4.2, 3.2))
bins = np.linspace(0, 1600, 33)

```

```

ax.hist(late, bins=bins, alpha=0.6, density=True, label="default switching")
ax.hist(late_slow, bins=bins, alpha=0.6, density=True, label="slow
↪ switching")
ax.set_xlabel("adapted pSTAT (molecules)")
ax.set_ylabel("probability per molecule")
ax.legend()
plt.show()

```

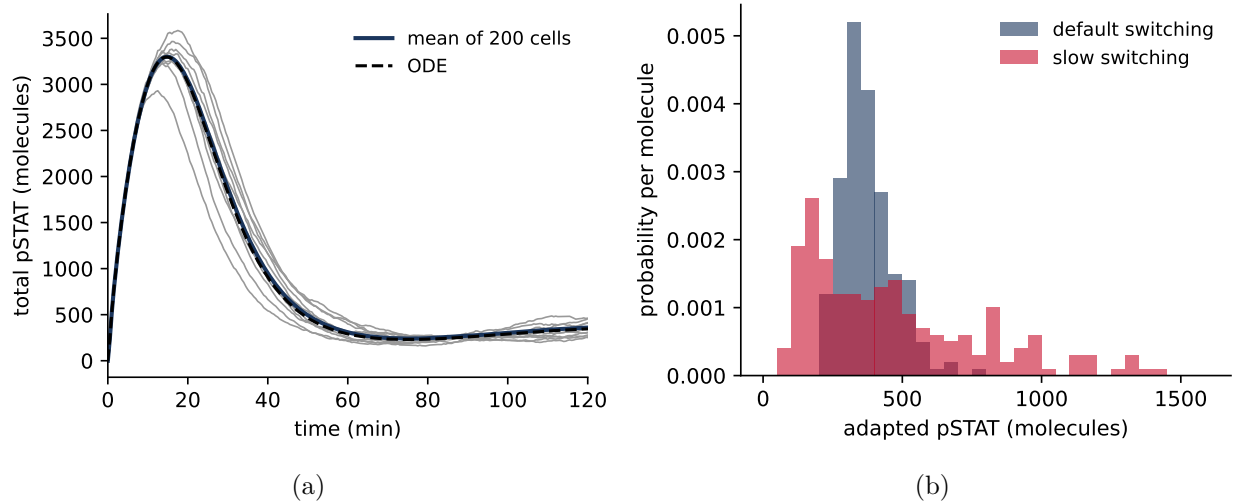


Figure 3.4: Stochastic simulation of the cytokine–STAT–SOCS circuit with a single SOCS gene. (a) Total pSTAT in ten individual cells (gray), the mean of 200 cells (blue), and the ODE solution (dashed). (b) Distribution of the adapted pSTAT level at 240 min across 200 cells, for the default parameters and for a promoter that switches 20 times more slowly with the same occupancy.

The mean of many cells follows the ODE solution closely, and the peak is nearly the same in every cell: its coefficient of variation is 5%. This is because the peak occurs before the feedback acts. It is set by the binding and activation reactions, which involve thousands of molecules, and by the time SOCS has been made the peak has passed. The adapted level, in contrast, is set by the balance between STAT activation and inhibition by SOCS, and the amount of SOCS depends on the activity of a single gene and on a handful of mRNA molecules. The adapted level thus varies by 25% from cell to cell with the default parameters, and by 69% when the promoter switches slowly.

These results show that the same circuit can make some features of its response reliable and others variable. A measurement that averages over a population, such as a western blot, reports the mean, which for this model follows the ODE solution closely. A measurement in single cells, such as flow cytometry or imaging, would reveal the spread in the adapted response, and the width of that spread carries information about the kinetics of the SOCS promoter that the population average does not.

3.8 Faster approximate methods

The cost of an exact stochastic simulation is proportional to the number of reaction events, which grows with the number of molecules. Models that contain both rare species, like a gene, and abundant ones, like a cytoplasmic protein present at millions of copies, spend nearly all of their time simulating reactions of the abundant species, whose fluctuations are small. Several approximate methods address this problem. In **tau leaping** each step advances time by an interval τ , chosen so that the propensities change little during it, and fires a Poisson-distributed number of each reaction (Gillespie 2007). In the **partial scaling algorithm** (PSA) of Lin, Feng, and Hlavacek, populations above a critical size are simulated as if they were scaled down, which reduces the number of events while leaving small populations exact (Lin et al. 2019). `bngsim` provides PSA through the same interface as SSA:

```
psa = bngsim.Simulator(model, method="psa", poplevel=100)
```

where `poplevel` is the critical population size. Larger values give more accurate results and less acceleration. Since PSA is approximate, it is good practice to compare it with exact SSA on a smaller version of the problem before relying on it. A third approach, network-free stochastic simulation, avoids generating the reaction network at all and is the subject of a later chapter.

3.9 Summary

- When species are present in small numbers, the random timing of reactions causes cells to differ, and the stochastic description is needed.
- Converting a model to molecule numbers leaves first-order rate constants unchanged and divides second-order constants by the number of molecules per μM .
- The propensity a_j of a reaction is the probability per unit time that it fires. The chemical master equation describes how the probability of each state evolves, and Gillespie's algorithm generates exact sample trajectories.
- When all molecule numbers grow with the volume, relative fluctuations scale as $1/\sqrt{N}$, and the ODEs describe the system well. Noise matters most for species present at small numbers and for the processes they drive, which can include abundant proteins made from a single gene.
- Slow promoter switching produces bursts of transcription, which increase the Fano factor of mRNA well above the Poisson value of 1.
- In the cytokine–STAT–SOCS circuit with a single SOCS gene, the peak of the response is nearly the same in every cell, while the adapted level varies with the activity of the gene.

3.10 Exercises

1. **Implement the algorithm.** Extend the `gillespie` function to simulate `stat_step4_release.bngl` from Chapter 1 in molecule numbers. Compare the mean of 100 trajectories with `bngsim`'s SSA and with the ODEs.

2. **System size.** In `lr_counts.bngl`, plot the full distribution of the number of LR complexes at equilibrium for $NAV = 10$ and 1000 . How does its shape change? Why is the coefficient of variation in Figure 3.2 smaller than $1/\sqrt{\langle N \rangle}$?
3. **Bursts.** In `gene_expression.bngl`, keep the mean mRNA number at 20 and vary the switching rates over four orders of magnitude, with $k_{act} = k_{inact}$. Plot the simulated Fano factor against the prediction of Equation 3.2.
4. **Gene copy number.** Give the cell in `stat_ssa_feedback.bngl` two copies of the SOCS gene ($G0 = 2$) and halve `ktx`, so that the mean transcription rate is unchanged. How does the spread of the adapted level change, and why?
5. **Speed and accuracy.** Set NAV to 30,000 in `stat_ssa_feedback.bngl` and compare the run time and the distribution of peak pSTAT from SSA and from PSA with `poplevel = 100` and 1000.

3.11 Further reading

Gillespie (2007) reviews the theory of stochastic chemical kinetics and the exact and approximate simulation methods built on it. Raj and Oudenaarden (2008) reviews the sources and consequences of noise in gene expression, and Elowitz et al. (2002) is the experiment that separated intrinsic from extrinsic noise. Lin et al. (2019) describes the partial scaling algorithm implemented in BioNetGen and bngsim.

4 A field guide to rules

Most models of signaling are assembled from a small number of recurring mechanisms: binding, modification by enzymes, synthesis, degradation, and the formation of larger complexes. Chapter 2 introduced the syntax of rules, and this chapter shows how to write rules for each of these mechanisms, what reactions BioNetGen generates from them, and where the generated network can differ from what the modeler intended. Each section is built around a small model in `models/ch04/`, and for each one we look at the reactions BioNetGen produces. Reading the generated network is the most reliable way to check a rule, and new modelers should make it a habit. The chapter closes with a list of common mistakes that are hard to spot without reading the generated network.

Learning goals

After this chapter you should be able to:

- write rules for binding, including binding of identical molecules and of molecules with several identical sites, and predict the statistical factors BioNetGen applies;
- use context to make a rate depend on the state of another site;
- model enzymatic modification with an explicit mechanism or with a Michaelis–Menten rate law, and choose between them;
- write rules for synthesis and degradation, including degradation of a molecule that is part of a complex;
- define rate laws with functions of observables;
- write rules for the formation of chains and rings, and limit the size of the network they generate.

4.1 Reading the generated network

Every rule in this chapter is checked the same way. BioNetGen generates the network, and we list the reactions it contains, written out with the full species on each side and the rate law of each reaction. The listing comes from the `.net` file that `generate_network` writes, which can also be opened directly in any text editor. The rate laws in this listing contain the statistical factors that BioNetGen derives from the structure of the molecules, and these factors are the main subject of the next section.

4.2 Binding

4.2.1 Identical partners

Consider two molecules of the same type that bind each other through a single site to form a dimer:

```
Dimerize: A(d) + A(d) <-> A(d!1).A(d!1) kf, kr
```

BioNetGen generates the following reactions:

```
A(d) + A(d) -> A(d!1).A(d!1) 0.5*kf [Dimerize]
A(d!1).A(d!1) -> A(d) + A(d) kr [Dimerize, reverse]
```

The rate constant of the forward reaction is $0.5 \cdot k_f$, not k_f . The factor of one half accounts for the fact that the two reactants are identical. In a system of N molecules there are $N(N-1)/2$ distinct pairs, not N^2 , and the factor in the generated reaction makes the deterministic and stochastic descriptions consistent with this count. The ODE for the dimer is thus

$$\frac{d[A_2]}{dt} = \frac{1}{2} k_f [A]^2 - k_r [A_2].$$

The practical consequence concerns rate constants taken from the literature. If a published model writes the dimerization rate as $k[A]^2$, the rule must use $2 \cdot k$ to reproduce it. It is important to note that conventions differ among papers and software packages, so a rate constant for a symmetric reaction should never be copied without checking which convention was used.

4.2.2 Multiple identical sites

The opposite situation arises when one molecule has several identical sites for a partner. In `bivalent.bngl`, molecule A has two equivalent sites for B, declared by repeating the component name, `A(b,b)`. A single rule describes binding at either site:

```
Bind: A(b) + B(a) <-> A(b!1).B(a!1) kon, koff
```

```
A(b,b) + B(a) -> A(b!1,b).B(a!1) 2*kon [Bind]
B(a) + A(b!1,b).B(a!1) -> A(b!1,b!2).B(a!1).B(a!2) kon [Bind]
A(b!1,b).B(a!1) -> A(b,b) + B(a) koff [Bind, reverse]
A(b!1,b!2).B(a!1).B(a!2) -> B(a) + A(b!1,b).B(a!1) 2*koff [Bind, reverse]
```

Free A binds B at rate $2 \cdot k_{on}$, since either of its two empty sites can be used, while A with one site filled binds a second B at rate k_{on} . Likewise, the complex with two bound B molecules releases one at rate $2 \cdot k_{off}$. These factors follow from counting the ways a rule can be applied to a species, and BioNetGen computes them automatically.

4.2.3 Cooperativity through context

Binding at one site often changes binding at another. In `cooperative.bngl`, a receptor has two sites, `a` and `b`, and ligand binds site `b` ten times faster when site `a` is already occupied:

```
First: L(r) + R(a,b) <-> L(r!1).R(a!1,b)    kon1, koff
Second: L(r) + R(a!+,b) <-> L(r!1).R(a!+,b!1) kon2, koff
```

```
L(r) + R(a,b) -> L(r!1).R(a!1,b)    kon1    [First]
L(r!1).R(a!1,b) -> L(r) + R(a,b)    koff     [First, reverse]
L(r) + L(r!1).R(a!1,b) -> L(r!1).L(r!2).R(a!2,b!1) kon2    [Second]
L(r!1).L(r!2).R(a!2,b!1) -> L(r) + L(r!1).R(a!1,b)    koff     [Second, reverse]
```

The context `R(a,b)` in the first rule restricts it to receptors with both sites empty, and `R(a!+,b)` restricts the second rule to receptors with site `a` filled. The generated network shows a consequence that is easy to miss. The reverse of each rule applies the same context as the forward direction, so ligand bound at site `a` can leave only while site `b` is empty. If both sites can release their ligand independently, the unbinding reactions must be written as separate rules without that context, as we did for STAT in Chapter 2. When the sites can be filled in either order, however, the four binding rate constants cannot be chosen independently, since both orders lead to the same doubly occupied receptor. Chapter 8 derives this constraint and shows how energy patterns satisfy it automatically.

4.3 Enzymatic modification

A kinase that phosphorylates a substrate can be modeled at two levels of detail. The explicit mechanism follows the elementary steps, binding of enzyme and substrate followed by catalysis and release:

```
# explicit mechanism
ESbind: E(s) + S(e,Y~0) <-> E(s!1).S(e!1,Y~0)    kon, koff
EScat:  E(s!1).S(e!1,Y~0) -> E(s) + S(e,Y~P)      kcat
```

When the enzyme–substrate complex reaches a quasi-steady state quickly compared with the depletion of substrate, the rate of phosphorylation takes the familiar Michaelis–Menten form,

$$v = \frac{k_{\text{cat}}[E]_T[S]}{K_m + [S]}, \quad (4.1)$$

where $[E]_T$ is the total enzyme, free and bound, $[S]$ is the free substrate, and $K_m = (k_{\text{off}} + k_{\text{cat}})/k_{\text{on}}$. The classical form of the approximation also replaces free by total substrate, which is valid only when little substrate is bound to enzyme, that is, when $[E]_T \ll [S]_T + K_m$ (Borghans et al. 1996). BNGL provides a Michaelis–Menten rate law as the built-in `MM(kcat,Km)`:

```
# consumed. MM treats the amounts matching the two patterns as totals.
MMcat: S2(Y~O) + E2() -> S2(Y~P) + E2()      MM(kcat,Km)
```

The substrate is written first and the enzyme second, and the enzyme appears on both sides of the rule, since it is not consumed. MM treats the amounts of the species that match the two reactant patterns as the totals $[S]_T$ and $[E]_T$, and it does not assume that they are nearly equal to the free amounts. It computes the free substrate from

$$[S] = \frac{1}{2} \left[([S]_T - K_m - [E]_T) + \sqrt{([S]_T - K_m - [E]_T)^2 + 4K_m[S]_T} \right]$$

and uses this value in Equation 4.1. This is the total quasi-steady-state approximation of Borghans et al. (1996), which accounts for the substrate held by the enzyme and remains accurate at much higher enzyme concentrations (Tzafriri 2003). When $[E]_T \ll [S]_T + K_m$, free and total substrate are nearly equal, and MM reduces to the classical form. Figure 4.1 compares the explicit mechanism, MM, and the classical form.

```
model = bngsim.Model.from_bngl(model_path("ch04", "enzyme"))
sim = bngsim.Simulator(model, method="ode")
kcat, Km, S0 = (model.get_param(p) for p in ("kcat", "Km", "S0"))

def classical(t, E0):
    # Integrate dSp/dt = kcat*E0*S/(Km + S) with S = S0 - Sp (Euler, fine
    ↪ grid)
    sp = np.zeros_like(t)
    for i in range(1, len(t)):
        S = S0 - sp[i-1]
        sp[i] = sp[i-1] + (t[i] - t[i-1]) * kcat * E0 * S / (Km + S)
    return sp

def t_half(t, y):
    return np.interp(S0 / 2, y, t)

half = {}
for E0, t_end in [(0.01, 200), (0.5, 8)]:
    model.set_param("E0", E0)
    model.reset()
    r = sim.run(t_span=(0, t_end), n_points=4001)
    cl = classical(r.time, E0)
    half[E0] = {"explicit": t_half(r.time, r.observables["Sp_explicit"]),
               "MM": t_half(r.time, r.observables["Sp_MM"]),
               "classical": t_half(r.time, cl)}
    fig, ax = plt.subplots(figsize=(4.2, 3.2))
    ax.plot(r.time, r.observables["Sp_explicit"], color="#1B365D",
    ↪ label="explicit")
    ax.plot(r.time, r.observables["Sp_MM"], "--", color="#C8102E",
    ↪ label="MM")
    ax.plot(r.time, cl, ":", color="0.4", label="classical")
```

```

ax.set_xlabel("time (min)")
ax.set_ylabel("phosphorylated substrate (μM)")
ax.legend(loc="lower right")
plt.show()
model.set_param("E0", 0.01)

```

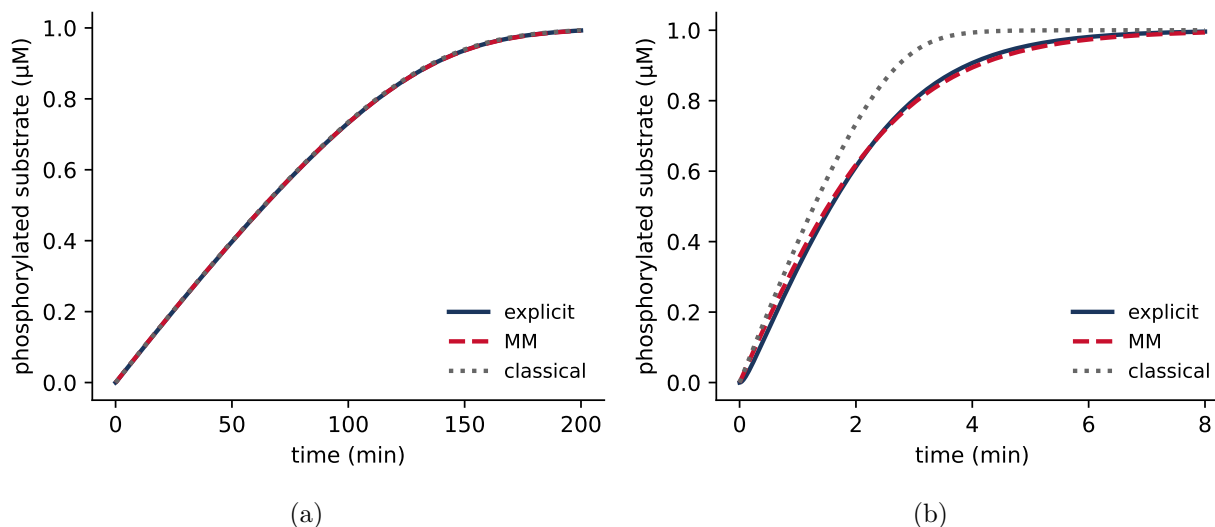


Figure 4.1: Phosphorylation by a kinase described by an explicit binding–catalysis mechanism (solid), by the MM rate law (dashed), and by the classical Michaelis–Menten form (dotted), with 1 μM substrate. (a) Scarce enzyme, 0.01 μM: the three agree. (b) Abundant enzyme, 0.5 μM: the classical form runs too fast, while MM remains close to the explicit mechanism.

The three descriptions agree closely when the enzyme is scarce. With enzyme at half the concentration of substrate, the classical form overestimates the rate, since it counts substrate held by the enzyme as free, and it predicts that half of the substrate is phosphorylated in 1.28 min rather than 1.57 min. MM gives 1.52 min. The rate law is more economical, since it introduces no enzyme–substrate complexes, and its two parameters, k_{cat} and K_m , are the ones usually reported by enzymologists. The explicit mechanism, on the other hand, is required whenever the enzyme–substrate complex matters in its own right: when substrates compete for the enzyme, when the enzyme is abundant enough to be sequestered, or when the complex has other interactions, as a receptor-bound STAT does.

The rate law also hides a trap, which appeared while the models for this chapter were being written. In the first version of `enzyme.bngl`, the same kinase molecule was used for both the explicit and the Michaelis–Menten rules. The explicit rules tied up most of the kinase in enzyme–substrate complexes. The MM rule counted as its total enzyme only the kinase that matched its enzyme pattern, which excluded the kinase held in these complexes, and the Michaelis–Menten reaction ran about five times too slowly. The model file now uses a separate kinase for each mechanism. The general lesson is that an enzyme described by an MM rate law should not also participate in explicit binding rules for the same reaction.

4.4 Making and removing molecules

Synthesis and degradation were introduced in Chapter 1. Synthesis from nothing, $0 \rightarrow A()$, describes production at a constant rate, for example from a gene that is not part of the model, and synthesis from a template, $G(s!+) \rightarrow G(s!+) + \text{mRNA}()$, describes production at a rate proportional to the amount of the template. Degradation needs more care when the molecule being degraded can be part of a complex. `degradation.bngl` contains three pairs of bound molecules and three ways of writing a degradation rule:

```
begin reaction rules
  # Deletes the whole complex that contains A, including B
  DegA: A() -> 0 kdeg
  # Deletes only C; its partner D is released
  DegC: C() -> 0 kdeg DeleteMolecules
  # Writes the outcome explicitly: E is removed and F is released
  DegE: E(f!1).F(e!1) -> F(e) kdeg
end reaction rules
```

```
A(b!1).B(a!1) -> 0 kdeg [DegA]
C(d!1).D(c!1) -> D(c) kdeg [DegC]
E(f!1).F(e!1) -> F(e) kdeg [DegE]
```

The first rule removes not only A but its partner B, since by default a molecule deleted by a rule takes the entire complex that contains it. The keyword `DeleteMolecules` in the second rule changes this behavior, so that only C is removed and its partner D is released. The third rule gives the same result as the second by writing the product explicitly. Which behavior is correct depends on the biology. Degradation of a receptor by internalization may reasonably take its bound ligand with it, while proteolysis of one subunit of a complex usually releases the others.

4.5 Rate laws from functions

Sometimes a process is not worth modeling at the level of elementary reactions, either because its mechanism is unknown or because its details are not the subject of the model. BNGL allows the rate of a rule to be any mathematical function of parameters and observables, defined in a `functions` block. In `hill.bngl`, transcription of a target gene is a Hill function of the amount of phosphorylated STAT, which replaces the explicit promoter binding of Chapter 1:

```
begin functions
  fTx() = vmax*pSTAT^n/(K^n + pSTAT^n)
end functions
begin reaction rules
  Activate: S(Y~O) <-> S(Y~P) kact, kdeact
  Transcribe: 0 -> mRNA() fTx()
  Degrade: mRNA() -> 0 kmdeg
```

end reaction rules

The function `fTx()` is evaluated as the simulation proceeds, using the current value of the observable `pSTAT`. A function used as a rate law takes the place of the rate constant, so the rate of each generated reaction is the value of the function multiplied by the amounts of the reactants, just as for mass action. Here the rule has no reactants, and the rate of mRNA synthesis is thus `fTx()` itself. With a Hill coefficient $n = 2$ and half-maximal `pSTAT` K , the rate rises steeply as `pSTAT` passes K , mimicking cooperative binding of STAT dimers to the promoter without modeling the dimers.

Lumped rate laws of this kind should be used deliberately. A Hill function fitted to a dose-response curve can reproduce the curve without saying anything about the mechanism that produced it, and it cannot predict how the response changes when the mechanism is perturbed, for example by a mutation that prevents dimerization. A useful rule of thumb is to use explicit mechanisms for the parts of a system that are the subject of the study, and lumped rate laws only for the parts that are not.

4.6 Chains, rings, and the size of the network

When molecules have more than one binding site, they can form chains and rings of any length. The classic example in immune signaling is the crosslinking of receptors by multivalent ligands. `aggregation.bngl` contains a ligand with two sites and a receptor with two sites, with one rule for binding between complexes and one for closing a ring:

```
Bind:  L(r) + R(l) <-> L(r!1).R(l!1)           kon, koff
# the reverse opens a ring without splitting the complex
Close: L(r).R(l) <-> L(r!1).R(l!1)           kring, koff
```

The rule `Close` has a single reactant pattern, `L(r).R(l)`, in which the dot requires the ligand and the receptor to be in the same complex without requiring a bond between them. It therefore describes binding between a free ligand site and a free receptor site anywhere in one complex, which closes a chain of any length into a ring, and it is a unimolecular reaction with rate constant `kring` in units of inverse time. In contrast, the rule `Bind` has two reactant patterns separated by `+`, and BioNetGen applies it only to sites on different complexes. The distinction matters because the rate of ring closure, which depends on the flexibility and geometry of the complex, has no relation to the rate of binding between molecules in solution. Our model gives every ring the same `kring`, although a real chain closes at a rate that depends on its length.

The same distinction applies in reverse. The reverse of `Bind` breaks a bond only when the two products are separate complexes, and the reverse of `Close` breaks a bond only when the product remains a single complex, which is the case for a bond in a ring. Each bond in the network is thus broken by exactly one of the two rules, and BioNetGen enforces this by checking that the products of each reaction match the `+` or `.` written in the rule. Without the reverse of `Close`, rings could never open, and the model would slowly convert every molecule into rings. Since only a ligand end and a receptor end can close, only chains with equal numbers of ligands and receptors form rings.

Without a limit, the rule `Bind` generates chains of every length, and network generation never finishes. The option `max_agg` in the `generate_network` action caps the number of molecules in any complex. Table 4.1 shows how the network grows as the cap is raised.

```
import pandas as pd, tempfile, os, re
src = model_path("ch04", "aggregation").read_text()
# A ring is a species with no free site, r or l, on any molecule.
rows = []
for max_agg in [2, 4, 6, 8, 10]:
    with tempfile.TemporaryDirectory() as d:
        path = os.path.join(d, "aggregation.bngl")
        open(path, "w").write(src.replace("max_agg=>4",
            ↪ f"max_agg=>{max_agg}"))
        m = bngsim.Model.from_bngl(path, cache=False)
        rings = [sp for sp in m.species_names if not re.search(r"[r1][,)]", sp)]
        rows.append({"max_agg": max_agg, "species": m.n_species,
            "reactions": m.n_reactions, "rings": len(rings)})
pd.DataFrame(rows).set_index("max_agg")
```

Table 4.1: Size of the network generated from `aggregation.bngl` as a function of `max_agg`, the largest complex allowed, and the number of distinct rings it contains.

	species	reactions	rings
max_agg			
2	4	4	1
4	8	16	2
6	12	34	3
8	16	60	4
10	20	92	5

The number of reactions grows faster than the number of species, since each new chain can react with every existing one. A cap on complex size is only justified when larger complexes are rare, which should be checked by simulating with a larger cap and confirming that the results do not change. When large aggregates are important, as they are in the activation of immunoreceptors, the network becomes too large to generate, and network-free simulation is the method of choice.

4.7 Common mistakes

The following mistakes are easy to make and hard to detect from simulation results alone.

1. **Ignoring the symmetry factor.** A rule for binding of identical molecules generates a reaction with half the written rate constant. Check the convention of any rate constant taken from the literature.

2. **Missing context.** A rule without context applies to every species that matches it, including complexes the modeler did not have in mind. Read the generated network for unintended reactions, as in Chapter 2, where removing one context from the ligand rule added reactions for every receptor complex.
3. **Context in the reverse direction.** A reversible rule applies its context in both directions. When the forward and reverse reactions should have different context, write them as two rules.
4. **Deleting whole complexes.** A degradation rule removes the entire complex containing the deleted molecule unless `DeleteMolecules` is given or the products are written explicitly.
5. **Mixing explicit and lumped descriptions of the same enzyme.** An MM rate law treats the amount of enzyme matching its pattern as the total enzyme, so an enzyme that also binds substrate through explicit rules will be undercounted.
6. **Counting the wrong thing.** A `Molecules` observable counts matches and a `Species` observable counts complexes. A function of an observable inherits this difference.
7. **Inconsistent units.** Rate constants and initial amounts must use the same units, either concentrations or molecule numbers, as discussed in Chapter 3.

4.8 Summary

- Reading the reactions BioNetGen generates is the most reliable way to check that a rule does what was intended.
- BioNetGen applies statistical factors automatically: one half for binding of identical molecules, and integer factors for molecules with several identical sites.
- Context makes rates depend on the state of other sites, and it applies in both directions of a reversible rule.
- Enzymatic modification can be modeled explicitly or with the MM rate law, which uses the total quasi-steady-state approximation. It agrees with the explicit mechanism, and with the classical Michaelis–Menten form, when the enzyme is scarce, and it remains accurate at higher enzyme concentrations than the classical form.
- Degradation removes the whole complex unless `DeleteMolecules` is used or the products are given explicitly.
- Rate laws can be functions of observables, which is useful for lumping processes that are not the subject of a model.
- Rules for multivalent binding generate chains and rings of unbounded size. `max_agg` limits network generation, and network-free simulation avoids it.

4.9 Exercises

1. **Symmetry.** Write a model in which two molecules of A bind through two distinct sites, $A(x,y)$, with x binding y . Compare the generated network with `homodimer.bngl`. Which reactions have statistical factors, and why?
2. **Cooperativity.** Modify `cooperative.bngl` so that either site can be occupied first and ligand can leave either site at any time. How many rules do you need? Plot the fraction of receptors with both sites filled as a function of ligand concentration for `kon2` = 0.1, 1, and 10. Do your rate constants satisfy the cycle constraint of Section 8.1?

3. **When does Michaelis–Menten fail?** In `enzyme.bngl`, vary `E0` from 0.001 to 1 μM and measure the time for half of the substrate to be phosphorylated by each mechanism. Explain where and why the two begin to disagree.
4. **Degradation.** Add a degradation rule for SOCS to `stat_rbm1_rules.bngl` from Chapter 2 that degrades SOCS whether it is free or bound to the receptor. Should the receptor be released or degraded with it? Write both versions and compare the adapted level of pSTAT.
5. **Capping aggregation.** Simulate `aggregation.bngl` with `max_agg` = 4, 6, and 8, and plot the observables `Ring2` and `Ring4` over time. Is a cap of 4 adequate for these parameters? How would you decide? What happens to the rings if `Close` is made irreversible?

4.10 Further reading

Sekar and Faeder (2012) contains an extensive library of example rules for signaling mechanisms, from binding at the membrane to gene regulation. Faeder et al. (2009) describes the syntax of rules, rate laws, and network generation options in detail.

Part II

Building realistic models

5 Compartments and transport

The models of Part I treat the cell as a single well-mixed volume. A real cell is divided by membranes into compartments, and where a molecule is matters as much as what state it is in. A cytokine binds its receptor in the extracellular medium, the receptor sits in the plasma membrane, STAT is phosphorylated in the cytoplasm, and the SOCS gene is transcribed in the nucleus. Moving between these places takes time, and the same reaction runs at different rates in compartments of different size, since the same number of molecules gives a higher concentration in a smaller volume. In this chapter we describe the compartments of a cell in BNGL, write rules that apply in every compartment with the correct rates, write rules for transport across membranes, and use them to place the cytokine–STAT–SOCS circuit in a cell with a nucleus and endosomes.

Before compartments were part of the language, modelers encoded location as an internal state of each molecule, writing for example `R(loc~surf~endo)`, and wrote every reaction once for each location with its own rate constant. This works, but the geometry appears nowhere in the model, the software cannot check that molecules in different places do not react, and the number of rules grows with the number of compartments. Leonard Harris, Justin Hogg, and the author introduced compartmental BNGL, or cBNGL, to address these problems (Harris et al. 2009). In cBNGL location is a property of the compartment that holds a molecule, compartments are declared with their sizes and topology, and BioNetGen scales the rate of each reaction by the size of the compartment in which it occurs.

Learning goals

After this chapter you should be able to:

- declare the compartments of a cell and the topology that connects them;
- specify the location of molecules and complexes in seed species, observables, and rules;
- explain how BioNetGen scales bimolecular rate constants by compartment size, and choose units that make the scaling transparent;
- write rules for transport of molecules and complexes between compartments, including endocytosis;
- recognize the ways a compartmental model can generate reactions the modeler did not intend.

5.1 Declaring compartments

A cell is a set of nested compartments of two kinds: **volumes**, which are three-dimensional, and **surfaces**, which are the two-dimensional membranes that separate volumes. Figure 5.1 shows the compartments used in this chapter. The plasma membrane (PM) separates the extracellular medium (EC) from the cytoplasm (CP); the nuclear envelope (NM) separates the cytoplasm from the nucleus

(NU); and the endosomal membrane (EM) separates the cytoplasm from the endosome lumen (EN). The tree on the right is the form BioNetGen uses. Volumes and surfaces must alternate along every branch, a surface may enclose only one volume, and a volume may contain several surfaces, as the cytoplasm contains both the nuclear and the endosomal membranes.

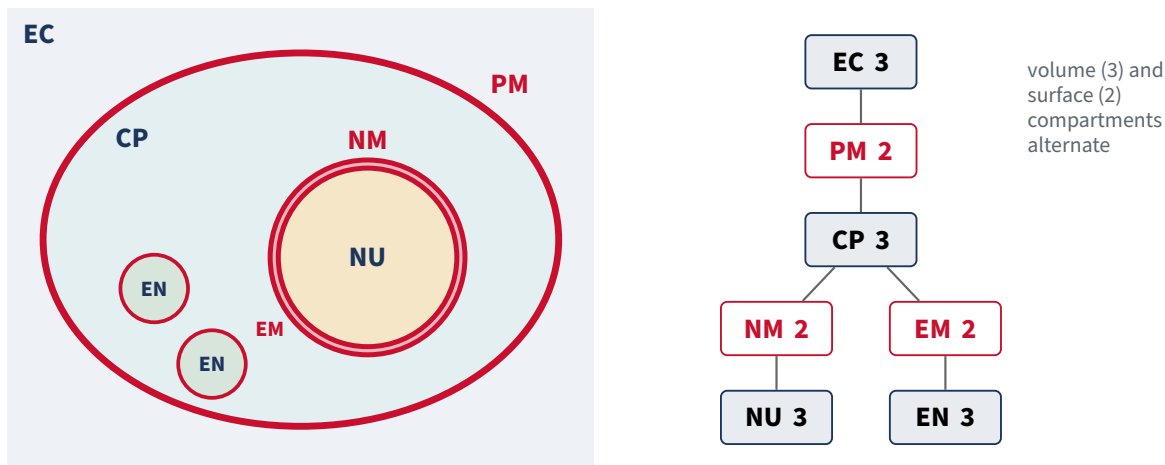


Figure 5.1: The compartments of a cell and their topology. Left, a cell with a nucleus and endosomes; the endosomes are lumped into one compartment in the models. Right, the tree of compartments declared in BNGL, with the dimension of each.

Compartments are declared in a block of their own. Each line gives a name, the dimension (3 for a volume, 2 for a surface), the size, and the compartment that encloses it. The compartments of `stat_compartments.bngl` are:

```
begin compartments
  EC  3  V_EC*NA_pL          # extracellular medium
  PM  2  A_PM*h*um3*NA_pL    EC  # plasma membrane
  CP  3  V_CP*NA_pL          PM  # cytoplasm
  NM  2  A_NM*h*um3*NA_pL    CP  # nuclear envelope
  NU  3  V_NU*NA_pL          NM  # nucleus
end compartments
```

The outermost compartment, here the medium, has no parent. The size of a volume is its volume. The size of a surface is its area multiplied by an effective thickness, h , which converts the area into a volume for the purpose of rate scaling, as discussed below. The value of h cancels from every comparison between surface reactions, but every surface in a model must use the same value.

5.2 Where molecules are

The location of a molecule is written with `@` after it, and the location of a whole species with `@` and a double colon before it. The seed species of the STAT model place each molecule in its compartment:

```

begin seed species
  L(r,r)@EC      L0
  R(l,s,socs)@PM R0
  S(r,Y~0,d,g)@CP S0
  G(s)@NU        G0
end seed species

```

A complex can span compartments. When the cytokine binds the receptor, the cytokine stays in the medium and the receptor stays in the membrane, and BioNetGen writes the complex as

```
@PM::L(r!1,r)@EC.R(l!1,s,socs)
```

The prefix `@PM::` gives the location of the complex as a whole, and the suffix `@EC` gives the location of the cytokine within it. A complex is located on a surface if any of its molecules is on that surface, and in a volume only if all of its molecules are in that volume. A complex may include one surface and molecules in the two volumes adjacent to it, but no more, so a receptor in the plasma membrane can bind a ligand outside the cell and STAT inside it, but a single complex cannot reach from the plasma membrane to the nucleus.

Observables and rules may name a compartment or leave it out. The observable `S()@NU` counts STAT molecules in the nucleus, while `S()` would count STAT everywhere. A rule that names no compartment is **universal**, meaning that it applies to matching molecules in every compartment where they can meet, and BioNetGen works out where that is.

5.3 How compartment size sets the rate

Consider a binding reaction that occurs in both the cytoplasm and the nucleus, with the same number of molecules of A and B in each, and with the nucleus four times smaller than the cytoplasm. The model `two_volumes.bngl` contains one universal rule for this reaction,

```

# One rule, no compartment named: it applies wherever A and B meet
Bind:  A(b) + B(a) <-> A(b!1).B(a!1)    kon, koff

```

and BioNetGen generates one forward reaction for each compartment:

```

@CP::A(b) + @CP::B(a) -> @CP::A(b!1).B(a!1)  kon/(V_CP*NA_pL)  [Bind]
@NU::A(b) + @NU::B(a) -> @NU::A(b!1).B(a!1)  kon/(V_NU*NA_pL)  [Bind]
@CP::A(b!1).B(a!1) -> @CP::A(b) + @CP::B(a)  koff  [Bind, reverse]
@NU::A(b!1).B(a!1) -> @NU::A(b) + @NU::B(a)  koff  [Bind, reverse]

```

The rate constant of each bimolecular reaction is the rate constant of the rule divided by the size of the compartment, while the unimolecular reverse reactions are not scaled. In general, a bimolecular rate constant is divided by the size of the compartment in which the reaction occurs, and if the reactants are in a volume and on an adjacent surface, by the size of the volume. Reactions of a single reactant, including transport, are never scaled.

The scaling is simplest to follow when compartment sizes are given in units that convert concentrations to numbers of molecules. In this chapter every size is written as NAV, the number of molecules per μM in the compartment, $N_A V \times 10^{-6}$ with V in liters (Chapter 3), which is 6.02×10^5 for a volume of 1 pL. The amounts of all species are then numbers of molecules per cell, and the rate constants of the rules keep the familiar units of $\mu\text{M}^{-1} \text{min}^{-1}$. Dividing k_{on} by NAV gives the rate constant per pair of molecules, which is the conversion we made by hand in Chapter 3, and compartments give each part of the cell its own value of NAV.

```

model = bngsim.Model.from_bngl(model_path("ch05", "two_volumes"))
sim = bngsim.Simulator(model, method="ode")
model.reset()
r = sim.run(t_span=(0, 5), n_points=201)
NO = model.get_param("NO")

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.plot(r.time, r.observables["AB_CP"], label="cytoplasm")
ax.plot(r.time, r.observables["AB_NU"], label="nucleus")
ax.set_xlabel("time (min)")
ax.set_ylabel("AB complexes (molecules)")
ax.legend(loc="lower right")
plt.show()

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.plot(r.time, r.observables["AB_CP"] / NO, label="cytoplasm")
ax.plot(r.time, r.observables["AB_NU"] / NO, label="nucleus")
ax.set_xlabel("time (min)")
ax.set_ylabel("fraction of A bound")
plt.show()
frac_cp = r.observables["AB_CP"][-1] / NO
frac_nu = r.observables["AB_NU"][-1] / NO

```

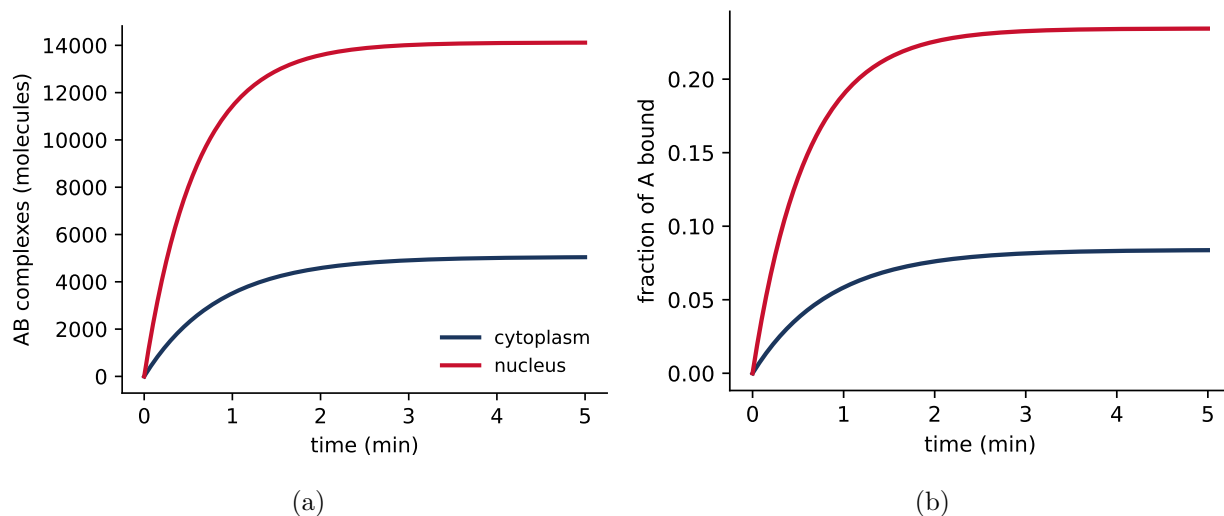


Figure 5.2: The same binding rule in two compartments. The cytoplasm (1 pL) and the nucleus (0.25 pL) start with the same numbers of A and B. (a) Number of AB complexes. (b) Fraction of A bound. Binding is faster and more complete in the nucleus, where the same numbers of molecules are four times more concentrated.

Figure 5.2 shows the consequence. With 60,220 molecules of A and of B in each compartment, the concentration of each is 0.1 μM in the cytoplasm and 0.4 μM in the nucleus. At equilibrium 8% of A is bound in the cytoplasm and 23% in the nucleus, as the dissociation constant of 1 μM predicts for these concentrations. Since the rule is written once with an intrinsic rate constant, the difference between the compartments comes entirely from their sizes. If the nucleus were made larger, the model would change only in the compartments block.

Surfaces follow the same arithmetic. Two receptors that meet in the plasma membrane react at a rate divided by the size of the membrane, which is NAV computed for an effective volume Ah , the area A times an effective thickness h . A smaller membrane, such as that of an endosome, concentrates its receptors and speeds their reactions in the same way. The thickness is a modeling convention rather than the physical width of a bilayer. It relates a two-dimensional rate constant k_{2D} , with units of area per molecule per time, to the three-dimensional constant used in the rule by $k_{3D} = k_{2D} h$, so that the rate per unit area is the same in either description. We fix h at 10 nm in this chapter. If h is changed, every rate constant of a reaction in a membrane must be changed in proportion, or the two-dimensional rates the model represents will change.

It is important to note that the rate constant of a compartmental rule is an intrinsic constant, independent of compartment size. A rate constant taken from a model that wrote one rule per location, or fitted separately to each location, may have the compartment size built into it already, for example if the source model counted molecules rather than concentrations. Copying such a constant into a compartmental rule applies the scaling twice, and the model runs without complaint but with every such rate wrong by a factor of the compartment size. The scaling applies to every bimolecular rule, whether it names a compartment or not, so the equations and units of the source model should be checked before any constant is copied.

5.4 Transport

Molecules move between compartments by **transport** rules, which change the location of a molecule or a complex without changing its bonds or states. Transport is unimolecular, so its rate constant is not scaled. The STAT model uses three kinds:

```
# pSTAT dimers are imported as a unit (species transport)
Import:      @CP:S(d!1).S(d!1) -> @NU:S(d!1).S(d!1)          kimp
# unphosphorylated monomers shuttle in both directions (molecule transport)
Shuttle:     S(r,Y~0,d,g)@CP <-> S(r,Y~0,d,g)@NU            kimp0, kexp
mExport:     mSOCS()@NU -> mSOCS()@CP                        kmexp
```

Shuttle and **mExport** are **molecule transport**. The named molecule changes compartment, here between the cytoplasm and the nucleus across the nuclear envelope. A molecule may move between two volumes that share a surface, or between a volume and an adjacent surface, and BioNetGen rejects a move that would leave a bond spanning compartments that are not adjacent.

Import is **species transport**, written with the location prefix on both sides. The whole complex, here a pSTAT dimer, moves to the nucleus as a unit. The rate of species transport applies once per complex. A dimer is imported at rate **kimp**, the same as a complex containing only one matching molecule would be. A third form, species transport between surfaces, describes endocytosis and is taken up in Section 5.6. Finally, the keyword **MoveConnected** after a molecule transport rule makes the molecule carry with it the molecules bound to it in the same compartment, which is useful for cargo carried by a transport receptor (Harris et al. 2009).

5.5 STAT in a cell with a nucleus

STAT proteins carry the cytokine signal from the membrane to the nucleus. In unstimulated cells STAT1 and STAT3 shuttle between the cytoplasm and the nucleus, with most of the protein in the cytoplasm at steady state. Phosphorylated STAT dimers are imported actively and accumulate in the nucleus, where they are dephosphorylated by the nuclear phosphatase TC45 and exported again (Reich and Liu 2006; Hoeve et al. 2002). `stat_compartments.bngl` adds these steps to the model of Chapter 2 (version 5, with STAT dimers). The membrane and receptor rules are unchanged and name no compartment:

```
# --- at the membrane: no compartments named, BNG finds where each applies
↪ ---
LRbind:      L(r,r) + R(l) <-> L(r!1,r).R(l!1)              kon, koff
Sbind:       R(l!+,s) + S(r,Y~0) -> R(l!+,s!1).S(r!1,Y~0)   kSon
# --- dimerization occurs in both cytoplasm and nucleus ---
Sdimer:      S(r,Y~P,d,g) + S(r,Y~P,d,g) <-> S(r,Y~P,d!1,g).S(r,Y~P,d!1,g)
↪ kdim, kdim
```

BioNetGen places each reaction where its reactants can meet. The cytokine binds the receptor at a rate scaled by the volume of the medium, and STAT docks on the receptor at a rate scaled by the

volume of the cytoplasm. pSTAT dimerizes both in the cytoplasm and in the nucleus, each at its own rate. Steps that happen in only one place name the compartment:

```
# --- in the nucleus ---
Sdephos:    S(d,g,Y~P)@NU -> S(d,g,Y~O)@NU          kdepshos
Transcribe: G(s!+) -> G(s!+) + mSOCS()@NU             ktx
# --- in the cytoplasm ---
Translate:  mSOCS()@CP -> mSOCS()@CP + SOCS(r)@CP      ktl
```

Dephosphorylation occurs only in the nucleus, as TC45 does. Transcription produces mRNA in the nucleus, which is exported to the cytoplasm and translated there. The generated network shows the reactions that involve nuclear transport, dimerization, and the promoter:

```
@CP::S(Y~O,d,g,r) -> @NU::S(Y~O,d,g,r)  kimp0  [Shuttle]
@NU::S(Y~O,d,g,r) -> @CP::S(Y~O,d,g,r)  kexp  [Shuttle, reverse]
@CP::S(Y~P,d,g,r) + @CP::S(Y~P,d,g,r) -> @CP::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r)
↳ 0.5*kdim/(V_CP*NA_pL)  [Sdimer]
@CP::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r) -> @CP::S(Y~P,d,g,r) + @CP::S(Y~P,d,g,r)
↳ kundim  [Sdimer, reverse]
@CP::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r) -> @NU::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r)
↳ kimp  [Import]
@NU::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r) -> @NU::S(Y~P,d,g,r) + @NU::S(Y~P,d,g,r)
↳ kundim  [Sdimer, reverse]
@NU::G(s) + @NU::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r) ->
↳ @NU::G(s!1).S(Y~P,d!2,g!1,r).S(Y~P,d!2,g,r)  2*kGon/(V_NU*NA_pL)
↳ [Gbind]
@NU::S(Y~P,d,g,r) + @NU::S(Y~P,d,g,r) -> @NU::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r)
↳ 0.5*kdim/(V_NU*NA_pL)  [Sdimer]
@NU::G(s!1).S(Y~P,d!2,g!1,r).S(Y~P,d!2,g,r) -> @NU::G(s) +
↳ @NU::S(Y~P,d!1,g,r).S(Y~P,d!1,g,r)  kGoff  [Gbind, reverse]
```

Dimerization is four times faster per pair of molecules in the nucleus than in the cytoplasm. A pSTAT dimer binds the promoter at twice the written rate, since either of its two DNA-binding sites can bind.

```
model = bngsim.Model.from_bngl(model_path("ch05", "stat_compartments"))
sim = bngsim.Simulator(model, method="ode")
model.reset()
r = sim.run(t_span=(0, 240), n_points=481)
o = r.observables

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.plot(r.time, o["pSTAT_CP"])
ax.plot(r.time, o["pSTAT_NU"])
ax.set_xlabel("time (min)")
ax.set_ylabel("pSTAT (molecules)")
```

```

ax2 = ax.twinx()
ax2.plot(r.time, o["R_SOCS"], color="#5B6770", ls="--", lw=1.5)
ax2.set_ylabel("SOCS-bound receptors", color="#5B6770")
ax2.spines["right"].set_visible(True)
ax2.set_ylim(0, 1.15 * model.get_param("R0"))
ax.text(60, 12400, "nucleus", color="#C8102E", fontsize=9)
ax.text(22, 3000, "cytoplasm", color="#1B365D", fontsize=9)
ax2.text(240, 0.88 * model.get_param("R0"), "SOCS-bound\nreceptors",
        color="#5B6770", fontsize=9, ha="right", va="top")
plt.show()

nuc = o["STAT_NU"] / (o["STAT_NU"] + o["STAT_CP"])
# block export at 60 min: continue from the state at 60 min with kexp = 0
model.reset()
r1 = sim.run(t_span=(0, 60), n_points=121)
model.set_param("kexp", 0)
r2 = sim.run(t_span=(60, 240), n_points=361)
model.set_param("kexp", 0.2)
nuc_lmb = np.concatenate([r1.observables["STAT_NU"] /
        (r1.observables["STAT_NU"] + r1.observables["STAT_CP"]),
        r2.observables["STAT_NU"] /
        (r2.observables["STAT_NU"] + r2.observables["STAT_CP"])]])
t_lmb = np.concatenate([r1.time, r2.time])

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.plot(r.time, nuc, label="default")
ax.plot(t_lmb, nuc_lmb, "--", label="export blocked at 60 min")
ax.set_xlabel("time (min)")
ax.set_ylabel("nuclear fraction of STAT")
ax.set_ylim(0, 1)
ax.legend(loc="upper left")
plt.show()

ptot = o["pSTAT_CP"] + o["pSTAT_NU"]
i_pk = np.argmax(ptot)
t_pk, nu_share = r.time[i_pk], o["pSTAT_NU"][i_pk] / ptot[i_pk]
nuc_pk = nuc.max()
nuc_rest = model.get_param("kimp0") / (model.get_param("kimp0") +
        model.get_param("kexp"))
conc_ratio = (o["pSTAT_NU"][i_pk] / model.get_param("V_NU")) /
        (o["pSTAT_CP"][i_pk] / model.get_param("V_CP"))

```

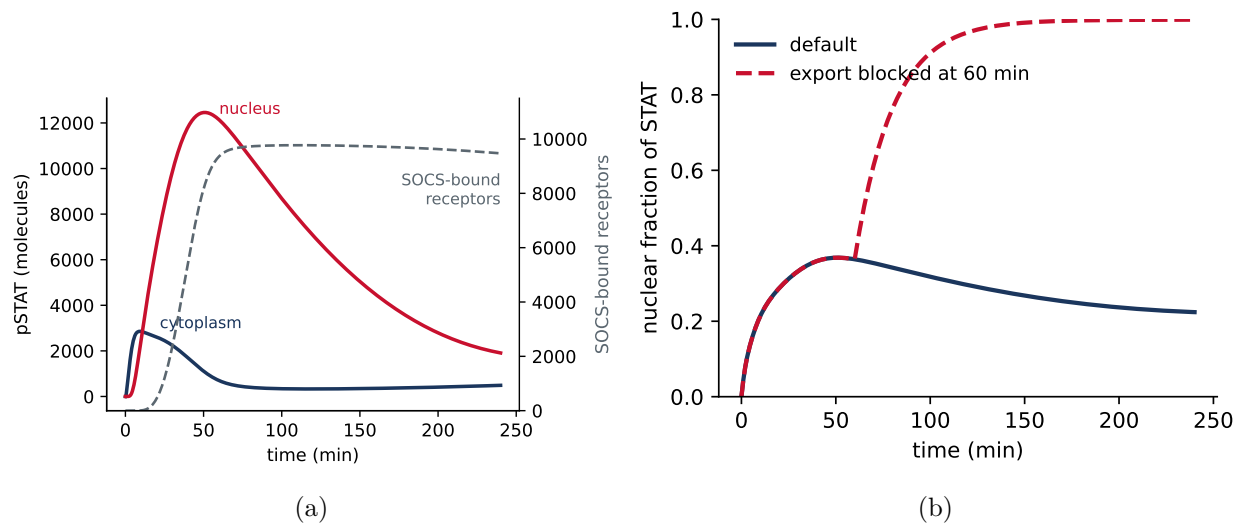


Figure 5.3: Nuclear accumulation of STAT after cytokine stimulation. (a) pSTAT molecules in the cytoplasm and the nucleus, with SOCS bound to receptors (right axis). (b) Fraction of all STAT that is in the nucleus, for the default model and with nuclear export blocked at 60 min.

Figure 5.3 shows the response to 1 nM cytokine. Total pSTAT peaks at 46 min, and at the peak 90% of it is in the nucleus. Since the nucleus is four times smaller than the cytoplasm, the concentration of pSTAT in the nucleus is then 36 times its concentration in the cytoplasm, and an image of the cell would show a bright nucleus. pSTAT accumulates in the nucleus because import of the dimer is fast and only unphosphorylated STAT is exported, so pSTAT is trapped until it is dephosphorylated. As SOCS protein accumulates and blocks the receptors, phosphorylation declines, and the nuclear fraction of STAT falls from 37% toward its resting value of 20%, which is set by the shuttling rate constants, $k_{\text{imp}0}/(k_{\text{imp}0} + k_{\text{exp}})$.

The model reproduces a classic experiment. Begitt and coworkers treated cells with leptomycin B, an inhibitor of nuclear export, after stimulation with interferon- and found that STAT1 remained in the nucleus instead of returning to the cytoplasm (Begitt et al. 2000). Setting k_{exp} to zero at 60 min has the same effect in the model (Figure 5.3 b). Dephosphorylated STAT can no longer leave, and the nuclear fraction rises even as the signal declines.

5.6 Endocytosis and receptor trafficking

Many receptors are internalized after they bind ligand. The membrane around the receptor buds into the cell to form an endosome, carrying the receptor into the endosomal membrane and its bound ligand into the endosome lumen. In the acidic endosome many ligands dissociate, and free receptors are recycled to the surface while the ligand is degraded. These processes control how many receptors a cell displays and how long it responds (Lauffenburger and Linderman 1993). `internalization.bngl` describes them with **surface-to-surface species transport**:

```

begin reaction rules
  # Binding applies on both membranes
  LRbind:      L(r,r) + R(1) <-> L(r!1,r).R(1!1)      kon, koff
  Crosslink:    L(r!+,r) + R(1) <-> L(r!+,r!1).R(1!1)    kx, koff
  # Acidic pH speeds ligand release in endosomes
  EndoRelease:  R(1!1)@EM.L(r!1) -> R(1)@EM + L(r)      koff_endo
  # Surface-to-surface species transport: the whole complex moves
  Internalize:  @PM:R(1!+) -> @EM:R(1!+)                kint
  Constitutive: @PM:R(1) -> @EM:R(1)                     kint0
  Recycle:      @EM:R(1) -> @PM:R(1)                     krec
  # Free ligand is degraded in the endosome
  LigandDeg:    L(r,r)@EN -> 0                          kdegL
end reaction rules

```

The rule **Internalize** moves any complex containing a ligand-bound receptor from the plasma membrane to the endosomal membrane. Molecules in the complex that are in the medium move into the endosome lumen, following the topology of endocytosis, in which the outside of the cell becomes the inside of the endosome:

```

@PM::L(r!1,r)@EC.R(1!1) -> @EM::L(r!1,r)@EN.R(1!1) kint [Internalize]
@PM::L(r!1,r!2)@EC.R(1!1).R(1!2) -> @EM::L(r!1,r!2)@EN.R(1!1).R(1!2) kint
↳ [Internalize]

```

The binding rules name no compartment, so they apply on the endosomal membrane as well as on the surface. This is intended, since ligand does bind receptors in endosomes, but it has a consequence that is easy to overlook. The endosome lumen is small, and the rate constant for ligand binding there is divided by its size:

```

@EC::L(r,r) + @PM::R(1) -> @PM::L(r!1,r)@EC.R(1!1) 2*kon/(V_EC*NA_pL)
↳ [LRbind]
@PM::L(r!1,r)@EC.R(1!1) -> @EC::L(r,r) + @PM::R(1) koff [LRbind, reverse]
@EM::L(r!1,r)@EN.R(1!1) -> @EM::R(1) + @EN::L(r,r) koff [LRbind, reverse]
@EM::R(1) + @EN::L(r,r) -> @EM::L(r!1,r)@EN.R(1!1) 2*kon/(V_EN*NA_pL)
↳ [LRbind]

```

```

model = bngsim.Model.from_bngl(model_path("ch05", "internalization"))
sim = bngsim.Simulator(model, method="ode")
runs = {}
for ke in [1, 100, 1000, 10000]:
    model.set_param("koff_endo", ke)
    model.reset()
    runs[ke] = sim.run(t_span=(0, 120), n_points=241)
model.set_param("koff_endo", 1000)
V_EN = model.get_param("V_EN")
R0 = model.get_param("R0")

```

```

fig, ax = plt.subplots(figsize=(4.2, 3.2))
for ke, rr in runs.items():
    ax.plot(rr.time, rr.observables["R_surface"] / R0,
    ↪ label=f"$k_{{off,endo}}$ = {ke:g} min-1")
ax.set_xlabel("time (min)")
ax.set_ylabel("fraction of receptors on surface")
ax.set_ylim(0, 1.02)
ax.legend(loc="upper right")
plt.show()

fig, ax = plt.subplots(figsize=(4.2, 3.2))
for ke, rr in runs.items():
    line, = ax.plot(rr.time, rr.observables["L_endosome"] / (V_EN * NA_pL),
    ↪ label=f"{ke:g}")
    ax.plot(rr.time, rr.observables["L_free_EN"] / (V_EN * NA_pL), "--",
    ↪ color=line.get_color())
ax.set_xlabel("time (min)")
ax.set_ylabel("ligand in endosome lumen (μM)")
plt.show()

# Rates at which a free receptor in the endosome is captured at 120 min:
# by free ligand in the lumen (LRbind, two sites per ligand, scaled by the
# lumen), and by ligand already held by another receptor (Crosslink, scaled
# by the endosomal membrane).
size_EN = V_EN * NA_pL
size_EM = model.get_param("A_EM") * model.get_param("h") *
    ↪ model.get_param("um3") * NA_pL
def at_end(ke, name):
    return runs[ke].observables[name][-1]
def capture(ke):
    by_free = 2 * model.get_param("kon") * at_end(ke, "L_free_EN") / size_EN
    by_bound = model.get_param("kx") * at_end(ke, "L_one_EN") / size_EM
    return by_free, by_bound
L_en_tot = at_end(1, "L_endosome") / size_EN
free_frac_L = {ke: at_end(ke, "L_free_EN") / at_end(ke, "L_endosome") for ke
    ↪ in runs}
free_frac_R = {ke: at_end(ke, "R_free_EM") / at_end(ke, "R_internal") for ke
    ↪ in runs}
cap_free_slow, cap_bound_slow = capture(1)
cap_fast = sum(capture(10000))
krec = model.get_param("krec")
surf = {ke: rr.observables["R_surface"][-1] / R0 for ke, rr in runs.items()}
L_med = runs[1000].observables["L_medium"][0] * 1e3 /
    ↪ (model.get_param("V_EC") * NA_pL)

```

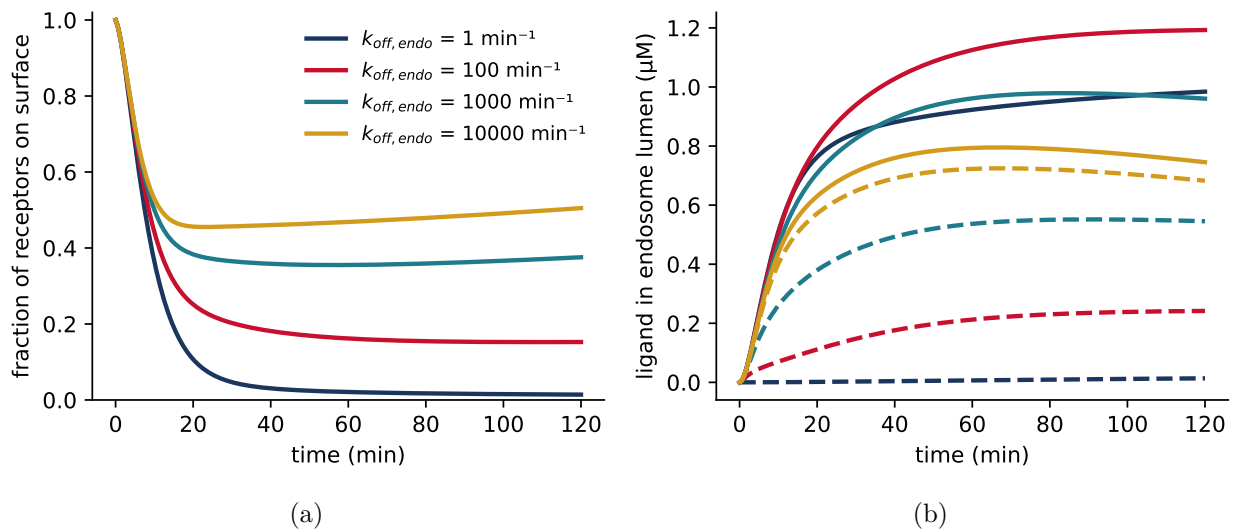


Figure 5.4: Receptor internalization and recycling with 0.1 nM ligand. (a) Receptors on the cell surface for several rates of ligand release in the endosome. (b) Ligand in the endosome lumen for the same cases. Total ligand (solid) reaches about 1 μM in the small lumen in every case, but the free fraction (dashed) depends on the rate of release. Receptors recycle only when release in the endosome is fast compared with their recapture by ligand.

Figure 5.4 shows what happens. With the medium at 0.1 nM, ligand-bound receptors are internalized within minutes. If the acidic endosome speeds ligand release only modestly, to 1 min^{-1} , receptors do not return to the surface, and only 1% of them are there after two hours. The reason is the small size of the lumen. Ligand carried in with the receptors accumulates in a volume of 0.01 pL, where its total concentration reaches 1.0 μM , 9,841 times its concentration in the medium. With slow release most of this ligand stays bound to receptors, and only 1% of it is free. A receptor in the endosome that loses its ligand is captured again in two ways, by binding a free ligand from the lumen, at a rate of about 16 min^{-1} , and by binding the free site of a ligand already held by another receptor in the endosomal membrane, at about 65 min^{-1} . Both rates are far larger than the rate of recycling, 0.1 min^{-1} , so a receptor returns to the surface only if it remains free, and the fraction of internalized receptors that are free is set by the competition between release and recapture. With release at 1 min^{-1} , slower than recapture, only 1% of the internalized receptors are free. With $k_{off,endo} = 10 \text{ min}^{-1}$, release is faster than recapture (about 832 min^{-1} , now almost entirely by free ligand), 92% of the internalized receptors are free, and 50% of all receptors are on the surface after two hours.

The lumped endosome of this model exaggerates the effect, since a real cell has many small endosomes that mature, fuse, and deliver their contents to lysosomes. However, the calculation demonstrates the general point that compartment size affects a model through every bimolecular rule that applies in the compartment, including rules written with another compartment in mind. A universal rule should be checked in every compartment where it can apply, by reading the generated network.

5.7 Common mistakes

1. **Scaling twice.** Rate constants copied from a model with one rule per location, or fitted separately for each location, usually include the compartment size. Divide it out before using the constant in a universal rule.
2. **Inconsistent surface sizes.** Every surface must use the same effective thickness, and the size of a surface must be an area times that thickness, not an area alone.
3. **Rules that reach too far.** A universal rule applies in every compartment where its reactants can meet. Read the generated network for reactions in compartments you did not have in mind, as with ligand rebinding in the endosome above.
4. **Overlapping transport rules.** If two transport rules match the same complex, both apply and their rates add. For example, a complex that contains both a receptor and a co-receptor is internalized twice as fast if each has its own internalization rule.
5. **Observables without a location.** An observable that names no compartment counts molecules everywhere. Name the compartment when the location matters, and convert counts to concentrations with the size of the right compartment.
6. **Impossible topology.** A complex can include only one surface and the two volumes beside it, and a membrane that receives complexes by endocytosis needs a lumen inside it. BioNetGen rejects models that violate these constraints.

5.8 Summary

- cBNGL declares compartments as a tree of alternating volumes and surfaces, with the size of each.
- The location of a molecule is written $M()@C$, and that of a whole complex $@C::$. A complex can span one surface and its two adjacent volumes.
- A rule that names no compartment applies wherever its reactants can meet. BioNetGen divides each bimolecular rate constant by the size of the compartment where the reaction occurs, so the rule's rate constant is intrinsic.
- Giving sizes as NAV, the number of molecules per μM , makes amounts numbers of molecules and keeps rate constants in concentration units.
- Transport rules move single molecules or whole complexes. Surface-to-surface species transport describes endocytosis.
- With nuclear transport, the STAT model shows nuclear accumulation of pSTAT and its reversal by SOCS feedback. With endocytosis, a small endosome concentrates released ligand enough to trap internalized receptors.

5.9 Exercises

1. **Location as a state.** Rewrite `two_volumes.bngl` without compartments, giving A and B a component `loc~cp~nu`. What rate constants must the two binding rules have to reproduce Figure 5.2? Check that the two models give identical trajectories.
2. **Nuclear size.** Vary `V_NU` in `stat_compartments.bngl` from 0.1 to 0.5 pL. How do the peak of nuclear pSTAT, measured in molecules and in concentration, and the adapted level of total pSTAT change? Explain which reactions are responsible.

3. **Constitutive import.** Some STATs enter the nucleus without phosphorylation (Reich and Liu 2006). Set `kimp0` to zero and compare the response with Figure 5.3. What does basal shuttling contribute to the response?
4. **Ligand depletion.** In `internalization.bngl`, lower `Lconc` to 0.01 nM and plot the ligand remaining in the medium. Explain why the cells deplete the medium at this dose and not at 0.1 nM. How does the result depend on `V_EC`, the volume of medium per cell?
5. **Signaling from endosomes.** Combine the two models of this chapter so that receptors are internalized and recycled and STAT can be phosphorylated by receptors on either membrane. Does internalization shorten or prolong the STAT response?
6. **Two anchors.** Add a co-receptor `N(1)` that binds a second site on the ligand and an internalization rule for complexes that contain `N`. Find the complexes that are internalized at the sum of two rates, and modify the rules so that every complex is internalized at a single rate.

5.10 Further reading

Harris et al. (2009) defines `cBNGL`, including its topology rules, volume scaling, and the four kinds of transport. Reich and Liu (2006) reviews the nuclear transport of STAT proteins. Lauffenburger and Linderman (1993) is the classic treatment of receptor binding and trafficking models.

6 Network-free simulation

Every simulation method in Part I begins by generating the reaction network implied by the rules. For the models of Chapter 2 this is quick, since the networks contain at most a few hundred reactions. For many models of signaling, however, the network is too large to generate, and for some it is infinite. Chapter 4 showed one example. A ligand with two binding sites and a receptor with two sites form chains of any length, and the network grows without limit unless the size of complexes is capped. Hlavacek and coworkers reviewed this combinatorial explosion, which they called the “complexity of complexes,” and argued for models built from molecular domains rather than lists of species (Hlavacek et al. 2003). In this chapter we describe how a rule-based model can be simulated without generating its network, use the method to study the aggregation of receptors by a trivalent ligand, and discuss when network-free simulation is and is not the right choice.

Learning goals

After this chapter you should be able to:

- explain why the network generated from a set of rules can be too large to use, and recognize models where this happens;
- describe how a network-free simulator applies rules directly to individual molecules;
- run network-free simulations with `bngsim` and analyze the complexes they produce;
- verify a network-free simulation against a network-based method;
- choose between network-based and network-free methods for a given model.

6.1 Rules without a network

Consider the stochastic simulation algorithm of Chapter 3. At each step it uses the propensity of every reaction, computed from the numbers of molecules of each species, chooses a reaction with probability proportional to its propensity, and updates the species counts. Efficient implementations recompute only the propensities that the last reaction changed, but the cost of a step still grows with the size of the network, and the list of reactions must be known in advance.

A network-free simulator applies the same logic to rules instead of reactions. The state of the system is a collection of individual molecules, each with its components, states, and bonds, and a complex is a set of molecules connected by bonds. The simulator proceeds as follows:

1. For each rule, it keeps a list of the molecules (or groups of molecules) that match each reactant pattern. The propensity of the rule is its rate constant times the number of ways the rule can be applied, which for a bimolecular rule is the product of the lengths of the two lists, corrected for symmetry.

2. It chooses the time of the next event from the total propensity of all rules and chooses a rule with probability proportional to its propensity, just as Gillespie's algorithm chooses a reaction.
3. It picks reactants at random from the lists of the chosen rule and checks any conditions that the lists do not capture, for example that two reactants of a $+$ rule belong to different complexes. If a condition fails, nothing happens, and the step is a null event.
4. Otherwise it applies the rule, changing bonds and states of the chosen molecules, and updates the lists for the molecules that changed and their neighbors.

Since the simulator never enumerates species or reactions, the cost of a step depends on the number of rules and on the size of the neighborhood that a rule changes, but not on the size of the network. Memory grows with the number of molecules rather than the number of species. The price is that every simulation is stochastic and tracks each molecule, so the cost grows with the number of molecules and the number of events, and, as we will see, with the size of the complexes.

The first network-free simulators appeared at about the same time. Danos, Feret, Fontana, and Krivine developed a scalable method for the Kappa language (Danos, Feret, Fontana, and Krivine 2007), and Yang, Monine, Faeder, and Hlavacek introduced a kinetic Monte Carlo method for BNGL, which they used to study multivalent ligand–receptor binding (Yang et al. 2008). Colvin and coworkers then wrote RuleMonkey, which implements an exact method that avoids null events (Colvin et al. 2010), and Michael Sneddon, the author, and Thierry Emonet wrote NFsim, which is designed for large models and supports functional rate laws (Sneddon et al. 2011). `bngsim` includes both NFsim and RuleMonkey.

6.2 Running a network-free simulation

Network-free simulators read the rules of a model from an XML file that BioNetGen writes with the action `writeXML()`. The model file itself needs no changes. In BNGL, the action `simulate({method=>"nf"})` writes the XML file and runs NFsim in one step. In Python, `bngsim` provides a session object that loads the XML file and runs the simulation:

```
xml = bng_xml("ch03", "stat_ssa_feedback")    # runs BioNetGen's writeXML()
with bngsim.NfsimSession(str(xml)) as nf:
    nf.initialize(seed=1)
    result = nf.simulate(0, 240, n_points=49)
```

The session returns the same `Result` object as the network-based methods, with the observables of the model as functions of time. Since NFsim tracks individual molecules, amounts must be numbers of molecules, as for the SSA, and rate constants must be in per-molecule units (Chapter 3).

The first test of any network-free simulation is a model whose network is small enough to simulate both ways. The stochastic STAT–SOCS model of Chapter 3 is such a model. Figure 6.1 compares averages over many runs of NFsim and of the SSA.

```
n_runs = 40
model = bngsim.Model.from_bngl(model_path("ch03", "stat_ssa_feedback"))
ssa = bngsim.Simulator(model, method="ssa")
```

```

model.reset()
t0 = time.perf_counter()
ssa_runs = ssa.run_replicates(n_runs, t_span=(0, 240), n_points=97, seed=1)
t_ssa = time.perf_counter() - t0

xml = str(bng_xml("ch03", "stat_ssa_feedback"))
t0 = time.perf_counter()
nf_runs = []
for seed in range(1, n_runs + 1):
    with bngsim.NfsimSession(xml) as nf:
        nf.initialize(seed=seed)
        nf_runs.append(nf.simulate(0, 240, n_points=97))
t_nf = time.perf_counter() - t0

t = ssa_runs[0].time
within = {}
for obs, label in [("pSTAT", "pSTAT (molecules)"), ("SOCS", "SOCS
↳ (molecules)"]):
    a = np.array([r.observables[obs] for r in ssa_runs])
    b = np.array([r.observables[obs] for r in nf_runs])
    se_a = a.std(0, ddof=1) / np.sqrt(n_runs)      # standard error of each
↳ mean
    se_b = b.std(0, ddof=1) / np.sqrt(n_runs)
    # fraction of time points (after t = 0) at which the two means differ by
    # less than two standard errors of their difference
    z = np.abs(a.mean(0) - b.mean(0))[1:] / np.sqrt(se_a**2 + se_b**2)[1:]
    within[obs] = np.mean(z < 2)
    fig, ax = plt.subplots(figsize=(4.2, 3.2))
    for x, se, style, name in [(a, se_a, "-", "SSA"), (b, se_b, "--",
↳ "Nfsim")]:
        line, = ax.plot(t, x.mean(0), style, label=name)
        ax.fill_between(t, x.mean(0) - 2 * se, x.mean(0) + 2 * se,
                        color=line.get_color(), alpha=0.15, lw=0)
    ax.set_xlabel("time (min)")
    ax.set_ylabel(label)
    ax.legend(loc="upper right")
    plt.show()

```

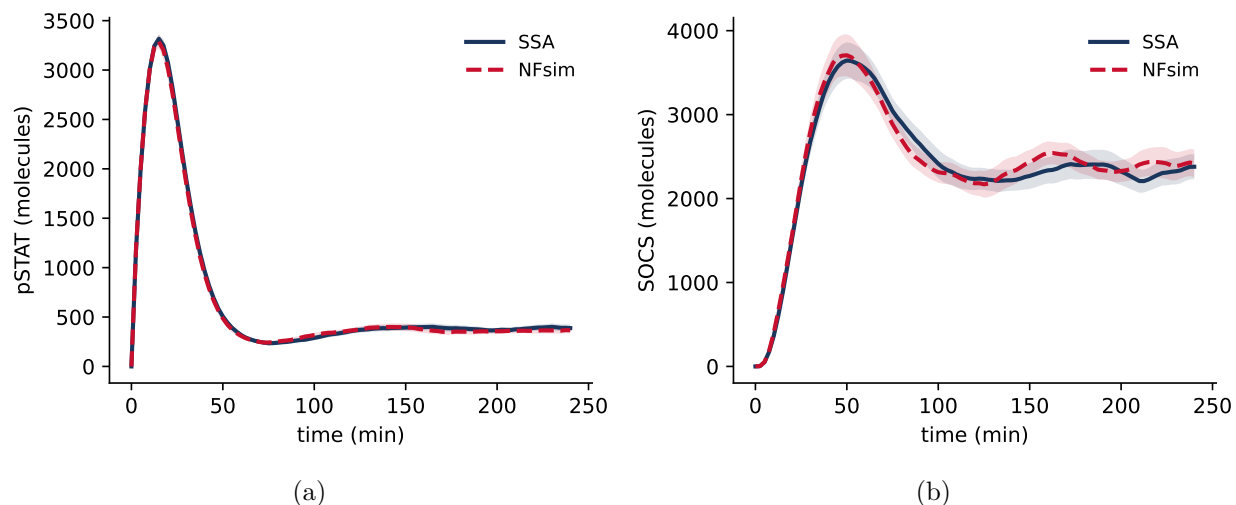


Figure 6.1: A network-free simulation agrees with a network-based one. Mean total pSTAT (a) and SOCS (b) from 40 stochastic simulations of the STAT–SOCS model of Chapter 3 with the SSA (solid) and with NFsim (dashed). Shaded bands show two standard errors of each mean.

The two methods agree to within the statistical error of the averages. At 99% of the time points for pSTAT and 97% for SOCS, the two means differ by less than two standard errors of their difference, about the 95% expected if the two methods sample the same process. They are not equally fast, however. On the computer that built this book, the 40 NFsim runs of this model took 12.5 s and the 40 SSA runs took 0.4 s, a factor of 34. For a network as small as this one, following individual molecules costs more than updating the counts of a few dozen species, and the network-based method is the better choice. Timings like these depend on the computer, the software versions, and the model, so they should be measured for the model at hand.

6.3 Receptor aggregation and the sol–gel transition

Consider now a model whose network cannot be generated. Many immune receptors, such as the high-affinity IgE receptor Fc RI on mast cells, signal when a multivalent antigen crosslinks them into aggregates. Goldstein and Perelson studied the simplest model that produces large aggregates, a ligand with three identical binding sites and a receptor with two (Goldstein and Perelson 1984). They showed that when crosslinking is strong enough, there is a range of ligand concentrations in which most receptors join a single aggregate whose size is limited only by the number of receptors on the cell. This is a sol–gel phase transition, like the one that turns a solution of polymers into a gel.

`tlbr.bngl` contains three rules: binding of free ligand to a receptor, and crosslinking by a ligand that already has one or two sites bound.

```
# Free ligand binds a receptor site
FreeBind:  L(r,r,r) + R(l) <-> L(r!1,r,r).R(l!1)      kp1, koff
```

```
# A ligand with one or two sites bound captures another receptor
Cross1:    L(r!+,r,r) + R(1) <-> L(r!+,r!1,r).R(1!1)      kp2, koff
Cross2:    L(r!+,r!+,r) + R(1) <-> L(r!+,r!+,r!1).R(1!1)    kp2, koff
```

The + in the crosslinking rules means that the ligand and the receptor must belong to different complexes, so the rules do not close rings. The binding rate constants are set by two dimensionless parameters introduced by Goldstein and Perelson, a ligand concentration $c = 3k_{p1}L_{\text{tot}}/k_{\text{off}}$ and a crosslinking constant $\beta = k_{p2}R_{\text{tot}}/k_{\text{off}}$. In the scans below we change c by changing k_{p1} , with the number of ligands fixed at 4200. Since each ligand has three sites and there are only 600 receptor sites, at most one ligand in seven can be bound, and the free ligand stays within 15% of the total. Changing k_{p1} at fixed ligand is therefore nearly equivalent to changing the amount of ligand at fixed affinity, which is what an experiment does, and the theory, which assumes that binding does not deplete the ligand, applies with small corrections.

These three rules generate an infinite network. Table 6.1 shows the network generated when complexes are limited to `max_agg` molecules, together with the time BioNetGen takes to generate it.

```
import pandas as pd
src = model_path("ch06", "tlbr").read_text()
src = src[: src.index("end model") + len("end model")]
rows = []
for max_agg in [3, 5, 7, 9]:
    with tempfile.TemporaryDirectory() as d:
        path = os.path.join(d, "tlbr.bngl")
        open(path, "w").write(src +
            ↪ f"\ngenerate_network({{overwrite=>1,max_agg=>{max_agg}}})\n")
        t0 = time.perf_counter()
        m = bngsim.Model.from_bngl(path, cache=False)
        rows.append({"max_agg": max_agg, "species": m.n_species,
            ↪ "reactions": m.n_reactions,
                "time (s)": round(time.perf_counter() - t0, 1)})
net = pd.DataFrame(rows).set_index("max_agg")
net
```

Table 6.1: Size of the network generated from `tlbr.bngl` as a function of `max_agg`, the largest complex allowed, and the time for `bngsim` to load the model, which includes parsing, network generation, and preparation for simulation.

	species	reactions	time (s)
max_agg			
3	5	6	0.2
5	10	28	0.3
7	19	92	1.2
9	36	280	8.2

The number of species grows slowly, but on the computer that built this book the time to load the model grows by a factor of about 3 for each molecule added to the largest complex. Part of the cost is that every complex a rule produces must be compared with the species already found to decide whether it is new, and the number of ways the rules can be applied grows with the size of the complexes. At this rate, a network that includes a gel of hundreds of molecules is far out of reach.

```

xml = str(bng_xml("ch06", "tlbr"))
Rtot = 300
koff = 0.01
c_values = [0.001, 0.01, 0.1, 0.84, 3, 10, 30, 100]
def t_equil(c):
    # Free ligand binds a receptor site at rate c*koff/3, so equilibration
    ↪ is
    # slow at low c. These run lengths were checked against longer runs.
    return max(1e4, 1.5 / (c * koff))
largest, largest_se, bonds_nf, drift = [], [], [], []
species_file = os.path.join(tempfile.gettempdir(), "tlbr_final.species")
for c in c_values:
    frac, b, b_half = [], [], []
    for seed in [1, 2, 3]:
        with bngsim.NfsimSession(xml) as nf:
            nf.set_param("cTot", c)
            nf.initialize(seed=seed)
            r = nf.simulate(0, t_equil(c), n_points=3)      # t = 0, t_end/2,
    ↪ t_end
            nf.save_species(species_file)
            frac.append(aggregate_sizes(species_file, "R").max() / Rtot)
            b.append(r.observables["Bonds"][-1])
            b_half.append(r.observables["Bonds"][1])
        largest.append(np.mean(frac))
        largest_se.append(np.std(frac, ddof=1) / np.sqrt(len(frac)))
        bonds_nf.append(np.mean(b))
        drift.append(abs(np.mean(b) - np.mean(b_half)) / np.mean(b))

# the same runs at c = 0.01, stopped early at 2000 s
early = []
for seed in [1, 2, 3]:
    with bngsim.NfsimSession(xml) as nf:
        nf.set_param("cTot", 0.01)
        nf.initialize(seed=seed)
        nf.simulate(0, 2000, n_points=2)
        nf.save_species(species_file)
        early.append(aggregate_sizes(species_file, "R").max() / Rtot)
early = np.mean(early)

# network-based ODEs with complexes capped at eight molecules
with tempfile.TemporaryDirectory() as d:

```

```

    path = os.path.join(d, "tlbr8.bngl")
    open(path, "w").write(src +
        ↪ "\ngenerate_network({overwrite=>1,max_agg=>8})\n")
    m8 = bngsim.Model.from_bngl(path)
    ode8 = bngsim.Simulator(m8, method="ode")
    c_fine = np.logspace(-3, 2, 31)
    bonds_ode = []
    for c in c_fine:
        m8.set_param("cTot", c)
        m8.reset()
        bonds_ode.append(ode8.run(t_span=(0, 10 * t_equil(c)),
            ↪ n_points=3).observables["Bonds"][-1])

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.errorbar(c_values, largest, yerr=largest_se, fmt="o-", capsize=2)
ax.set_xscale("log")
ax.set_xlabel("ligand concentration, $c$")
ax.set_ylabel("fraction of receptors\nin largest aggregate")
ax.set_ylim(0, 1)
plt.show()

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.semilogx(c_fine, np.array(bonds_ode) / Rtot, "-", color="#5B6770",
    ↪ label="ODE, max_agg = 8")
ax.semilogx(c_values, np.array(bonds_nf) / Rtot, "o", label="NFsim")
ax.set_xlabel("ligand concentration, $c$")
ax.set_ylabel("bonds per receptor")
ax.set_ylim(0, 2.05)
ax.legend(loc="lower right")
plt.show()

i84 = c_values.index(0.84)
ode_at = np.interp(np.log(c_values), np.log(c_fine), bonds_ode)
err = 1 - ode_at / np.array(bonds_nf)      # relative error of the capped
    ↪ ODEs
err84 = err[i84]

```

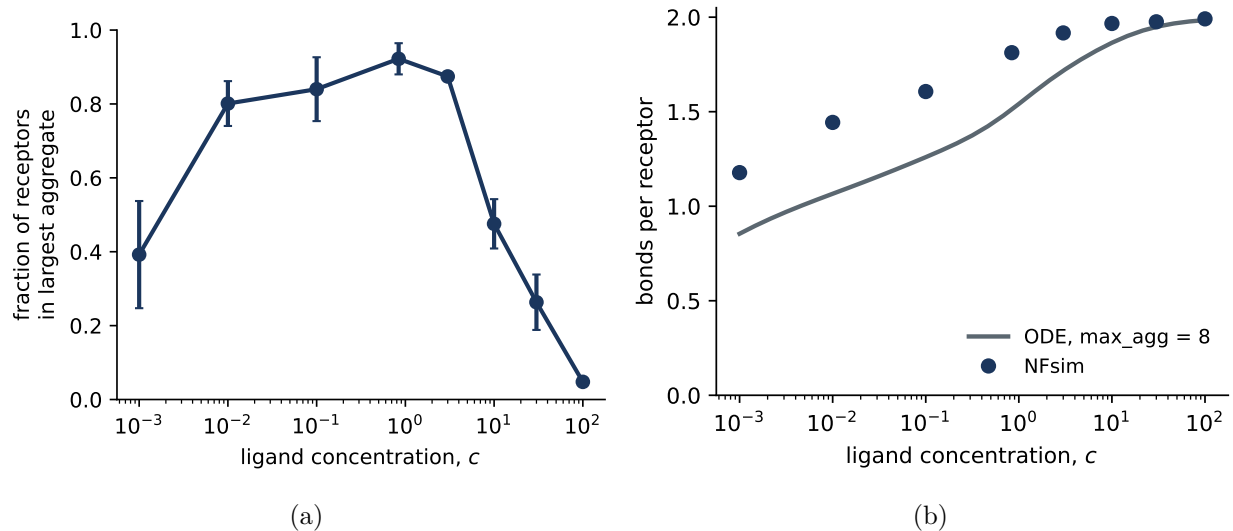


Figure 6.2: The sol-gel transition in the TLBR model with $\beta = 50$, simulated with NFsim. (a) Fraction of receptors in the largest aggregate at equilibrium as a function of the dimensionless ligand concentration c , with the standard error of the mean of three runs. (b) Ligand-receptor bonds from NFsim (points) and from the ODEs for the network with aggregates capped at eight molecules (line).

Figure 6.2 shows the transition. Over a wide range of ligand concentrations, a single aggregate forms that contains most of the receptors, for example 92% of them at $c = 0.84$. Since the simulation contains only 300 receptors, the size of the gel is limited by the size of the system, which is the finite-cell analog of the infinite aggregate of the theory. At high concentration each receptor site is occupied by a different ligand, and the largest aggregate shrinks to 5% of the receptors at $c = 100$. At low concentration there is too little ligand to link the receptors, but with strong crosslinking the gel persists to low doses, and at $c = 0.001$ the largest aggregate still contains 39% of them.

Each point in Figure 6.2 is an equilibrium value, and reaching equilibrium takes much longer at low concentration, since a free ligand binds a receptor site only at the rate $c k_{\text{off}}/3$, which is once every 3×10 s at $c = 0.001$. A run that stops too early underestimates the aggregates. At $c = 0.01$, for example, runs stopped at 2000 s, which is long enough at high concentration, give a largest aggregate of 10% of the receptors rather than 80%. The runs here last at least 10 s and longer at low concentration, and the mean number of bonds changes by at most 3% over the second half of each run. Since NFsim is stochastic, each point is the mean of three runs, and the error bars show its standard error. The scatter is largest at the edges of the gel phase, where the size of the largest aggregate fluctuates widely in time.

The network capped at eight molecules gives the wrong answer in this range. At $c = 0.84$ it underestimates the number of ligand-receptor bonds by 17%, because the cap forbids the aggregates that the system actually forms. At $c = 100$, where aggregates are small, the capped network and NFsim differ by 0.3%, but at $c = 0.001$, where large aggregates remain, the cap still causes an error of 27%. It is important to note that the size of this error cannot be known in advance. The only way to check that a cap is adequate is to compare with a larger cap or with a network-free simulation, as recommended in Chapter 4.

Monine and coworkers used network-free simulations of this kind to compare the TLBR model with measurements of a trivalent antigen binding IgE–Fc RI on mast cells (Monine et al. 2010). They found that the simple model predicts extensive aggregation at antigen doses where cells secrete strongly, and examined how steric constraints and the formation of rings would change the prediction. Such questions require keeping track of the structure of each aggregate, which a network-free simulation does automatically.

6.4 Cost and scaling

```
factors = [1, 2, 4, 8]
n_mol = 4500 * np.array(factors)
cost = {}
for c in [100, 0.84]:
    cost[c] = []
    for f in factors:
        with bngsim.NfsimSession(xml) as nf:
            nf.set_param("cTot", c)
            nf.set_param("Lig_tot", 4200 * f)
            nf.set_param("Rec_tot", 300 * f)
            nf.initialize(seed=1)
            t0 = time.perf_counter()
            nf.simulate(0, 500, n_points=3)
            cost[c].append(time.perf_counter() - t0)
fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.loglog(n_mol, cost[100], "o-", label="sol, $c$ = 100")
ax.loglog(n_mol, cost[0.84], "o-", label="gel, $c$ = 0.84")
ax.loglog(n_mol, cost[100][0] * n_mol / n_mol[0], "k--", lw=1)
ax.loglog(n_mol, cost[0.84][0] * (n_mol / n_mol[0])**2, "k--", lw=1)
ax.set_xlabel("number of molecules")
ax.set_ylabel("simulation time (s)")
ax.legend(loc="upper left")
plt.show()
slope_sol = np.polyfit(np.log(n_mol), np.log(cost[100]), 1)[0]
slope_gel = np.polyfit(np.log(n_mol), np.log(cost[0.84]), 1)[0]
```

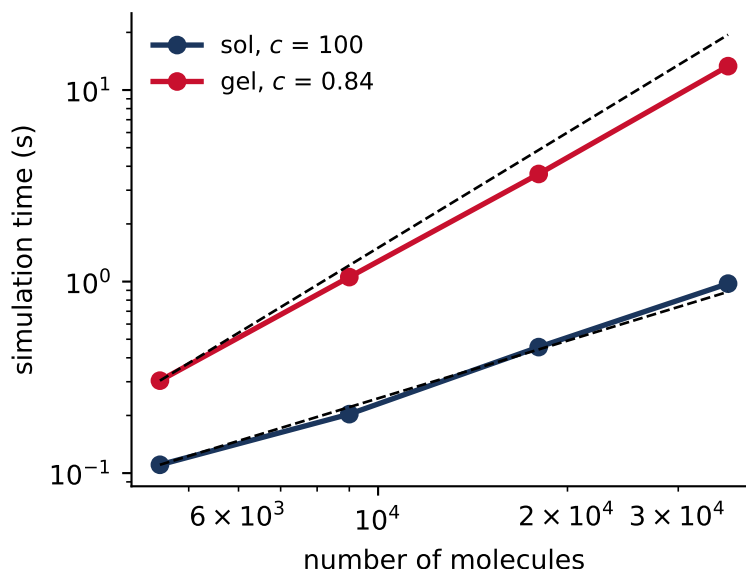


Figure 6.3: The cost of a network-free simulation depends on the size of the complexes as well as the number of molecules. Time to simulate 500 s of the TLBR model with NFsim, starting from free ligands and receptors, with the numbers of ligands and receptors multiplied by a common factor and β fixed, in the sol phase ($c = 100$) and in the gel phase ($c = 0.84$). The dashed lines have slopes 1 and 2.

Figure 6.3 shows how the cost of a network-free simulation grows with the size of the system. The rate of events grows in proportion to the number of molecules, and in the sol phase the cost of each event stays about the same, so the time to simulate a fixed interval grows linearly (the fitted slope is 1.1). In the gel phase the time grows about as the square of the number of molecules (slope 1.8) in these measurements. One expects faster than linear growth because the gel grows with the system, and some events require the simulator to traverse the whole complex that contains the reactants. When a bond in the gel breaks, for example, the simulator must determine whether the complex has split in two, and a $+$ rule must check that its reactants belong to different complexes. Network-free simulation thus trades the cost of the network, which depends on the number of possible species, for a cost that depends on the molecules actually present and on how they are connected. Table 6.2 summarizes when each kind of method is the better choice.

Table 6.2: Choosing a simulation method.

	Network-based (ODE, SSA, PSA)	Network-free (NFsim, RuleMonkey)
Requires	a finite network small enough to generate	rules only
Cost grows with Results	number of reactions deterministic (ODE) or stochastic	number of molecules and events stochastic only
Suited to	small and medium networks, fitting, steady states, sensitivity analysis	large or infinite networks, aggregation, polymerization, large models with many sites

	Network-based (ODE, SSA, PSA)	Network-free (NFsim, RuleMonkey)
Information on complexes	counts of each species	the structure of every complex

A practical consequence is that a modeler can often use both kinds of methods on the same model. The BNGL file is the same, so a model can be developed and fitted with a network capped at a moderate size and then checked with a network-free simulation, or simulated network-free in the regime where the cap fails.

6.5 Common mistakes

1. **Using network-free simulation by default.** When the network is small, NFsim is much slower than the SSA, and an ODE solution may suffice. Use network-free methods when the network is large or infinite.
2. **Too few runs.** Network-free results are stochastic. Average over enough runs to make the statistical error smaller than the effect being studied, and report it.
3. **Concentration units.** As for the SSA, amounts must be numbers of molecules and rate constants must be per molecule (Chapter 3).
4. **Rings formed by intermolecular rules.** A + between reactant patterns means that the reactants are in different complexes. bngsim enforces this by default, but the standalone NFsim program does not unless it is given the `-bscb` option. Write ring closure as a separate unimolecular rule, as in Chapter 4.
5. **Expensive observables.** `Species` observables require the simulator to track whole complexes, which can slow a simulation with very large aggregates. Use `Molecules` observables where they suffice.
6. **Unsupported features.** Some BNGL features available to network-based methods, such as compartments (Chapter 5), are not supported by every network-free simulator. Check the documentation of the simulator before relying on them.

6.6 Summary

- Rules for multivalent binding can generate networks that are too large to generate or infinite.
- A network-free simulator applies rules directly to individual molecules, choosing events with the same probabilities as Gillespie’s algorithm, so its cost does not depend on the size of the network.
- bngsim runs NFsim and RuleMonkey from the XML file that BioNetGen writes from the model, with no change to the model.
- A network-free simulation agrees with the SSA, within statistical error, on a model with a small network, but for that model it is much slower.
- The TLBR model shows a sol–gel transition in which most receptors join one aggregate. A capped network misses this behavior, while a network-free simulation captures it. Near the edges of the gel phase, and especially at low ligand concentration, equilibration is slow, and run lengths must be checked.

- Network-based and network-free methods can be used on the same BNGL model. Choose by the size of the network and the questions being asked.

6.7 Exercises

1. **RuleMonkey.** Repeat Figure 6.1 with `bngsim.RuleMonkeySession` in place of `NfsimSession`. Do the results agree? Which is faster for this model and for TLBR?
2. **Crosslinking strength.** Repeat the scan of Figure 6.2 with $\mu = 5$ and $\sigma = 500$, using run lengths long enough to reach equilibrium at every dose. How does the range of ligand concentrations in which a gel forms depend on μ ? Is there a value of μ below which no gel forms? Compare your results with the theory of Goldstein and Perelson (1984).
3. **Aggregate size distribution.** At $c = 0.01$, 0.84 , and 30 , plot the distribution of the number of receptors per aggregate from the final species of an NFsim run. How does the distribution change across the transition?
4. **When does a cap work?** For $c = 30$, find the smallest `max_agg` for which the ODE solution for the number of bonds agrees with NFsim to within 2%. Repeat for $c = 0.84$. What does the difference tell you?
5. **Rings.** Add a unimolecular rule that lets a bound ligand bind a free receptor site in the same aggregate, with rate constant `kring`. Simulate with NFsim for several values of `kring`. How do rings affect the size of the largest aggregate? Compare with the discussion in Monine et al. (2010).

6.8 Further reading

Sneddon et al. (2011) describes NFsim, its algorithm, and its performance on large models. Yang et al. (2008) and Colvin et al. (2010) describe the kinetic Monte Carlo method and RuleMonkey. Danos, Feret, Fontana, and Krivine (2007) describes network-free simulation for the Kappa language. Goldstein and Perelson (1984) is the classic theory of aggregation by a trivalent ligand, and Monine et al. (2010) compares it with experiment.

7 Analyzing models

A simulation answers one question: what does the model do for one set of parameters and one initial condition? Most questions about a model are of a different kind. We wish to know which parameters control a response, how the response depends on the strength of the stimulus, how quickly the system settles and whether it oscillates on the way, and whether it can settle in more than one state. The tools that answer these questions are older than rule-based modeling and apply to any model that can be written as a set of ODEs. In this chapter we apply them to the cytokine–STAT–SOCS model of Chapter 1, extending the exercise on the shape of the pSTAT pulse at the end of that chapter, and to a variant of the circuit with positive feedback that acts as a switch with memory.

Learning goals

After this chapter you should be able to:

- find the conservation laws of a reaction network and explain what they imply for its steady states;
- compute a steady state and determine its stability from the eigenvalues of the Jacobian;
- relate eigenvalues to the timescales and the damped oscillations of a response;
- compute dose–response curves with parameter scans;
- compute normalized sensitivity coefficients and use them to find the parameters that control a response;
- recognize bistability and hysteresis, and trace them with a continuation scan.

7.1 Conservation laws

Consider the model `stat_step6_feedback.bngl` from Chapter 1. Its twelve species are not independent. Ligand, receptor, STAT, and the SOCS promoter are neither made nor destroyed, only moved from one complex to another, so the total amount of each is fixed by the initial conditions. `bngsim` finds these **conservation laws** from the stoichiometry matrix when it loads a model:

```
model = bngsim.Model.from_bngl(model_path("ch01", "stat_step6_feedback"))
laws = model.conservations_laws
for row, total in zip(np.array(laws["coefficients"]), laws["constants"]):
    terms = [(name if c == 1 else f"{c:g} {name}")
              for c, name in zip(row, model.species_names) if c != 0]
    print(" + ".join(terms), "=", total)
```

$G() + GSp() = 1.0$

$$\begin{aligned} L() + LR() + LRS() + LRSp() + LRSoCS() &= 10.0 \\ S() + LRS() + LRSp() + Sp() + GSp() &= 10.0 \\ R() + LR() + LRS() + LRSp() + LRSoCS() &= 1.0 \end{aligned}$$

Each law is a combination of species whose total does not change, which is the statement that a left null vector of the stoichiometry matrix exists (Chapter 1). With 4 conservation laws, the state of the system is determined by 8 independent variables, and the remaining species follow from the totals. Conservation laws matter in two ways. First, they confine the system to the set of states that share its totals, so that the steady states available to it depend on the initial conditions only through the totals. Second, they make the Jacobian of the full system singular, which is why steady-state solvers work in the reduced space of independent variables.

The first point does not mean that every initial condition with the same totals reaches the same steady state, since a set of states with fixed totals can contain more than one steady state. Whether it does is a property of the particular model, which must be checked. For this model, we may start with the receptors already bound to ligand and the STAT already phosphorylated, at the same totals, and compare the result with the steady state reached from the usual initial condition (computed in the next section):

```
sim = bngsim.Simulator(model, method="ode")
def steady_pSTAT(initial):
    model.reset()
    for name, value in initial.items():
        model.set_concentration(name, value)
    d = sim.steady_state(method="newton").to_dict()
    return d["Sp()"] + d["LRSp()"] + d["GSp()"]

print(f"free ligand and receptor: {steady_pSTAT({}):\ .5f} uM")
print(f"bound, STAT active:      "
      f"{steady_pSTAT({'L()': 9, 'R()': 0, 'LR()': 1, 'S()': 0, 'Sp()':
      ↪ 10}):\ .5f} uM")
```

```
free ligand and receptor: 0.32247 uM
bound, STAT active:      0.32247 uM
```

The two agree, and so does every other initial condition with these totals that we have tried. In Section 7.5, by contrast, we will meet a model in which two initial conditions at the same parameters reach different steady states.

7.2 Steady states and their stability

A **steady state** is a state at which every rate of change is zero. The simplest way to find one is to simulate until nothing changes, and bngsim does this, followed by a Newton iteration that refines the result:

```

model.reset()
ss = sim.steady_state(method="newton")
ss_pSTAT = ss.to_dict()["Sp()"] + ss.to_dict()["LRSp()"] +
    ↪ ss.to_dict()["GSp()"]
print(f"converged: {ss.converged}; steady-state pSTAT = {ss_pSTAT:.4f} uM")

```

```
converged: True; steady-state pSTAT = 0.3225 uM
```

The steady state is the adapted level of Chapter 1. Whether the system actually returns to it after a small disturbance is a separate question, which is answered by linearizing the equations about the steady state. Let $\mathbf{x}^*$ be the steady state and $\delta = \mathbf{x} - \mathbf{x}^*$ a small deviation. To first order,

$$\frac{d\delta}{dt} = J \delta, \quad J_{ij} = \left. \frac{\partial f_i}{\partial x_j} \right|_{\mathbf{x}^*},$$

where J is the **Jacobian** matrix. The solution is a sum of terms $e^{\lambda t}$, one for each eigenvalue λ of J . The Jacobian of the full system has a zero eigenvalue for each conservation law, which describes a change of the totals rather than a return to the steady state, so the eigenvalues that matter are those of the Jacobian of the independent variables. bngsim reports these. If every one of them has a negative real part, every small deviation decays and the steady state is locally stable. If any has a positive real part, some deviation grows and the steady state is unstable. If the largest real part is zero, the linear analysis is inconclusive.

The binding reaction of Chapter 1 shows how this works in the simplest case. With the two conservation laws, the state is determined by a single variable, $x = [\text{LR}]$, which obeys

$$\frac{dx}{dt} = f(x) = k_{\text{on}}(L_0 - x)(R_0 - x) - k_{\text{off}}x.$$

The Jacobian of this reduced system is a 1×1 matrix,

$$J = \frac{df}{dx} = -k_{\text{on}}[(L_0 - x) + (R_0 - x)] - k_{\text{off}} = -(k_{\text{on}}([L] + [R]) + k_{\text{off}}),$$

and its single eigenvalue is negative for any concentrations, so the equilibrium is stable. A small deviation decays at the rate $k_{\text{on}}([L] + [R]) + k_{\text{off}}$, which is the relaxation rate given in Chapter 1. Had we kept all three species as variables, the Jacobian would be a 3×3 matrix with this eigenvalue and two zero eigenvalues, one for each conservation law. The STAT-SOCS model is treated the same way, with more variables.

```

-0.0293 + 0.0496i (1/min)
-0.0293 - 0.0496i (1/min)
-0.1860 + 0.0173i (1/min)
-0.1860 - 0.0173i (1/min)
-2.0003 + 0.0000i (1/min)
-2.0667 + 0.0000i (1/min)

```

```

-8.8205 + 0.0000i    (1/min)
-13.5605 + 0.0000i    (1/min)

```

All 8 eigenvalues, one for each independent variable, have negative real parts, so the adapted state is stable. The real parts give the timescales of relaxation, $\tau = -1/\text{Re } \lambda$, and the imaginary parts give the frequencies of oscillation. The fastest mode decays with a time constant of 4 s, which is comparable to the time for a ligand-bound receptor to capture STAT, $1/(k_{\text{Son}}[\text{S}]) \approx 6$ s with 10 μM STAT, the fastest process in the model. The slowest pair of eigenvalues, $-0.029 \pm 0.050i \text{ min}^{-1}$, is complex. A complex pair describes a **damped oscillation**, with period $2\pi/|\text{Im } \lambda| = 127$ min and decay time $1/|\text{Re } \lambda| = 34$ min. This is the slow approach to the adapted level that we saw in Chapter 1, in which pSTAT undershoots and then recovers as SOCS overshoots and decays.

```

model.reset()
r = sim.run(t_span=(0, 600), n_points=6001)
p = r.observables["pSTAT"]

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.plot(r.time, p)
ax.axhline(ss_pSTAT, color="#5B6770", ls="--", lw=1)
ax.set_xlim(0, 300)
ax.set_xlabel("time (min)")
ax.set_ylabel("total pSTAT ( $\mu\text{M}$ )")
plt.show()

fig, ax = plt.subplots(figsize=(4.2, 3.2))
dev = np.abs(p - ss_pSTAT)
ax.semilogy(r.time, dev)
t_ref = 150
i_ref = np.searchsorted(r.time, t_ref)
env = dev[i_ref - 600:i_ref + 600].max() * np.exp(slow.real * (r.time -
    ↪ t_ref))
ax.semilogy(r.time[r.time > 50], env[r.time > 50], "k--", lw=1)
ax.set_ylim(1e-7, 10)
ax.set_xlabel("time (min)")
ax.set_ylabel("|pSTAT - steady state| ( $\mu\text{M}$ )")
plt.show()

```

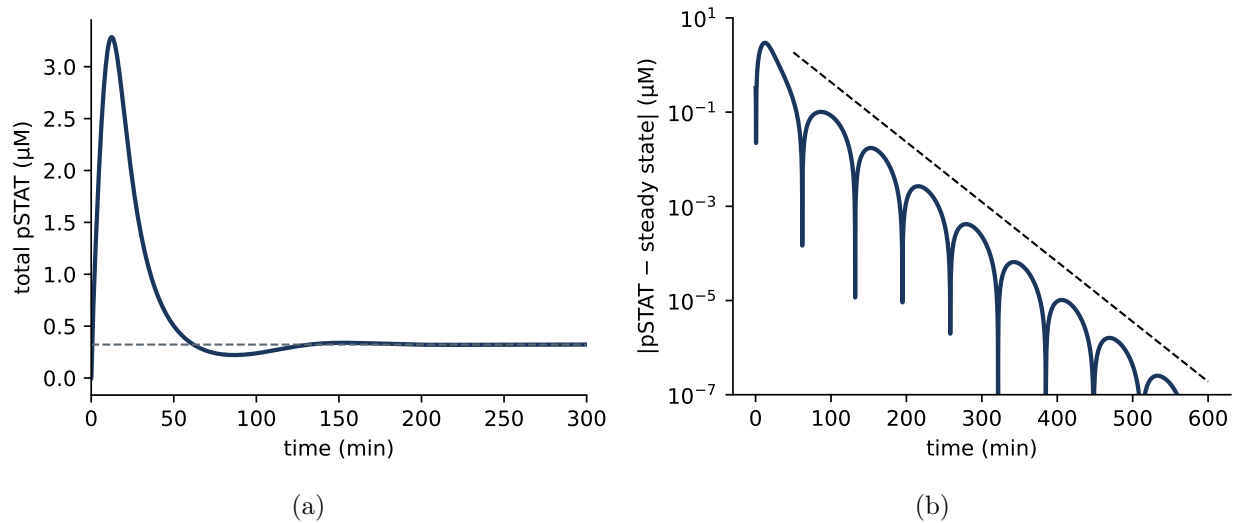


Figure 7.1: Relaxation to the adapted state is governed by the slowest eigenvalues. (a) Total pSTAT after stimulation (solid) and the steady state from the Newton solver (dashed). (b) Distance of pSTAT from its steady-state value on a logarithmic scale. At late times the envelope decays as $e^{\text{Re} \lambda t}$ for the slowest complex pair (dashed), and the dips mark the times at which the damped oscillation crosses the steady state.

Figure 7.1 b shows that this description is accurate well before the system reaches its steady state. After the first 50 min, the distance from the steady state shrinks along the envelope predicted by the slowest pair of eigenvalues, and it passes through zero roughly every half period.

7.3 Dose–response curves

The response of a cell to a signal depends on the strength of the signal, and a **dose–response curve** shows how. A parameter scan runs one simulation for each value of a parameter. `bngsim` provides it as a single call, which resets the initial state before each run:

```
L_values = np.logspace(-4, 1, 26)
model.reset()
scan = sim.parameter_scan("L0", list(L_values), t_span=(0, 600),
    ↪ n_points=6001)
peak = np.array([res.observables["pSTAT"].max() for res in scan])
adapted = np.array([res.observables["pSTAT"][-1] for res in scan])
model.set_param("L0", 10)

fig, ax = plt.subplots(figsize=(5.0, 3.4))
ax.loglog(L_values, peak, "o-", ms=4, label="peak")
ax.loglog(L_values, adapted, "o-", ms=4, label="adapted level")
lo = L_values < 0.02
ax.loglog(L_values[lo], peak[0] * L_values[lo] / L_values[0] * 2, "k--",
    ↪ lw=1)
```

```

ax.loglog(L_values[lo], adapted[0] * np.sqrt(L_values[lo] / L_values[0]) *
    ↪ 2, "k:", lw=1)
ax.set_xlabel("ligand, $L_0$ (μM)")
ax.set_ylabel("total pSTAT (μM)")
ax.legend(loc="lower right")
plt.show()

def ec50(y):
    half = 0.5 * y[-1]
    return np.exp(np.interp(half, y, np.log(L_values)))
ec50_peak, ec50_adapted = ec50(peak), ec50(adapted)
slope_peak = np.polyfit(np.log(L_values[:6]), np.log(peak[:6]), 1)[0]
slope_adapted = np.polyfit(np.log(L_values[6:12]), np.log(adapted[6:12]),
    ↪ 1)[0]

```

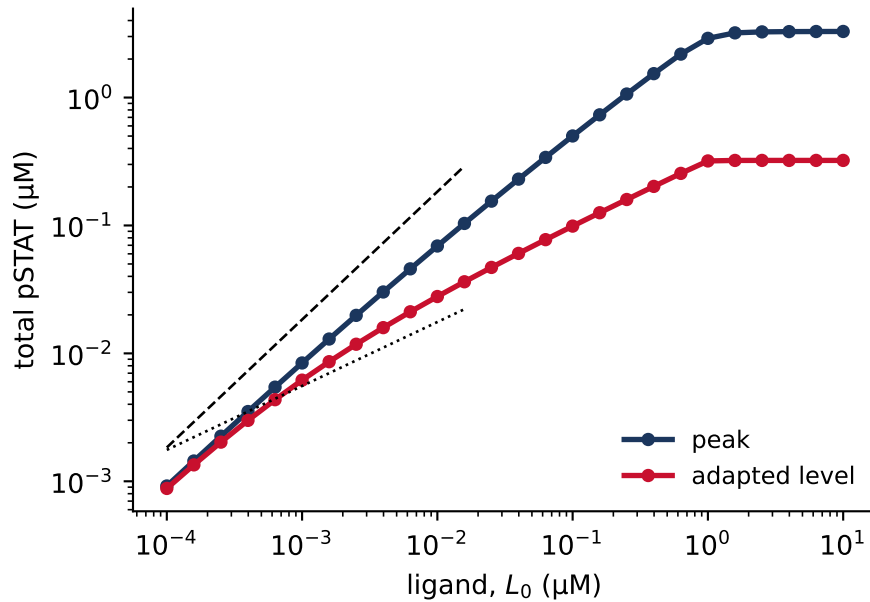


Figure 7.2: Dose–response curves of the cytokine–STAT–SOCS model. Peak and adapted levels of total pSTAT as functions of the initial ligand concentration. The dashed lines have slopes 1 and 1/2.

Figure 7.2 shows the peak and the adapted level of pSTAT over five decades of ligand concentration. Both saturate once the ligand exceeds the receptor concentration of 1 μM . Since the ligand binds tightly ($K_D = 0.1 \mu\text{M}$), nearly every ligand molecule finds a receptor until the receptors run out. Below saturation the two curves differ in shape. The peak grows in proportion to the dose (log–log slope 0.96 at the lowest doses), while the adapted level grows more slowly, with a slope of 0.62 in the middle of the range. The concentrations of ligand that give half of the maximal response, the EC_{50} values, are 0.43 μM for the peak and 0.26 μM for the adapted level. Negative feedback thus compresses the adapted response. Between 0.001 and 0.1 μM ligand, a hundredfold range, the peak changes 60-fold and the adapted level only 16-fold. The next section explains where the square root

comes from.

7.4 Sensitivity analysis

An exercise in Chapter 1 asked how doubling each parameter changes the time of the pSTAT peak, its height, and the adapted level. **Sensitivity analysis** makes this question systematic. The **normalized sensitivity coefficient** of an output y to a parameter p is

$$S_p^y = \frac{\partial \ln y}{\partial \ln p} = \frac{p}{y} \frac{\partial y}{\partial p},$$

the percentage change in y per percentage change in p for small changes. A coefficient of 1 means that y is proportional to p , a coefficient of -1 that it is inversely proportional, and a coefficient of 0 that p has no effect. Normalized coefficients have no units, so they can be compared across parameters of different kinds. For example, when ligand is in large excess over receptor, the fraction of receptors bound at equilibrium is $y = [L]/([L] + K_D)$. Since $\ln y = \ln[L] - \ln([L] + K_D)$, the sensitivity to the dissociation constant is $S_{K_D}^y = -K_D/([L] + K_D)$. With 1 μM ligand and $K_D = 0.1 \mu\text{M}$, $S_{K_D}^y \approx -0.09$, so a 10% increase in K_D lowers the fraction bound by about 0.9%. The sensitivity is $-1/2$ when $[L] = K_D$, and it approaches -1 at low ligand, where the fraction bound is proportional to $1/K_D$. The idea goes back to metabolic control analysis, in which Heinrich and Rapoport, and independently Kacser and Burns, asked how the flux through a metabolic pathway is shared among its enzymes (Heinrich and Rapoport 1974).

We compute the coefficients for the three properties of the pulse by central differences, changing each parameter by $\pm 1\%$. The time of the peak is found by fitting a parabola through the three highest points of the simulated curve, so that it is not limited by the spacing of the output times.

```
def pulse():
    model.reset()
    res = sim.run(t_span=(0, 600), n_points=6001)
    y, t = res.observables["pSTAT"], res.time
    i = np.argmax(y)
    a, b, c = np.polyfit(t[i-1:i+2], y[i-1:i+2], 2)
    t_peak = -b / (2 * a)
    return np.array([t_peak, c - b**2 / (4 * a), y[-1]])

base = pulse()
params = list(model.primary_param_names)
S = {}
h = 0.01
for name in params:
    v = model.get_param(name)
    model.set_param(name, v * (1 + h)); up = pulse()
    model.set_param(name, v * (1 - h)); down = pulse()
    model.set_param(name, v)
    S[name] = (np.log(up) - np.log(down)) / (np.log(1 + h) - np.log(1 - h))
```

```

order = sorted(params, key=lambda n: -abs(S[n][2]))
y = np.arange(len(order))
fig, ax = plt.subplots(figsize=(6.0, 4.6))
w = 0.27
for k, (label, color) in enumerate([("time of peak", "#5B6770"), ("peak
↳ height", "#1F7A8C"), ("adapted level", "#C8102E")]):
    ax.barh(y + (k - 1) * w, [S[n][k] for n in order], height=w,
    ↳ color=color, label=label)
ax.set_yticks(y)
ax.set_yticklabels(order, fontsize=8)
ax.invert_yaxis()
ax.axvline(0, color="black", lw=0.6)
ax.set_xlabel("normalized sensitivity")
ax.legend(loc="lower right", fontsize=8)
plt.show()

# steady-state sensitivities from the solver, for comparison
model.reset()
ss_s = sim.steady_state(method="newton", sensitivity_params=params)
k_obs = ss_s.observable_names.index("pSTAT")
S_ss = {n: ss_s.sensitivities_observables[k_obs][j] * model.get_param(n) /
↳ base[2]
        for j, n in enumerate(params)}
initial = [n for n in params if n.endswith("0")]
rates = [n for n in params if n not in initial]
max_diff = max(abs(S_ss[n] - S[n][2]) for n in rates)
half = [n for n in params if abs(abs(S[n][2]) - 0.5) < 0.05]
tmax = max(abs(S[n][0]) for n in params)

```

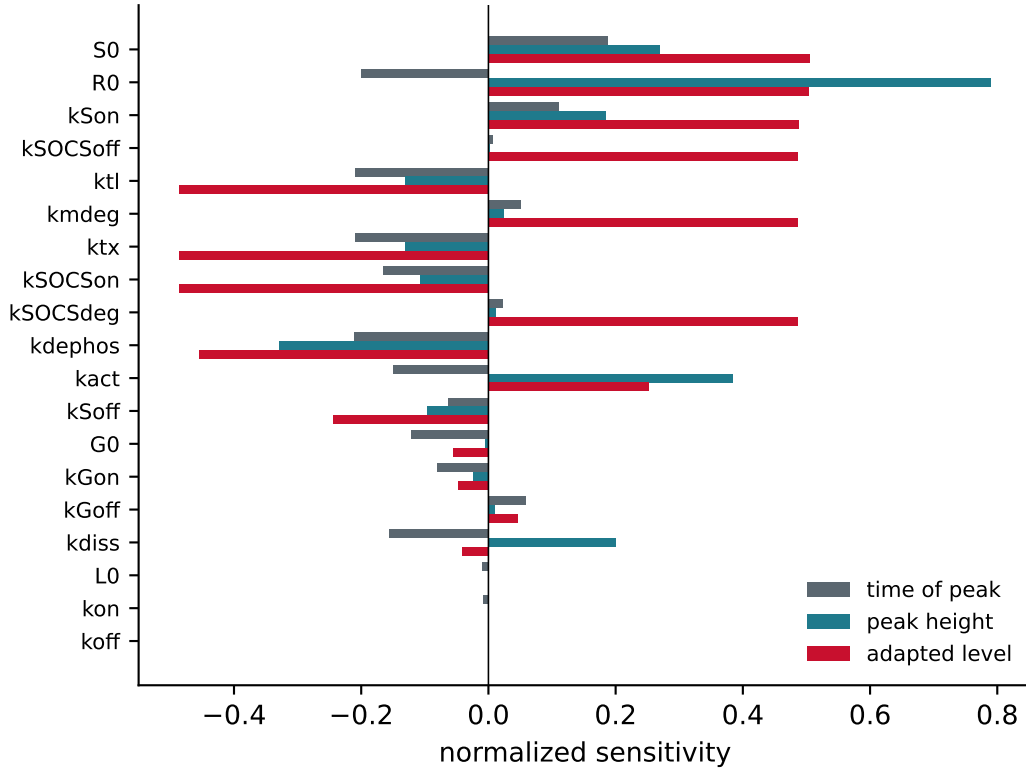


Figure 7.3: Normalized sensitivities of the pSTAT pulse to each parameter of the cytokine–STAT–SOCS model, computed by central differences with $\pm 1\%$ changes. Parameters are ordered by their effect on the adapted level.

Figure 7.3 shows the result. The three properties of the pulse depend on the parameters in different ways. The height of the peak depends most on the amount of receptor and on the rates of STAT phosphorylation and dephosphorylation, as one would expect for the early, feedback-free part of the response. The time of the peak is moderately sensitive to many parameters, none with a coefficient larger than 0.21 in magnitude. It is set by the race between the rise of pSTAT and the accumulation of SOCS, so it depends both on the parameters of the feedback loop and on those that make pSTAT rise faster, such as the amount of receptor. For the rate constants, the steady-state sensitivities computed by the solver, which differentiates the steady-state equations directly, agree with the finite-difference values for the adapted level to within $1e-05$. The initial amounts are an exception. They affect the steady state only through the totals of the conservation laws, and the solver’s parameter sensitivities hold those totals fixed, so for the initial amounts the finite-difference values are the correct ones.

For the adapted level, 10 of the 19 parameters have coefficients close to $+1/2$ or $-1/2$. Consider the feedback loop at the adapted state, where nearly all ligand-bound receptors carry SOCS. The rate of STAT phosphorylation is proportional to the small fraction of receptors without SOCS, which is inversely proportional to the amount of SOCS. The amount of SOCS, in turn, is proportional to pSTAT, since the promoter is far from saturated and SOCS is made at a rate proportional to the occupied promoter. Setting production of pSTAT equal to its removal therefore gives an equation of the form $[\text{pSTAT}] \propto A/[\text{SOCS}] \propto A/[\text{pSTAT}]$, where A collects the parameters that favor signaling. The adapted level is thus proportional to $\sqrt{A}$, and every parameter that enters A or

the SOCS–pSTAT proportionality as a simple factor has a sensitivity of $\pm 1/2$. The argument rests on three conditions, that SOCS occupies nearly all ligand-bound receptors, that the promoter is far from saturation, and that production and removal are linear in the relevant species. At the adapted state of this model, SOCS occupies 96% of the ligand-bound receptors and pSTAT occupies 14% of the promoters, so the first two conditions hold, and the other reactions of the model are first order or bimolecular with one partner in excess. The same argument explains the square-root dependence on dose in Figure 7.2, over the range of doses in which the conditions hold. Under these conditions, the negative feedback loop halves the sensitivity of the adapted state to most of its parameters, which is one way in which feedback can make a response less sensitive to variation in the amounts of its components.

Local sensitivities describe the effect of small changes around one set of parameter values. They can change sign or magnitude elsewhere in parameter space, and they say nothing about combinations of large changes. Global methods, which sample the whole range of plausible parameter values, and the related question of which parameters can be determined from data are taken up in Part III.

7.5 Switches and memory

Negative feedback, as in the STAT–SOCS loop, can make a response less sensitive to its parameters, although with delays it can also produce oscillations. Positive feedback can create two stable states, so that a cell responds to a graded signal with an all-or-none switch. Ferrell and Machleder found such a switch in the maturation of *Xenopus* oocytes, where individual oocytes respond to progesterone in an all-or-none fashion because the MAPK cascade is embedded in a positive feedback loop (Ferrell and Machleder 1998). Xiong and Ferrell later showed that the same loop gives the oocytes a memory, keeping the cascade active after the stimulus is removed (Xiong and Ferrell 2003).

A positive feedback loop is present in JAK–STAT signaling as well, since the gene for STAT1 is itself induced by STAT1. `stat_autoloop.bngl` is a minimal model of this loop. Receptor activity enters as the rate constant of STAT phosphorylation, `kact`, and STAT is made at a rate that is a Hill function of pSTAT, using a functional rate law (Chapter 4):

```
begin functions
  fsyn() = ksyn0 + vmax*pSTAT^n/(K^n + pSTAT^n)
end functions
begin reaction rules
  Activate:  S(Y~0) <-> S(Y~P)    kact, kdephos
  Synthesis: 0 -> S(Y~0)           fsyn()
  Degrade:   S() -> 0              kdeg
end reaction rules
```

To map the steady states as a function of the signal, we use a **continuation scan**, `bifurcate`. Unlike a parameter scan, it does not reset the state between points. Each run starts from the end of the previous one, so the scan follows a branch of steady states as the parameter changes. Scanning `kact` up and then down reveals whether the branch depends on the direction of the scan.

```

auto = bngsim.Model.from_bngl(model_path("ch07", "stat_autoloop"))
asim = bngsim.Simulator(auto, method="ode")
k_values = np.logspace(-2, 0, 41)
auto.set_param("kact", k_values[0])
auto.reset()
up = asim.bifurcate("kact", list(k_values), t_span=(0, 5000), n_points=3)
down = asim.bifurcate("kact", list(k_values[::-1]), t_span=(0, 5000),
    ↪ n_points=3)
S_up = np.array([res.observables["STAT"][-1] for res in up])
S_down = np.array([res.observables["STAT"][-1] for res in down])[:, -1]
k_on = k_values[np.argmax(S_up > 2)]
k_off = k_values[np.argmax(S_down > 2)]

# Exact steady states. At steady state pSTAT = kact*S0/(kdephos + kdeg) and
# synthesis balances degradation, fsyn(pSTAT) = kdeg*STAT, so each value of
# pSTAT gives one value of kact and of total STAT, including on the unstable
# branch, which no forward simulation converges to.
P = {k: auto.get_param(k) for k in ["kdephos", "ksyn0", "vmax", "K", "n",
    ↪ "kdeg"]}
sp = np.logspace(-3, 1.5, 3000)
fsyn = P["ksyn0"] + P["vmax"] * sp**P["n"] / (P["K"]**P["n"] + sp**P["n"])
ratio = fsyn / (P["kdeg"] * sp) - 1
keep = ratio > 0
k_branch = (P["kdephos"] + P["kdeg"]) / ratio[keep]
sp, S_branch = sp[keep], fsyn[keep] / P["kdeg"]
stable = []
for k, p_ss, S_ss in zip(k_branch, sp, S_branch):
    auto.set_param("kact", k)
    J = np.asarray(auto.jacobian(np.array([S_ss - p_ss, p_ss])))
    stable.append(np.linalg.eigvals(J).real.max() < 0)
stable = np.array(stable)
turn = np.where(np.diff(np.sign(np.diff(k_branch)))) != 0)[0] + 1
k_fold_off, k_fold_on = sorted(k_branch[turn])

fig, ax = plt.subplots(figsize=(4.2, 3.2))
in_range = (k_branch > k_values[0]) & (k_branch < k_values[-1])
for mask, style in [(stable, "-"), (~stable, "--")]:
    ax.semilogx(np.where(mask & in_range, k_branch, np.nan), S_branch,
    ↪ style, color="0.4", lw=1)
ax.semilogx(k_values, S_up, "o", ms=4, label="increasing")
ax.semilogx(k_values, S_down, "x", ms=5, label="decreasing")
ax.axvspan(k_fold_off, k_fold_on, color="#D39B1E", alpha=0.15, lw=0)
ax.set_xlabel("receptor activity, kact (min-1)")
ax.set_ylabel("steady-state STAT (μM)")
ax.legend(loc="upper left")
plt.show()

```

```

# memory: hold kact = 0.3, apply a strong pulse, return to 0.3
auto.set_param("kact", 0.3)
auto.reset()
seg1 = asim.run(t_span=(0, 100), n_points=101)
auto.set_param("kact", 1.0)
seg2 = asim.run(t_span=(100, 160), n_points=61)
auto.set_param("kact", 0.3)
seg3 = asim.run(t_span=(160, 500), n_points=341)
t_all = np.concatenate([seg1.time, seg2.time, seg3.time])
s_all = np.concatenate([seg1.observables["STAT"], seg2.observables["STAT"],
    ↪ seg3.observables["STAT"]])
auto.set_param("kact", 0.1)

fig, ax = plt.subplots(figsize=(4.2, 3.2))
ax.axvspan(100, 160, color="#D39B1E", alpha=0.2, lw=0)
ax.plot(t_all, s_all)
ax.set_xlabel("time (min)")
ax.set_ylabel("total STAT (μM)")
plt.show()

# stability of each branch at kact = 0.3, from the eigenvalues of the
    ↪ Jacobian
def eigenvalues_at(state):
    return np.linalg.eigvals(np.asarray(auto.jacobian(np.asarray(state))))
auto.set_param("kact", 0.3)
auto.reset()
asim.run(t_span=(0, 5000), n_points=3)
eig_low = eigenvalues_at(auto.get_state())
auto.set_state(np.array([1.0, 9.0]))          # start near the upper branch
asim.run(t_span=(0, 5000), n_points=3)
eig_high = eigenvalues_at(auto.get_state())
auto.set_param("kact", 0.1)
max_re = max(np.real(eig_low).max(), np.real(eig_high).max())

```

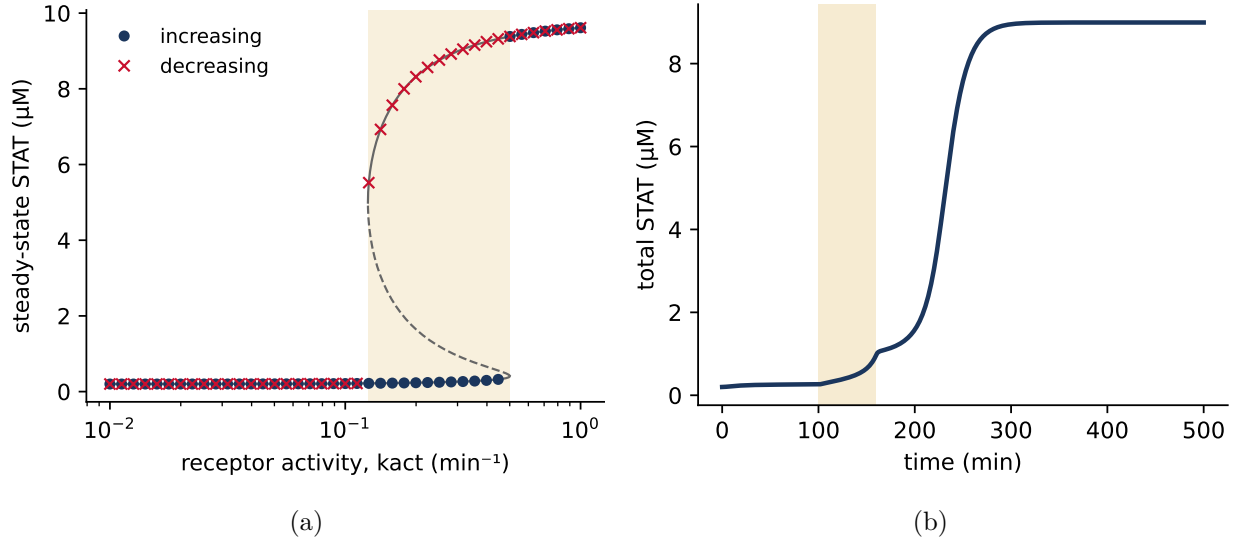


Figure 7.4: A positive feedback loop makes a switch with memory. (a) Steady-state STAT as a function of receptor activity, from continuation scans with increasing (circles) and decreasing (crosses) k_{act} , and from the exact solution of the steady-state equations, with stable (solid) and unstable (dashed) branches. Between the two saddle-node bifurcations (shaded) the system is bistable. (b) A transient signal switches the system to the upper state. With k_{act} held at 0.3 min⁻¹ inside the bistable range, a 60-min pulse of strong signal (shaded) moves the system to the upper state, where it remains after the signal returns to 0.3.

Figure 7.4 a shows the result. When the signal increases from a low value, STAT stays near its basal level until k_{act} reaches about 0.50 min⁻¹, and then jumps to a level about 48 times higher. When the signal decreases from a high value, STAT stays high until k_{act} falls below about 0.13 min⁻¹. Between these thresholds the system has two stable steady states, and which one it occupies depends on its history. This dependence on history is called **hysteresis**. The eigenvalues confirm that both states are stable, since at $k_{act} = 0.3$ min⁻¹ the largest real part of any eigenvalue on either branch is -0.049 min⁻¹. Between the two stable states lies a third, unstable, steady state, which separates the initial conditions that go to one stable state from those that go to the other. A forward simulation does not converge to it, so a continuation scan cannot show it. For this two-species model, however, the steady-state equations can be solved exactly, and the solution traces all three branches in Figure 7.4 a. The thresholds are **saddle-node bifurcations**, the points at which the unstable state collides with one of the stable states and both disappear (Tyson et al. 2003). The exact solution places them at 0.125 and 0.500 min⁻¹. A continuation scan locates them only to within the spacing of its grid, and near a bifurcation it converges slowly, since the eigenvalue that governs the approach to the steady state passes through zero there. Thresholds from a scan should therefore be checked with a finer grid and longer runs, or, when the equations allow it, by solving for the steady states directly.

Figure 7.4 b shows the practical consequence. A cell that sees a moderate signal remains in the low state, but a transient strong signal moves it to the high state, where it stays after the signal returns to its moderate level. In this deterministic model it stays there for as long as k_{act} remains in the bistable range, and the circuit thus remembers the strong signal, as the oocytes of Xiong and Ferrell do. In a stochastic model, fluctuations can return the cell to the low state, although such

transitions become rare as the numbers of molecules grow. Whether STAT1 autoregulation creates memory of this kind in real cells is not established by this model, which is deliberately minimal. The model does show what properties such a memory would require: feedback that is both strong and cooperative enough, here through a Hill coefficient of 2, to produce two stable states over the relevant range of signals.

7.6 Common mistakes

1. **Stopping a simulation too early.** A response that has not reached steady state can look flat on a linear scale. Check by plotting the distance from the steady state on a logarithmic scale, as in Figure 7.1, or use a steady-state solver.
2. **Forgetting the conservation laws.** The steady state depends on the totals of conserved quantities. Changing the initial amount of a species in a scan changes a total, and changing a species that is not in the seed species may have no effect at all.
3. **Reading too much into local sensitivities.** A sensitivity describes small changes around one point in parameter space and can be very different elsewhere.
4. **Finite-difference steps of the wrong size.** Steps that are too large give the sensitivity of a nonlinear function averaged over a range; steps that are too small are swamped by the numerical error of the solver. Changes of about 1% with tight solver tolerances are a reasonable starting point, and the solver's own sensitivities avoid the problem.
5. **Missing bistability.** A parameter scan that resets the initial state for every point follows only one branch. Use a continuation scan in both directions, or start simulations from several initial conditions, whenever a model contains positive feedback.

7.7 Summary

- Conservation laws reduce the number of independent variables and confine the system to states with fixed totals. Whether those totals determine a single steady state must be checked for each model.
- A steady state is locally stable if all eigenvalues of the Jacobian of the independent variables have negative real parts. The real parts give the timescales of relaxation, and the imaginary parts of complex pairs give the periods of damped oscillations.
- The slowest complex pair of the STAT–SOCS model accounts for the slow, oscillatory approach of pSTAT to its adapted level.
- Parameter scans give dose–response curves. Negative feedback compresses the adapted response, which grows roughly as the square root of the dose.
- Normalized sensitivity coefficients identify the parameters that control a response. The adapted pSTAT level has sensitivities of about $\pm 1/2$ to most parameters, a consequence of the feedback loop when SOCS occupies most receptors and the promoter is far from saturation.
- Positive feedback can create two stable states. Continuation scans in both directions reveal hysteresis, and a transient signal can switch such a system to its upper state, where a deterministic model keeps it while the signal stays in the bistable range.

7.8 Exercises

1. **Conservation laws.** Find the conservation laws of `stat_rbm1_rules.bngl` from Chapter 2 with `model.conservations_laws`. How many are there, and how do they compare with those of the network model?
2. **Timescales.** Compute the eigenvalues at the adapted state of `stat_step6_feedback.bngl` as `kSOCSdeg` is varied from 0.01 to 0.5 min⁻¹. How do the period and decay time of the slowest mode change? Is there a value at which the oscillation becomes sustained?
3. **Sensitivity of the peak.** Explain from the structure of the model why increasing the amount of receptor raises the peak and makes it occur earlier, while increasing the rate of SOCS translation lowers the peak and also makes it occur earlier.
4. **Testing the square-root law.** Use the model to check the argument for the $\pm 1/2$ sensitivities. Plot the fraction of ligand-bound receptors carrying SOCS and the fraction of occupied promoters at the adapted state as functions of `L0`. Where does the argument break down, and how do the sensitivities change there?
5. **Removing the switch.** Repeat Figure 7.4 a with Hill coefficients `n` = 1, 1.5, and 3. For which values is there a bistable range? How does its width depend on `n`?
6. **Global sensitivity.** Draw 200 sets of parameters for `stat_step6_feedback.bngl`, each parameter varied by a random factor between 1/3 and 3 on a logarithmic scale. Plot the adapted level against each parameter. Which parameters still stand out, and does the square-root pattern survive?

7.9 Further reading

Tyson et al. (2003) is an accessible introduction to the dynamics of signaling motifs, including switches and oscillators, and Tyson and Novák (2010) extends it. Ferrell and Machleder (1998) and Xiong and Ferrell (2003) are the classic experimental studies of a bistable switch with memory. Heinrich and Rapoport (1974) introduced the sensitivity coefficients of metabolic control analysis.

8 Energy-based rule-based modeling

Consider a receptor with distinct ligand-binding sites **a** and **b**. We can write a binding rule for each site without specifying the occupancy of the other, as in Chapter 2. If occupation of one site changes the affinity of the other, however, the corresponding rate constants must satisfy a constraint. Both orders of ligand addition reach the same doubly occupied receptor, so both must give the same equilibrium free-energy change.

Energy-based modeling expresses this constraint by assigning free energies to molecular structures and using energy differences to determine ratios of forward and reverse rate constants. Sekar, Hogg, and Faeder implemented this approach in BioNetGen (Sekar et al. 2016). We retain rules to specify which structural changes can occur; energy patterns determine how the local environment changes their rates. Thus energies become another way to parameterize the models developed in the preceding chapters.

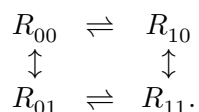
Learning goals

After this chapter you should be able to:

- test whether reversible binding rates satisfy detailed balance;
- construct energy patterns for binding and cooperative interactions;
- derive equilibrium constants and interpret the kinetic parameters of an energy-based rule;
- account for symmetry when comparing generated rates with equilibrium measurements;
- distinguish equilibrium constraints from the constraints on a chemically driven signaling model;
- choose energy and kinetic parameters for estimation from data.

8.1 A steady state with a circulating flux

Denote the receptor states by R_{00} , R_{10} , R_{01} , and R_{11} , where the subscripts give the occupancy of **a** and **b**. Addition of ligand follows either path around the square:



Each forward step consumes a free ligand molecule, which is omitted from the diagram. Let $K_{A0} = k_{A0}^+/k_{A0}^-$ be the association constant for site **a** when **b** is empty, and define the other constants similarly. **Detailed balance** means that the forward and reverse fluxes of each elementary reversible reaction are equal at equilibrium. Applying that condition along both paths gives

$$\frac{[R_{11}]}{[R_{00}][L]^2} = K_{A0}K_{B1} = K_{B0}K_{A1}.$$

The ligand concentration cancels when we compare the paths. We therefore require $K_{A0}K_{B1}/(K_{B0}K_{A1}) = 1$ for passive binding, whatever the total amounts of receptor and ligand.

Consider the following context-dependent rules. The parameter **boost** increases association at **b** when **a** is occupied, while all other rates retain their original values.

```
begin parameters
  R0      1
  L0      1
  kon     1
  koff    0.1
  boost   10
end parameters
```

```
begin reaction rules
  A_first: R(a,b) + L(r) <-> R(a!1,b).L(r!1)      kon, koff
  A_second: R(a,b!+) + L(r) <-> R(a!1,b!+).L(r!1)    kon, koff
  B_first:  R(a,b) + L(r) <-> R(a,b!1).L(r!1)      kon, koff
  B_second: R(a!+,b) + L(r) <-> R(a!+,b!1).L(r!1)    boost*kon, koff
end reaction rules
```

The cycle ratio is **boost**. The rules still conserve receptor and ligand, and an ODE simulation still approaches a steady state. To test equilibrium we must examine reaction fluxes as well as concentrations:

```
bad = load("two_site_inconsistent")
bad_result = run(bad)
o = bad_result.observables
kon, koff, boost = (bad.get_param(p) for p in ("kon", "koff", "boost"))
currents = {
  "A binds first": kon*o["Lfree"]*o["R00"] - koff*o["R10"],
  "B binds second": boost*kon*o["Lfree"]*o["R10"] - koff*o["R11"],
  "A leaves": koff*o["R11"] - kon*o["Lfree"]*o["R01"],
  "B leaves": koff*o["R01"] - kon*o["Lfree"]*o["R00"],
}
bad_current = float(currents["A binds first"][-1])
print("Final clockwise fluxes (uM/min):")
for name, current in currents.items():
  print(f"{name:18s} {current[-1]:.6g}")
```

```
Final clockwise fluxes (uM/min):
A binds first      0.0115861
```

B binds second	0.0115861
A leaves	0.0115861
B leaves	0.0115861

The final currents agree at approximately $0.0116 \mu\text{M}/\text{min}$. Receptor enters and leaves each state at equal rates, which explains the constant concentrations, but ligand addition and removal continue to circulate around the cycle. The binding model contains no fuel or other driving process to sustain that current. The parameter choice therefore fails a thermodynamic test even though it passes conservation and steady-state tests.

Given an explicit chemical driving process, a circulating current could be physical. We return to that case in Section 8.5. For the present binding model, the cycle constraint determines how a change in affinity at one site must be accompanied by a change at the other.

8.2 Free energies of molecular structures

Write $g = G/(RT)$ for a dimensionless free energy, where R is the gas constant and T is the absolute temperature. For our receptor, assign binding contributions g_A and g_B , and an additional contribution g_{AB} when both sites are occupied:

$$\begin{aligned} g_{00} &= 0, & g_{10} &= g_A, \\ g_{01} &= g_B, & g_{11} &= g_A + g_B + g_{AB}. \end{aligned}$$

The free-energy change on binding at **a** is g_A when **b** is empty and $g_A + g_{AB}$ when **b** is occupied. Binding at **b** gives g_B or $g_B + g_{AB}$. Both routes to R_{11} therefore sum to the same energy.

An **energy pattern** assigns a free-energy contribution to each occurrence of a specified molecular structure. BioNetGen sums the contributions of all matching patterns to obtain the energy of a species. The patterns for the receptor are:

```
begin energy patterns
  R(a!1).L(r!1)  gA
  R(b!1).L(r!1)  gB
  R(a!+,b!+)     gAB
end energy patterns
```

The first pattern counts an **a**-ligand bond, the second counts a **b**-ligand bond, and the third counts a receptor with both sites bound. The third pattern need not name the ligand molecules since their identities are already restricted by the binding rules. In a model where several kinds of partner can bind those sites, the pattern would need additional context if the cooperative energy were specific to one partner.

Receptor structure	Matching contributions	Relative energy
$R(a,b)$	none	0
$R(a!1,b).L(r!1)$	a bond	g_A
$R(a,b!1).L(r!1)$	b bond	g_B

Receptor structure	Matching contributions	Relative energy
R(a!1,b!2).L(r!1).L(r!2)	both bonds and double occupancy	$g_A + g_B + g_{AB}$

8.2.1 Concentration references and affinity

For association of two distinct molecules, use a concentration reference C_{ref} to form a dimensionless equilibrium constant. If the dissociation constant is K_D , then

$$g_{\text{bind}} = \ln \left(\frac{K_D}{C_{\text{ref}}} \right), \quad \frac{[RL]}{[R][L]} = \frac{e^{-g_{\text{bind}}}}{C_{\text{ref}}}.$$

A more negative binding energy corresponds to a smaller dissociation constant. The numerical value of the energy depends on the concentration reference. A conventional thermodynamic standard state and a computational reference expressed in micromolar units require a corresponding conversion of the association energy; the predicted physical affinity must agree.

Our BNGL models use micromolar concentrations, minutes, and $C_{\text{ref}} = 1 \mu\text{M}$. The parameter **Cref** records that convention; BioNetGen’s **Arrhenius** expression uses the numerical rates in those units and does not read **Cref** as a separate unit conversion. Keep **Cref** fixed when running these files. Changing concentration units requires converting the bimolecular rates and association energies together.

```

begin parameters
  R0      1
  L0      1
  Cref    1                # concentration reference (uM); keep at 1 here
  KD_A    0.1              # isolated-site dissociation constant (uM)
  KD_B    0.1
  omega   10               # enhancement of affinity when the other site
↪ binds
  gA      ln(KD_A/Cref)
  gB      ln(KD_B/Cref)
  gAB     -ln(omega)
  kA      1                # kinetic scale (1/min)
  kB      1
  EA      -ln(kA)          # log kinetic scale relative to 1/min
  EB      -ln(kB)
  phi     0                # allocation of energy change to forward rate
end parameters

```

Here $g_{AB} = -\ln \omega$, where ω is the factor by which occupation of one site increases the affinity of the other. Given $\omega > 1$, the doubly occupied receptor receives a favorable energy contribution. Thus $K_{D,A|B} = K_{D,A}/\omega$ and $K_{D,B|A} = K_{D,B}/\omega$. The same cooperative factor applies in both directions, as required by the cycle constraint. An unfavorable interaction has $\omega < 1$.

These are effective free energies of the molecular states represented in the model. Solvent effects and unresolved conformations can contribute to them. An energy pattern therefore need not correspond to a bond energy calculated from an atomic structure, and its parameter can be estimated from affinity measurements.

8.3 From energy differences to rates

Equilibrium constrains a ratio of rates. To predict a time course we also need a kinetic scale. BioNetGen supplies the reversible rate law `Arrhenius(phi,Ea)`, which computes numerical forward and reverse rates from the dimensionless energy change Δg :

$$k_+ = e^{-E_a - \phi \Delta g}, \quad k_- = e^{-E_a + (1-\phi) \Delta g}.$$

Their ratio is $e^{-\Delta g}$. The parameter E_a determines their common scale, while ϕ determines how an energy change is divided between the forward and reverse rates. We usually restrict ϕ to the interval from zero to one in this phenomenological parameterization.

For the association convention used here, write the dimensional rates as

$$k_{\text{on}} = \frac{k_*}{C_{\text{ref}}} e^{-\phi \Delta g}, \quad k_{\text{off}} = k_* e^{(1-\phi) \Delta g}.$$

The BNGL parameter `EA = -ln(kA)` sets the numerical kinetic scale relative to the time reference in the file. Taking a logarithm of a rate requires that reference; `EA` is an effective parameter rather than a measured activation energy. In our concentration convention, the numerical association rate produced by `Arrhenius` has units of $\mu\text{M}^{-1}\text{min}^{-1}$ and the dissociation rate has units of min^{-1} .

```
begin reaction rules
  BindA: R(a) + L(r) <-> R(a!1).L(r!1)  Arrhenius(phi,EA)
  BindB: R(b) + L(r) <-> R(b!1).L(r!1)  Arrhenius(phi,EB)
end reaction rules
```

Each rule leaves the other receptor site unspecified. BioNetGen generates the relevant reactions, evaluates their energy differences, and gives them the corresponding rates. At $\phi = 0$, the association rate is independent of context, while the dissociation rate changes with affinity. At $\phi = 1$, the dissociation rate is independent of context. Intermediate values change both rates. None of these choices changes the equilibrium constant.

8.3.1 Checking the generated dynamics

We can express the same model with ordinary mass-action rules by specifying the occupancy of the other site and deriving each rate explicitly:

```
begin reaction rules
  A_first: R(a,b) + L(r) <-> R(a!1,b).L(r!1)      konA, koffA
```

```

A_second: R(a,b!+) + L(r) <-> R(a!1,b!+).L(r!1) konAB, koffAB
B_first:  R(a,b) + L(r) <-> R(a,b!1).L(r!1)      konB, koffB
B_second: R(a!+,b) + L(r) <-> R(a!+,b!1).L(r!1) konBA, koffBA
end reaction rules

```

The file `two_site_rates.bngl` defines the rates using the equations above. For example, `konAB` and `koffAB` use $\Delta g = g_A + g_{AB}$. Comparing the generated time courses checks that the energy rules give the intended kinetics, including the context-dependent dissociation rates.

```

energy = load("two_site_energy")
explicit = load("two_site_rates")
energy_result = run(eenergy)
explicit_result = run(explicit)
max_difference = max(
    np.max(np.abs(eenergy_result.observables[n] -
        ↪ explicit_result.observables[n]))
    for n in energy.observable_names
)
plt.plot(eenergy_result.time, eenergy_result.observables["R11"], label="energy
    ↪ patterns")
plt.plot(explicit_result.time, explicit_result.observables["R11"],
    "--", label="explicit rates")
plt.xlabel("time (min)")
plt.ylabel("doubly occupied receptor (μM)")
plt.xlim(0, 300)
plt.legend()
plt.show()

```

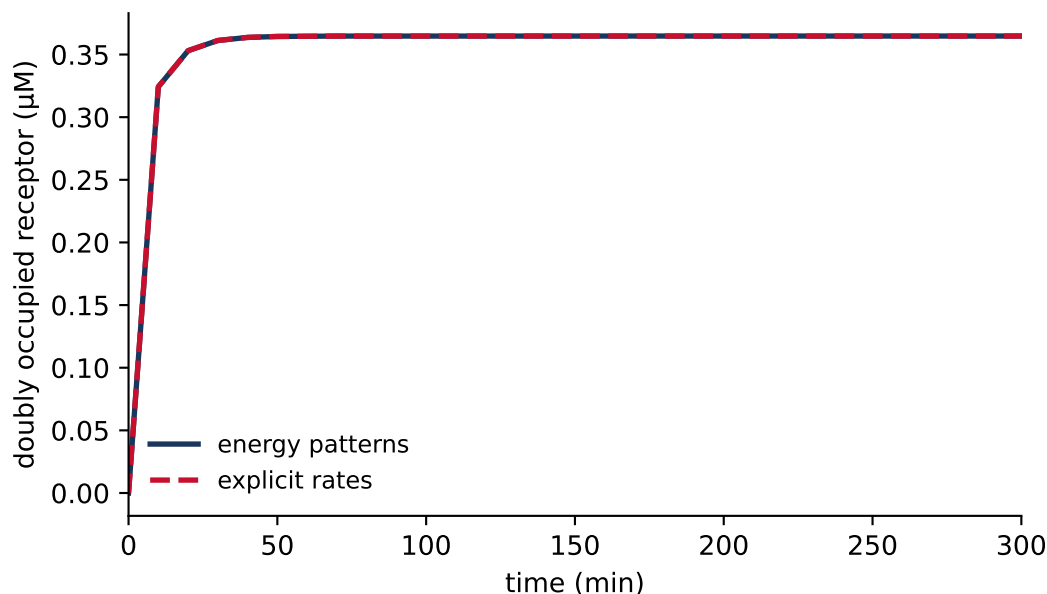


Figure 8.1: Doubly occupied receptor in the energy-based and explicit-rate models. Both start with free receptor and free ligand.

The maximum absolute difference across the observables is $2.2\text{e-}14 \mu\text{M}$ for this calculation. At the final state, every reversible reaction also balances individually:

```
e = energy_result.observables
ka = energy.get_param("kA")
ga, gab = energy.get_param("gA"), energy.get_param("gAB")
first_flux = ka*e["Lfree"][-1]*e["R00"][-1] - ka*np.exp(ga)*e["R10"][-1]
second_flux = ka*e["Lfree"][-1]*e["R01"][-1] -
    ↪ ka*np.exp(ga+gab)*e["R11"][-1]
print(f"A binding, B empty:      {first_flux:.3g} uM/min")
print(f"A binding, B occupied: {second_flux:.3g} uM/min")
```

```
A binding, B empty:      8.86e-16 uM/min
A binding, B occupied: -1.07e-15 uM/min
```

The calculation uses the default $\phi = 0$; the identical site parameters give the same balance at **b**. The numerical residuals reflect integration tolerance. Energy-based parameterization has removed the circulating flux of Section 8.1 by enforcing the relation between both binding paths.

```
for phi in (0, 0.5, 1):
    r = run(energy, end=300, n_points=601, phi=phi)
    plt.plot(r.time, r.observables["R11"], label=f"$\\phi$ = {phi:g}")
plt.xlim(0, 300)
plt.xlabel("time (min)")
plt.ylabel("doubly occupied receptor (μM)")
```

```
plt.legend()
plt.show()
energy.set_param("phi", 0)
```

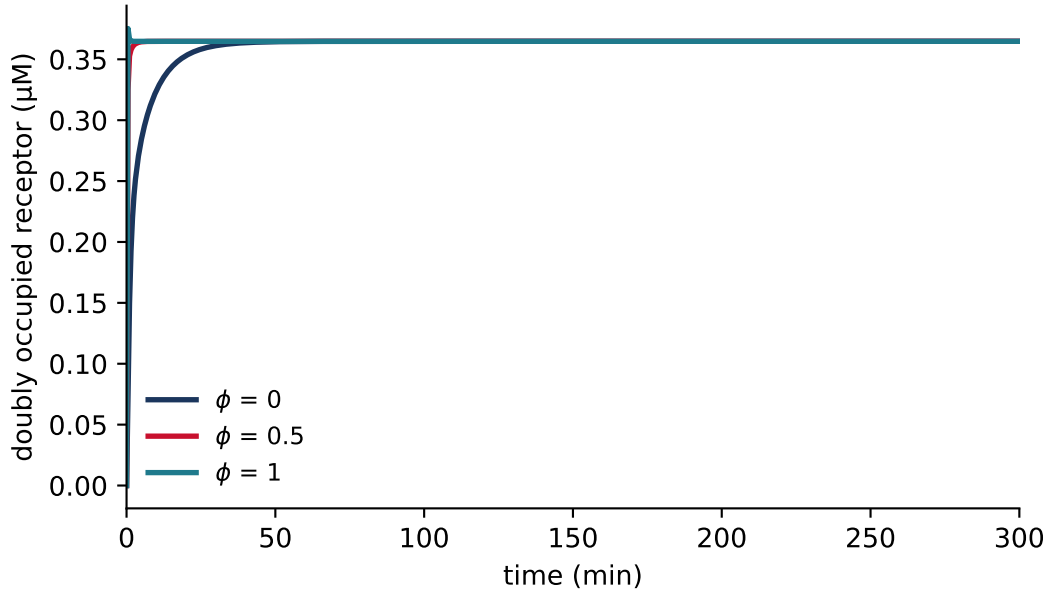


Figure 8.2: Changing the allocation parameter changes relaxation while preserving the equilibrium occupancy.

Since binding is favorable in this example, increasing ϕ increases both rates at fixed E_a . The receptor therefore approaches the same equilibrium more rapidly (Figure 8.2). An affinity measurement alone cannot determine which allocation gives the observed relaxation.

8.4 Cooperativity and a binding polynomial

At a specified free ligand concentration, define $x_A = [L]/K_{D,A}$ and $x_B = [L]/K_{D,B}$. The equilibrium weights of the receptor states are 1, x_A , x_B , and $\omega x_A x_B$. Their sum is the **binding polynomial**

$$Z = 1 + x_A + x_B + \omega x_A x_B.$$

Dividing each weight by Z gives its receptor fraction. In particular, $f_{11} = \omega x_A x_B / Z$. These fractions use free ligand concentration. In a closed experiment the measured input is often total ligand, so we must also impose

$$L_{\text{tot}} = [L] + R_{\text{tot}} \frac{x_A + x_B + 2\omega x_A x_B}{Z}.$$

We can check this prediction against a scan of the rule-based model, using the simulated free ligand concentration to evaluate the polynomial:

```

totals = np.logspace(-2, 1.3, 25)
polynomial_errors = []
for omega in (1, 10):
    free, simulated, predicted = [], [], []
    for total in totals:
        r = run(energy, L0=total, omega=omega)
        ell = last(r, "Lfree")
        xa = ell/energy.get_param("KD_A")
        xb = ell/energy.get_param("KD_B")
        z = 1 + xa + xb + omega*xa*xb
        free.append(ell)
        simulated.append(last(r, "R11")/energy.get_param("R0"))
        predicted.append(omega*xa*xb/z)
    polynomial_errors.append(np.max(np.abs(np.array(simulated)-predicted)))
    ell = np.logspace(np.log10(min(free)), np.log10(max(free)), 200)
    xa, xb = ell/energy.get_param("KD_A"), ell/energy.get_param("KD_B")
    curve = omega*xa*xb/(1 + xa + xb + omega*xa*xb)
    line, = plt.semilogx(ell, curve, label=f"$\\omega$ = {omega:g}")
    plt.semilogx(free, simulated, "o", color=line.get_color(), markersize=3)
plt.xlabel("free ligand ( $\mu$ M)")
plt.ylabel("fraction doubly occupied")
plt.legend()
plt.show()

```

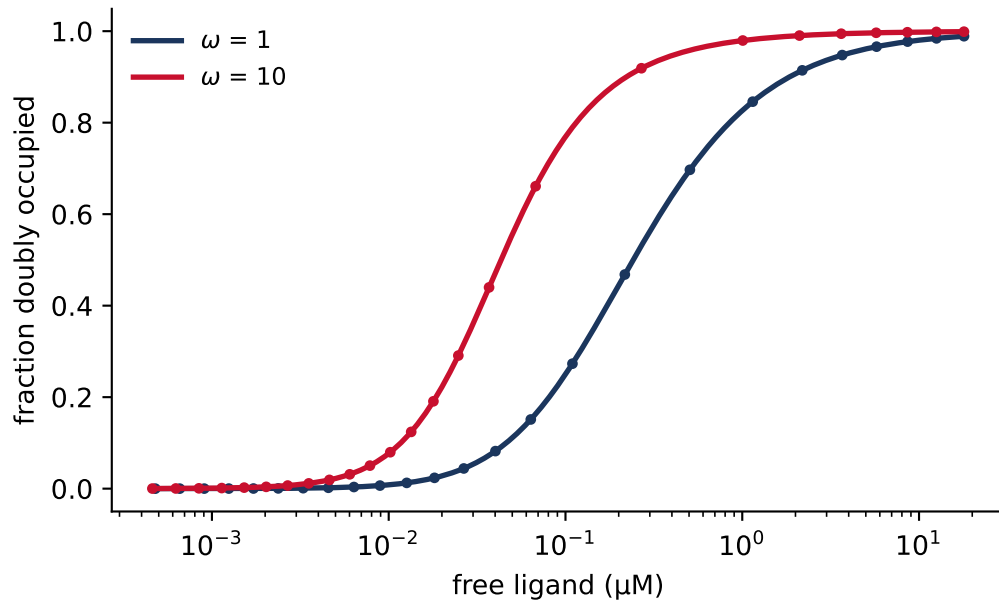


Figure 8.3: Equilibrium double occupancy with independent sites and favorable coupling. Lines are binding-polynomial predictions evaluated at the simulated free ligand concentration; points are ODE results.

Favorable coupling moves double occupancy toward lower free ligand concentrations. The largest difference between the simulated and predicted fractions is 1.6e-08. Using total ligand in x_A and x_B would violate the mass balance when receptor binding depletes the free pool.

8.4.1 A reciprocal interaction between RAF dimerization and drug binding

Poulikakos and coworkers observed that RAF inhibitors can increase ERK signaling in cells with wild-type BRAF, and connected this response to RAF dimers (Poulikakos et al. 2010). Consider an illustrative assembly model in which inhibitor binding favors RAF dimerization. We represent each RAF protomer by a dimerization site **d** and an inhibitor site **i**.

```
begin energy patterns
  RAF(d!1).RAF(d!1)      gDim
  RAF(i!1).I(r!1)        gDrug
  RAF(d!1,i!+).RAF(d!1) gCouple
end energy patterns
```

```
begin reaction rules
  Dimerize: RAF(d) + RAF(d) <-> RAF(d!1).RAF(d!1) Arrhenius(phi,EDim)
  DrugBind: RAF(i) + I(r) <-> RAF(i!1).I(r!1)      Arrhenius(phi,EDrug)
end reaction rules
```

The first pattern assigns an energy to a dimer bond and the second to an inhibitor bond. The third assigns a coupling contribution to each inhibitor-bound protomer in a dimer. It has one match in a singly occupied dimer and two in a doubly occupied dimer. Thus binding an inhibitor stabilizes the dimer, and dimerization increases inhibitor affinity by the same coupling factor. The reciprocal effect follows from the shared energy assignment.

The dimer bond joins identical molecules. BioNetGen counts that symmetric energy pattern once per bond and includes a factor of one half in the generated association rate for two identical free protomers. With the present parameter convention, the macroscopic dissociation constant of the inhibitor-free dimer is therefore

$$\frac{[\text{RAF}]^2}{[\text{RAF}_2]} = 2C_{\text{ref}}e^{g_{\text{Dim}}}.$$

The parameter KD_dim defines $C_{\text{ref}}e^{g_{\text{Dim}}}$ in this file; the measured macroscopic constant includes the additional symmetry factor. Association of inhibitor to an empty dimer has a factor of two because either protomer can bind, while inhibitor dissociation from a doubly occupied dimer has the corresponding factor of two. Generated statistical factors must be included when comparing rate constants with a bulk equilibrium constant.

To explore the consequence of the coupling, suppose that an unoccupied protomer in a dimer is catalytically available, a drug-bound protomer is inhibited, and a monomer contributes no activity. The observable **Active** counts available protomers under those assumptions. It is an activity proxy; the model contains no substrate turnover or downstream ERK reactions, and its parameters are illustrative.

```

raf = load("raf_inhibitor_energy")
doses = np.logspace(-4, 1, 35)
fig0, ax0 = plt.subplots(figsize=(4.2, 3.2))
fig1, ax1 = plt.subplots(figsize=(4.2, 3.2))
axes = [ax0, ax1]
raf_peaks = {}
for coupling in (1, 100):
    baseline = last(run(raf, I0=0, omega=coupling), "Active")
    activity, dimers = [], []
    for dose in doses:
        r = run(raf, I0=dose, omega=coupling)
        activity.append(last(r, "Active")/baseline)
        dimers.append(last(r, "DimerRAF")/raf.get_param("RAFO"))
    raf_peaks[coupling] = max(activity)
    axes[0].semilogx(doses, activity, label=f"$\\omega$ = {coupling:g}")
    axes[1].semilogx(doses, dimers, label=f"$\\omega$ = {coupling:g}")
axes[0].set_ylabel("available dimer protomers / baseline")
axes[1].set_ylabel("fraction of RAF in dimers")
for ax in axes:
    ax.set_xlabel("total inhibitor ( $\mu$ M)")
    ax.legend()
plt.show()

```

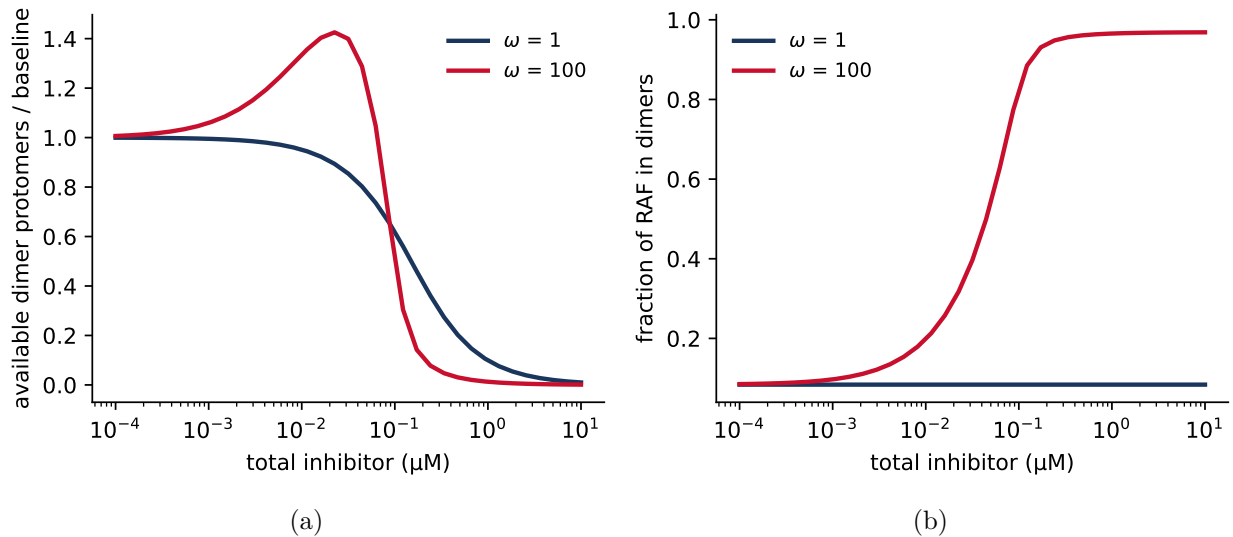


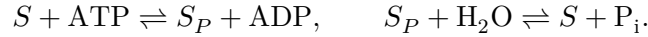
Figure 8.4: An illustrative RAF assembly model, without coupling ($\omega = 1$) and with favorable coupling ($\omega = 100$). (a) Available dimer protomers relative to their number without inhibitor. Favorable coupling increases this number at low inhibitor concentrations, and inhibitor occupancy reduces it at higher doses. (b) Fraction of RAF in dimers.

With favorable coupling, the available-protomer count reaches 1.43 times its inhibitor-free value in this scan. At low doses, inhibitor recruits RAF into dimers, creating additional unoccupied dimer partners. As the dose rises, occupation of those partners reduces the activity proxy even though the

dimer fraction remains high. Removing the coupling gives the comparison curve in Figure 8.4. The calculation demonstrates a possible assembly mechanism for a nonmonotonic response; predicting a cellular ERK response would require additional catalytic reactions and experimental constraints.

8.5 Equilibrium and chemically driven signaling

A signaling pathway can maintain currents by consuming chemical fuel. For example, a kinase and a phosphatase can carry a substrate around the cycle



The net reaction is ATP hydrolysis. Including the fuel species, the reaction free energy is

$$\Delta G_{\text{hyd}} = \mu_{\text{ADP}} + \mu_{\text{P}_i} - \mu_{\text{ATP}} - \mu_{\text{H}_2\text{O}}, \quad \mu_i = \mu_i^\circ + RT \ln a_i,$$

where a_i is a dimensionless activity. With ATP, ADP, and phosphate maintained by external processes, their chemical potentials can sustain $\Delta G_{\text{hyd}} < 0$. Such externally maintained pools are called **chemostats**. The substrate concentrations can then become stationary while phosphorylation and dephosphorylation continue at equal nonzero rates.

Local detailed balance relates the forward-to-reverse propensity ratio of an elementary transition to its free-energy change, including changes in the reservoirs. Around the complete fuel-consuming cycle, the dimensionless forward-to-reverse product is $\exp[-\Delta G_{\text{hyd}}/(RT)]$. A favorable hydrolysis reaction therefore permits a product different from unity. For a closed system at equilibrium, the chemical potentials adjust until the net affinity of each cycle vanishes.

Assigning energies only to S and S_P and applying the same reversible energy law to both enzyme channels would eliminate this driving force. A driven model must represent the fuel reaction explicitly or retain its chemical-potential contribution in effective forward and reverse rates. For an enzymatic mechanism, this constraint applies to elementary steps; a fitted irreversible rate or a coarse-grained catalytic rate can describe a regime where the reverse flux is negligible, but it does not by itself specify the complete thermodynamic cycle.

Garcia and coworkers used thermodynamic constraints in a model of mitochondrial ATP production (Garcia et al. 2023). Energy-based binding and assembly modules can likewise be combined with driven catalytic modules in a signaling model, provided that the reservoir contributions and reaction conventions are consistent. A global demand that every signaling reaction balance individually would remove the sustained activity that the model is intended to describe.

8.6 Choosing parameters for calibration

For the two-site receptor, the equilibrium parameters are g_A , g_B , and g_{AB} , or equivalently $K_{D,A}$, $K_{D,B}$, and ω . Given those parameters, the four context-dependent affinity constants are constrained. There are three independent equilibrium parameters, since the cycle constraint removes one degree of freedom from the four affinity constants. The kinetic scales for the two binding rules and their allocation parameters determine how the system approaches equilibrium.

An equilibrium binding curve can constrain affinity and coupling, provided that the measurements distinguish their effects. It contains no direct information about the kinetic scale. Conversely, association and washout time courses can constrain rates, particularly when experiments probe several ligand concentrations and occupancy contexts. At a single fixed energy difference, different combinations of E_a and ϕ can give the same forward and reverse rates, so estimating both requires additional context or an imposed kinetic assumption.

In the RAF example, an activity-proxy curve combines dimer formation with inhibitor occupancy. Measurements of the dimer fraction and bound inhibitor would constrain these processes more directly. Since activity also depends on our assumption about which protomers can signal, fitting a curve cannot establish that assumption without independent evidence.

Energy parameters can be shared across many reactions generated by a rule. Such sharing reduces the number of independent affinity parameters and enforces cycle consistency, but the chosen patterns still express a mechanistic hypothesis. If coupling depends on a third component or a conformation omitted from the pattern, a fit may compensate by shifting the available energy parameters. The parameter-estimation and uncertainty chapters develop ways to distinguish a well-constrained energy from a parameter that absorbs missing model structure.

8.7 Summary

- In a passive binding model, the product of association constants around any cycle of reactions must be the same in both directions. A model that violates this condition can still conserve mass and reach a steady state, but its steady state carries a circulating flux that no energy source sustains.
- An energy pattern assigns a free-energy contribution to each occurrence of a molecular structure, and BioNetGen sums the contributions to obtain the energy of each species. Cooperative effects defined this way are reciprocal, and every cycle is consistent.
- The `Arrhenius` rate law sets the ratio of forward and reverse rates from the energy change. `Ea` sets their common scale and `phi` divides the energy change between them, and neither affects the equilibrium.
- The binding polynomial gives equilibrium occupancies as functions of the free ligand concentration. In a closed system the free concentration follows from mass balance.
- Statistical factors for identical molecules and sites enter the generated rates and must be included when rate constants are compared with measured equilibrium constants.
- A signaling pathway driven by ATP hydrolysis maintains currents that a closed system at equilibrium cannot. A driven model must represent the fuel or its chemical potential rather than impose detailed balance on every reaction.
- Equilibrium measurements constrain energies, while time courses are needed to constrain the kinetic parameters.

8.8 Exercises

1. Derive the cycle constraint for the two-site receptor directly from the four equilibrium mass-action equations. In `two_site_inconsistent.bngl`, change a second rate so that the cycle ratio becomes unity. Confirm conservation and individual reaction balance after simulation.

2. Set `KD_A` and `KD_B` to different values in `two_site_energy.bngl`. Derive all four equilibrium weights and compare the binding-polynomial prediction with the ODE result. Explain why cooperative enhancement must remain reciprocal even for unequal sites.
3. Compare independent, favorable, and unfavorable coupling. Plot double occupancy against both total and free ligand. Identify the regime in which ligand depletion changes the interpretation of the input dose.
4. At fixed affinities, change the kinetic scales and `phi`. Design association and dissociation measurements that could discriminate among the resulting time courses. State which parameters would remain ambiguous if only equilibrium occupancy were measured.
5. Inspect the network generated from `raf_inhibitor_energy.bngl`. Identify the symmetry factors in dimer association and inhibitor binding. Derive the inhibitor-free dimer equilibrium and verify it numerically at several total RAF concentrations.
6. Replace the RAF activity proxy by the concentration of entirely inhibitor-free dimers. Compare its dose response with the proxy in Figure 8.4 and explain which mechanistic assumption changed.
7. Write effective forward and reverse rate ratios for the kinase and phosphatase cycle that include the nucleotide and phosphate activities. Show that their product is determined by the ATP-hydrolysis free energy. Explain how a stationary substrate distribution can coexist with continuing fuel consumption.

8.9 Further reading

Sekar, Hogg, and Faeder describe energy patterns and their implementation in BioNetGen (Sekar et al. 2016). The accompanying models in this chapter provide small tests of cycle consistency, rate allocation, and symmetry. Poulidakos and coworkers provide the experimental RAF example (Poulidakos et al. 2010), while Garcia and coworkers illustrate thermodynamic constraints in ATP production (Garcia et al. 2023).

References

- Begitt, Andreas, Thomas Meyer, Mathias van Rossum, and Uwe Vinkemeier. 2000. “Nucleocytoplasmic Translocation of Stat1 Is Regulated by a Leucine-Rich Export Signal in the Coiled-Coil Domain.” *Proceedings of the National Academy of Sciences of the USA* 97 (19): 10418–23. <https://doi.org/10.1073/pnas.190318397>.
- Blinov, Michael L., James R. Faeder, Byron Goldstein, and William S. Hlavacek. 2004. “BioNetGen: Software for Rule-Based Modeling of Signal Transduction Based on the Interactions of Molecular Domains.” *Bioinformatics* 20 (17): 3289–91. <https://doi.org/10.1093/bioinformatics/bth378>.
- Borghans, J. A. M., R. J. de Boer, and L. A. Segel. 1996. “Extending the Quasi-Steady State Approximation by Changing Variables.” *Bulletin of Mathematical Biology* 58 (1): 43–63. <https://doi.org/10.1007/BF02458281>.
- Bouhaddou, Mehdi, Anne Marie Barrette, Alan D. Stern, et al. 2018. “A Mechanistic Pan-Cancer Pathway Model Informed by Multi-Omics Data Interprets Stochastic Cell Fate Responses to Drugs and Mitogens.” *PLoS Computational Biology* 14 (3): e1005985. <https://doi.org/10.1371/journal.pcbi.1005985>.
- Chylek, Lily A., Vyacheslav Akimov, Jörn Dengjel, et al. 2014. “Phosphorylation Site Dynamics of Early T-Cell Receptor Signaling.” *PLoS ONE* 9 (8): e104240. <https://doi.org/10.1371/journal.pone.0104240>.
- Colvin, Joshua, Michael I. Monine, Ryan N. Gutenkunst, William S. Hlavacek, Daniel D. Von Hoff, and Richard G. Posner. 2010. “RuleMonkey: Software for Stochastic Simulation of Rule-Based Models.” *BMC Bioinformatics* 11: 404. <https://doi.org/10.1186/1471-2105-11-404>.
- Creamer, Matthew S., Edward C. Stites, Meraj Aziz, et al. 2012. “Specification, Annotation, Visualization and Simulation of a Large Rule-Based Model for ERBB Receptor Signaling.” *BMC Systems Biology* 6: 107. <https://doi.org/10.1186/1752-0509-6-107>.
- Danos, Vincent, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. 2007. “Rule-Based Modelling of Cellular Signalling.” *CONCUR 2007 – Concurrency Theory*, Lecture notes in computer science, 17–41. https://doi.org/10.1007/978-3-540-74407-8_3.
- Danos, Vincent, Jérôme Feret, Walter Fontana, and Jean Krivine. 2007. “Scalable Simulation of Cellular Signaling Networks.” *Programming Languages and Systems (APLAS 2007)*, Lecture notes in computer science, 139–57. https://doi.org/10.1007/978-3-540-76637-7_10.
- Elowitz, Michael B., Arnold J. Levine, Eric D. Siggia, and Peter S. Swain. 2002. “Stochastic Gene Expression in a Single Cell.” *Science* 297 (5584): 1183–86. <https://doi.org/10.1126/science>.

1070919.

- Faeder, James R., Michael L. Blinov, and William S. Hlavacek. 2009. "Rule-Based Modeling of Biochemical Systems with BioNetGen." *Methods in Molecular Biology* 500: 113–67. https://doi.org/10.1007/978-1-59745-525-1_5.
- Feret, Jérôme, Vincent Danos, Jean Krivine, Russ Harmer, and Walter Fontana. 2009. "Internal Coarse-Graining of Molecular Systems." *Proceedings of the National Academy of Sciences of the United States of America* 106 (16): 6453–58. <https://doi.org/10.1073/pnas.0809908106>.
- Ferrell, James E., and Eric M. Machleder. 1998. "The Biochemical Basis of an All-or-None Cell Fate Switch in *Xenopus* Oocytes." *Science* 280 (5365): 895–98. <https://doi.org/10.1126/science.280.5365.895>.
- Garcia, Guadalupe C., Kavya Gupta, Thomas M. Bartol, Terrence J. Sejnowski, and Padmini Rangamani. 2023. "Mitochondrial Morphology Governs ATP Production Rate." *Journal of General Physiology* 155 (9): e202213263. <https://doi.org/10.1085/jgp.202213263>.
- Gillespie, Daniel T. 2007. "Stochastic Simulation of Chemical Kinetics." *Annual Review of Physical Chemistry* 58: 35–55. <https://doi.org/10.1146/annurev.physchem.58.032806.104637>.
- Goldstein, Byron, and Alan S. Perelson. 1984. "Equilibrium Theory for the Clustering of Bivalent Cell Surface Receptors by Trivalent Ligands. Application to Histamine Release from Basophils." *Biophysical Journal* 45 (6): 1109–23. [https://doi.org/10.1016/S0006-3495\(84\)84259-9](https://doi.org/10.1016/S0006-3495(84)84259-9).
- Harris, Leonard A., Justin S. Hogg, and James R. Faeder. 2009. "Compartmental Rule-Based Modeling of Biochemical Systems." *Proceedings of the 2009 Winter Simulation Conference*, 908–19. <https://doi.org/10.1109/WSC.2009.5429719>.
- Harris, Leonard A., Justin S. Hogg, José-Juan Tapia, et al. 2016. "BioNetGen 2.2: Advances in Rule-Based Modeling." *Bioinformatics* 32 (21): 3366–68. <https://doi.org/10.1093/bioinformatics/btw469>.
- Heinrich, Reinhart, and Tom A. Rapoport. 1974. "A Linear Steady-State Treatment of Enzymatic Chains. General Properties, Control and Effector Strength." *European Journal of Biochemistry* 42 (1): 89–95. <https://doi.org/10.1111/j.1432-1033.1974.tb03318.x>.
- Hlavacek, William S., James R. Faeder, Michael L. Blinov, Alan S. Perelson, and Byron Goldstein. 2003. "The Complexity of Complexes in Signal Transduction." *Biotechnology and Bioengineering* 84 (7): 783–94. <https://doi.org/10.1002/bit.10842>.
- Hlavacek, William S., James R. Faeder, Michael L. Blinov, Richard G. Posner, Michael Hucka, and Walter Fontana. 2006. "Rules for Modeling Signal-Transduction Systems." *Science's STKE* 2006 (344): re6. <https://doi.org/10.1126/stke.3442006re6>.
- Hoeve, Johanna ten, Maria de Jesus Ibarra-Sanchez, Yubin Fu, et al. 2002. "Identification of a Nuclear Stat1 Protein Tyrosine Phosphatase." *Molecular and Cellular Biology* 22 (16): 5662–68. <https://doi.org/10.1128/MCB.22.16.5662-5668.2002>.

- Husar, Adam, Mariam Ordyan, Guadalupe C. Garcia, et al. 2024. “MCell4 with BioNetGen: A Monte Carlo Simulator of Rule-Based Reaction-Diffusion Systems with Python Interface.” *PLOS Computational Biology* 20 (4): e1011800. <https://doi.org/10.1371/journal.pcbi.1011800>.
- Lauffenburger, Douglas A., and Jennifer J. Linderman. 1993. *Receptors: Models for Binding, Trafficking, and Signaling*. Oxford University Press.
- Lin, Yen Ting, Song Feng, and William S. Hlavacek. 2019. “Scaling Methods for Accelerating Kinetic Monte Carlo Simulations of Chemical Reaction Networks.” *The Journal of Chemical Physics* 150 (24): 244101. <https://doi.org/10.1063/1.5096774>.
- Mitra, Eshan D., Ryan Suderman, Joshua Colvin, et al. 2019. “PyBioNetFit and the Biological Property Specification Language.” *iScience* 19: 1012–36. <https://doi.org/10.1016/j.isci.2019.08.045>.
- Monine, Michael I., Richard G. Posner, Paul B. Savage, James R. Faeder, and William S. Hlavacek. 2010. “Modeling Multivalent Ligand-Receptor Interactions with Steric Constraints on Configurations of Cell-Surface Receptor Aggregates.” *Biophysical Journal* 98 (1): 48–56. <https://doi.org/10.1016/j.bpj.2009.09.043>.
- Peccoud, Jean, and Bernard Ycart. 1995. “Markovian Modeling of Gene-Product Synthesis.” *Theoretical Population Biology* 48 (2): 222–34. <https://doi.org/10.1006/tpbi.1995.1027>.
- Poulikakos, Poulikos I., Chao Zhang, Gideon Bollag, Kevan M. Shokat, and Neal Rosen. 2010. “RAF Inhibitors Transactivate RAF Dimers and ERK Signalling in Cells with Wild-Type BRAF.” *Nature* 464 (7287): 427–30. <https://doi.org/10.1038/nature08902>.
- Raj, Arjun, and Alexander van Oudenaarden. 2008. “Nature, Nurture, or Chance: Stochastic Gene Expression and Its Consequences.” *Cell* 135 (2): 216–26. <https://doi.org/10.1016/j.cell.2008.09.050>.
- Reich, Nancy C., and Ling Liu. 2006. “Tracking STAT Nuclear Traffic.” *Nature Reviews Immunology* 6 (8): 602–12. <https://doi.org/10.1038/nri1885>.
- Sekar, John A. P., and James R. Faeder. 2012. “Rule-Based Modeling of Signal Transduction: A Primer.” *Methods in Molecular Biology* 880: 139–218. https://doi.org/10.1007/978-1-61779-833-7_9.
- Sekar, John A. P., Justin S. Hogg, and James R. Faeder. 2016. “Energy-Based Modeling in BioNetGen.” *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 1460–67. <https://doi.org/10.1109/BIBM.2016.7822739>.
- Sekar, John Arul Prakash, Jose-Juan Tapia, and James R. Faeder. 2017. “Automated Visualization of Rule-Based Models.” *PLOS Computational Biology* 13 (11): e1005857. <https://doi.org/10.1371/journal.pcbi.1005857>.
- Sneddon, Michael W., James R. Faeder, and Thierry Emonet. 2011. “Efficient Modeling, Simulation and Coarse-Graining of Biological Complexity with NFsim.” *Nature Methods* 8 (2): 177–83.

<https://doi.org/10.1038/nmeth.1546>.

- Thomas, Brandon R., Lily A. Chylek, Joshua Colvin, et al. 2016. “BioNetFit: A Fitting Tool Compatible with BioNetGen, NFsim and Distributed Computing Environments.” *Bioinformatics* 32 (5): 798–800. <https://doi.org/10.1093/bioinformatics/btv655>.
- Tyson, John J., Katherine C. Chen, and Béla Novák. 2003. “Sniffers, Buzzers, Toggles and Blinkers: Dynamics of Regulatory and Signaling Pathways in the Cell.” *Current Opinion in Cell Biology* 15 (2): 221–31. [https://doi.org/10.1016/S0955-0674\(03\)00017-6](https://doi.org/10.1016/S0955-0674(03)00017-6).
- Tyson, John J., and Béla Novák. 2010. “Functional Motifs in Biochemical Reaction Networks.” *Annual Review of Physical Chemistry* 61: 219–40. <https://doi.org/10.1146/annurev.physchem.012809.103457>.
- Tzafiriri, A. R. 2003. “Michaelis–Menten Kinetics at High Enzyme Concentrations.” *Bulletin of Mathematical Biology* 65 (6): 1111–29. [https://doi.org/10.1016/S0092-8240\(03\)00059-4](https://doi.org/10.1016/S0092-8240(03)00059-4).
- Xiong, Wen, and James E. Ferrell. 2003. “A Positive-Feedback-Based Bistable ‘Memory Module’ That Governs a Cell Fate Decision.” *Nature* 426 (6965): 460–65. <https://doi.org/10.1038/nature02089>.
- Xu, Wen, Adam M. Smith, James R. Faeder, and G. Elisabeta Marai. 2011. “RuleBender: A Visual Interface for Rule-Based Modeling.” *Bioinformatics* 27 (12): 1721–22. <https://doi.org/10.1093/bioinformatics/btr197>.
- Yang, Jin, Michael I. Monine, James R. Faeder, and William S. Hlavacek. 2008. “Kinetic Monte Carlo Method for Rule-Based Modeling of Biochemical Networks.” *Physical Review E* 78 (3): 031910. <https://doi.org/10.1103/PhysRevE.78.031910>.

A Installing the tools

A.1 No installation: BNG Playground

[BNG Playground](#) runs BioNetGen in a web browser. Paste any model from this book into its editor and click **Run**. It is the quickest way to try a model, and it works on any computer.

A.2 Visual Studio Code

1. Install [Visual Studio Code](#).
2. In the Extensions panel, search for **BioNetGen** and install it.
3. Open a `.bngl` file and click the run button in the editor toolbar. Results are written to a time-stamped folder and the main output (`.gdat`) opens in a plot viewer.

The extension needs Python and Perl. It installs PyBioNetGen, which contains BioNetGen itself, when first used. macOS and Linux include Perl. On Windows, install [Strawberry Perl](#).

To keep results out of your working folders, set `bngl.general.result_folder` in VS Code's settings to an absolute path, for example a **BNG-results** folder in your project.

A.3 Python

The book's Python code uses two BioNetGen packages and the standard scientific packages:

- [bngsim](#): fast ODE, stochastic, and network-free simulation of BioNetGen models, run in-process from Python.
- [PyBioNetGen](#): supplies BioNetGen, which bngsim uses to turn a `.bngl` file into a reaction network.
- `numpy`, `pandas`, and `matplotlib` for analysis and plotting, and `jupyter` for notebooks.

Install them with

```
pip install bngsim bionetgen numpy pandas matplotlib jupyter
```

Every model and figure in the book runs through this route. BNG Playground and the VS Code extension run most models, but they may not support every feature used in later chapters, such as network-free simulation through bngsim. If a model fails in one of them, run it from Python.

A.4 The book's models and code

The models are in the book's repository on GitHub. Download it, either as a ZIP file from the repository page or with git:

```
git clone https://github.com/RuleWorld/bionetgen-book.git
cd bionetgen-book
```

Paths such as `models/ch01/stat_step1_binding.bngl` in the book are relative to this folder. Check the installation by running Python from the folder and simulating the first model of Chapter 1:

```
import bngsim
model = bngsim.Model.from_bngl("models/ch01/stat_step1_binding.bngl")
sim = bngsim.Simulator(model, method="ode")
model.reset()
result = sim.run(t_span=(0, 20), n_points=101)
print(result.observables["LR"][-1])    # about 0.989
```

The chapters also use a few helper functions, for example to display model files and generated networks, from `scripts/bookutils.py`. Each chapter imports them in a hidden cell at the start. To run a chapter's code yourself, start Python in the repository folder and run

```
import sys
sys.path.insert(0, "scripts")
from bookutils import *
```

A.5 Reproducing the book

To reproduce every figure and number, create the environment used to build the book, which also installs Quarto and Graphviz, the program that lays out the contact maps and rule diagrams:

```
conda env create -f environment.yml
conda activate bionetgen-book
quarto render
```

Stochastic simulations in the book set their random seeds, so they give the same results with the same software versions. This version of the book was built with the following versions:

package	version
Python	3.12.15
bngsim	0.16.0
bionetgen	0.8.7
numpy	2.5.4

package	version
pandas	3.0.6
matplotlib	3.11.2

B BNGL language reference

BNGL describes molecular structures, patterns that select those structures, and rules that change them. A file also contains actions that generate a reaction network, run simulations, or export the model. The tables below give the syntax used throughout this book; the accompanying files in `models/appendix_b/` are executable examples.

The reference was checked against the [BioNetGen source snapshot 9601746](#) and tested with BioNetGen 2.9.3. The parser accepts some constructs that a particular simulator or exporter may handle differently. Those distinctions are identified where they affect the meaning of a model, especially in [Section B.7](#) and [Section B.11](#).

B.1 File structure and lexical conventions

A model definition is enclosed by `begin model` and `end model`. Within it, each block begins with `begin block-name` and ends with the corresponding `end block-name`. Actions follow the model, either as individual commands or inside `begin actions` and `end actions`. An explicit model wrapper is recommended even though the parser also accepts older files without it.

Block	Contents	Typical dependency
<code>parameters</code>	Numeric values and algebraic expressions	Define quantities before using them
<code>compartments</code>	Compartment names, dimensions, sizes, and containment	Declare enclosing compartments first
<code>molecule types</code>	Molecule names, component names, and allowed states	Define types before species and patterns
<code>seed species</code>	Fully specified initial molecular species and amounts	Requires parameters, types, and any compartments
<code>observables</code>	Patterns used to count molecules or complexes	Define observables before functions that use them
<code>functions</code>	Expressions that can depend on observables or time	Requires the referenced quantities
<code>energy patterns</code>	Structural patterns with free-energy contributions	Requires molecular types and energy parameters
<code>reaction rules</code>	Reactant patterns, product patterns, and rate laws	Requires the referenced definitions

Some blocks are optional. For example, a model with mass-action rules needs no `functions` block, and a spatially homogeneous model needs no `compartments` block. Use the order above to make

dependencies visible. The `species`, `reactions`, and `groups` blocks also occur in generated `.net` files; they describe an enumerated network rather than the compact rule definition.

Construct	Syntax and interpretation
Comment	# begins a comment extending to the end of the line
Continuation	A trailing backslash joins a logical statement to the next line
Identifier	Use a letter or underscore followed by letters, digits, or underscores; names are case-sensitive
Number	Integer, decimal, or scientific notation, such as 1, 0.1, or 1e-3
Rule label	Bind: precedes a rule and names it in generated output
String	Quoted text in action arguments, such as "ode" or "R(1)"
Action options	{name=>value,...}; string values are quoted and arrays use square brackets

Choose distinct names for parameters, observables, and functions since they share the expression namespace. Parentheses enclose molecule components, function arguments, and action arguments, with different meanings in each context. Semicolons are unnecessary in the examples in this book.

The following file has the core blocks and an ODE action:

```

begin model
begin parameters
  kon      0.1
  koff     0.2
  kdimer   0.01
  kundim   0.1
  kp       0.3
  km       0.1
  K        2
  R0       3
  L0       5
end parameters
begin molecule types
  R(l,d,Y~u~p)
  L(r)
end molecule types
begin seed species
  R(l,d,Y~u)           R0
  L(r)                 L0
  R(l,d!1,Y~p).R(l,d!1,Y~u)  1
  R(l!1,d,Y~p).L(r!1)      1
end seed species

```

```

begin observables
  Molecules Rtotal      R()
  Species    Rcomplexes R()
  Molecules Rphos       R(Y~p)
  Molecules Rbound      R(1!+)
  Molecules Roverlap    R(1!+) R(Y~p)
  Species    Rpair       R()==2
  Molecules Lttotal     L()
end observables
begin functions
  phos_rate() = kp/(1+Rphos/K)
end functions
begin reaction rules
  Bind:    R(l) + L(r) <-> R(l!1).L(r!1) kon,koff
  Dimer:   R(d) + R(d) <-> R(d!1).R(d!1) kdimer,kundim
  Phos:    R(Y~u) -> R(Y~p) phos_rate()
  Dephos:  R(Y~p) -> R(Y~u) km
end reaction rules
end model
generate_network({overwrite=>1})
simulate({method=>"ode",t_end=>10,n_steps=>20})

```

B.2 Parameters, expressions, and units

Each parameter line has the form **name expression**. The expression can refer to previously defined parameters. A derived parameter such as `KD koff/kon` records an algebraic dependence; observables that change during a simulation belong in functions when they control a rate.

Operation	BNGL syntax
Arithmetic	<code>+, -, *, /, ^</code>
Grouping	<code>(expression)</code>
Comparison	<code><, <=, >, >=, ==, !=</code>
Logical combination	<code>&&, </code>
Conditional	<code>if(condition,expression_if_true,expression_if_false)</code>
Exponential and logarithms	<code>exp(x), ln(x), log10(x), log2(x)</code>
Magnitude and root	<code>abs(x), sqrt(x)</code>
Trigonometric functions	<code>sin, cos, tan, asin, acos, atan</code>
Hyperbolic functions	<code>sinh, cosh, tanh, asinh, acosh, atanh</code>
Aggregation	<code>min(...), max(...), sum(...), avg(...)</code>
Nearest integer	<code>rint(x)</code> , implemented as $\lfloor x + 1/2 \rfloor$
Mathematical constants	<code>_pi(), _e()</code>
Simulation time	<code>time()</code> in a dynamic expression

Use `^` for exponentiation and explicit parentheses to clarify compound conditions. Internal states such as `Y~p` are labels; they are distinct from the numeric quantities used in expressions. Avoid

undefined operations, including division by zero and logarithms of nonpositive values, in all reachable model states.

BNGL does not infer a unit system from parameter names. For an amount variable x , a unimolecular rate coefficient has units of inverse time, a bimolecular coefficient has units of inverse amount per time, and a source rate has units of amount per time. Select concentration or molecule count units and convert every parameter consistently.

`substanceUnits("Concentration")` and `substanceUnits("Number")` declare an amount convention for processing and export. They do not perform an automatic conversion of arbitrary seed amounts and kinetic parameters. Stochastic simulation requires molecule counts and rates appropriate to those counts. Compartment sizes and the amount convention together affect compartmental scaling (Section B.8).

B.3 Molecule types and molecular species

A molecule type declares the components that can occur on a molecule and the internal states allowed for each component:

```
begin molecule types
  R(l,d,Y~u~p)
  L(r)
end molecule types
```

`R(l,d,Y~u~p)` declares a receptor with components `l`, `d`, and `Y`. Component `Y` can have state `u` or `p`; a particular receptor has one of those states. A component can carry both a state and a bond, as in `R(Y~p!1).K(r!1)` if the declared molecule types allow that structure.

Repeated component names represent equivalent sites. For example, `A(b,b,b)` declares interchangeable binding sites on `A`; the declaration can compress symmetric interactions but also introduces matching multiplicities. Distinct names such as `b1` and `b2` distinguish sites and permit site-specific rates.

Structure	Meaning
<code>L(r)</code>	A ligand whose <code>r</code> component is unbound
<code>R(l,d,Y~u)</code>	A complete receptor with both binding components free
<code>R(l!1,d,Y~p).L(r!1)</code>	A receptor-ligand complex, with the bond endpoints sharing label <code>1</code>
<code>R(l,d!1,Y~p).R(l,d!1,Y~u)</code>	A dimer containing receptors in different internal states
<code>A(x!1,y!2).B(a!1).C(a!2)</code>	A connected complex with distinct labeled bonds

A dot joins molecules within one species or pattern. Bond labels identify the endpoints of an edge within that expression; reusing `!1` in another expression does not connect the expressions. Each

explicit bond has two endpoints. Molecule order and bond numbering can change when BioNetGen writes a canonical species string, while the molecular graph remains the same.

A seed-species line has the form **complete_species amount**. Specify every component and internal state needed to identify the species. A seed species must be connected, so separate unbound molecules occupy separate seed lines. A zero initial amount is useful when later actions must address that species explicitly. Prefixing a seed species with \$, as in **\$A() A0**, fixes the amount of that particular species. Complexes containing the same molecule type are separate species and acquire their own amounts.

B.4 Patterns and matching

A **pattern** selects molecular structures by specifying some of their properties. Components omitted from a pattern impose no condition on their states or bonds. A component that appears without a bond annotation is required to be unbound. These two forms therefore select different sets of molecules:

Pattern	Required properties
R()	A receptor in any internal state or binding context
R(1)	The 1 component is present and unbound
R(1!+)	The 1 component is bound, with the partner unspecified
R(1!?)	The 1 component may be bound or unbound
R(Y~p)	State p at Y, with Y unbound
R(Y~p!?)	State p at Y, with either bond status
R(1,Y~p)	Both named components are unbound and Y has state p
R(1!1).L(r!1)	A receptor bound to a ligand through the specified components

If Y is purely an internal-state component in the model, its unbound condition adds no restriction. When a component can also bind a partner, include **!?** to select its state independently of binding. Omitting **~state** leaves the internal state unspecified; **!?** refers to bonds.

Consider **R(Y~u) -> R(Y~p) kp**. The rule preserves omitted components and their partners, so the phosphorylation event can occur in a ligand-bound receptor or a dimer. Adding **1** to both sides restricts the rule to receptors with a free ligand-binding component. Adding **1!+** restricts it to bound receptors. Such unchanged requirements are the rule's context.

A pattern can have several embeddings in a species, for example when a dimer contains several receptors in the required state. Those embeddings can affect observable weights and reaction rates. The pattern attribute **{MatchOnce}** restricts its matching to at most one embedding in a given species. For example, **{MatchOnce}R()** requests one match per receptor-containing complex. Apply it only when the intended event or measurement is defined per complex; restricting a match that identifies distinct reactive sites can remove physical events.

B.5 Observables

An observable line has the form `type name pattern`, followed by any additional patterns. Patterns may be separated by spaces or commas.

Type or modifier	Interpretation
Molecules	Sum the pattern embeddings, weighted by the amount of each species
Species	Count each matching species instance once per listed pattern, irrespective of that pattern's embedding multiplicity
Species Pair <code>R()==2</code>	Select species with exactly two embeddings of <code>R()</code>
Species Cluster <code>R()>2</code>	Select species with more than two embeddings of <code>R()</code>

Species counts the amount of matching complexes. It does not report the number of distinct species definitions in the generated network. For a single-molecule pattern, **Molecules** counts molecules of that type. For a multimolecule motif, it counts motif embeddings, which can include a symmetry multiplicity. Quantifiers such as `==2` refer to the number of embeddings of the preceding pattern; for `R()` this is the number of receptors in a complex.

```
begin observables
  Molecules Rtotal      R()
  Species   Rcomplexes  R()
  Molecules Rphos       R(Y~p)
  Molecules Rbound      R(1!+)
  Molecules Roverlap    R(1!+) R(Y~p)
  Species   Rpair       R()==2
  Molecules Ltotal      L()
end observables
```

The seed mixture gives `Rtotal` = 6 and `Rcomplexes` = 5 in the chosen amount units. Each dimer contributes its receptors to `Rtotal` and one complex to `Rcomplexes`.

Contributions from multiple patterns are added. A molecule matching both patterns can be counted twice, and a complex matching several **Species** patterns can contribute once for each pattern. In the example, `Roverlap` = 3, since a ligand-bound phosphorylated receptor contributes to both patterns. Use disjoint patterns or an expression with an overlap correction when the desired measurement is a union of sets.

B.6 Reaction rules and graph changes

A reaction rule gives reactant patterns, an arrow, product patterns, and a rate law. An optional label precedes the reactants.

Rule form	Rate-law entries
label: reactants -> products k	One forward rate law
label: reactants <-> products kf,kr	Forward and reverse rate laws
0 -> A() ks	Source with no molecular reactant
A() -> 0 kd	Deletion
A() -> A() + B() ks	Template-dependent production of B
A(x) + B(y) <-> A(x!1).B(y!1) kon,koff	Intermolecular association and dissociation
A(Y~u) -> A(Y~p) kp	Internal-state change

The plus sign separates reactant or product species patterns. Each occurrence represents a species instance. Thus `A() + A()` requests two reactant instances, while a dotted pattern can select molecules within one complex. Intracomplex bond formation can be represented with a dotted reactant pattern selecting components already in the same complex.

On the product side, BioNetGen changes the molecular properties named by the rule and retains the surrounding molecular context. When a new molecule is created, the product must specify the components and internal states needed to identify it. With repeated molecule types or an ambiguous transformation, use consistent molecule tags on both sides and inspect the generated reactions to check the correspondence.

Rules that overlap contribute additional reaction channels. A more specific rule does not automatically replace a general rule matching the same molecules. Partition contexts if different rates are intended for mutually exclusive cases.

B.6.1 Deletion and rule modifiers

```
begin reaction rules
  LoseComplex: R() -> 0 k
  LoseReceptor: R() -> 0 k DeleteMolecules
end reaction rules
```

For a bound R-L seed complex, `LoseComplex` removes the entire complex. `LoseReceptor` removes the matched receptor and its incident bond, leaving the ligand as a free molecule. `DeleteMolecules` is therefore needed when unmentioned partners should survive deletion. A rule that explicitly retains a partner on the product side can also specify that survival.

Modifier	Meaning and scope
<code>DeleteMolecules</code>	Delete specified molecules while retaining unmentioned partners, which can become separate product species
<code>MoveConnected</code>	A transported molecule carries connected, unmentioned molecules in its original compartment

Modifier	Meaning and scope
TotalRate	NFsim interprets a global functional rate as a total rule rate; see Section B.7 for the tested network behavior
<code>include_reactants(i,pattern)</code>	Require a matching pattern in the whole species selected by reactant position <i>i</i>
<code>exclude_reactants(i,pattern)</code>	Reject a matching pattern in that reactant species
<code>include_products(i,pattern)</code>	Apply an inclusion filter to the generated product species at position <i>i</i>
<code>exclude_products(i,pattern)</code>	Apply an exclusion filter to that product species

Modifiers follow the rate law. Positions in the filter arguments begin at one and refer to the plus-separated patterns. Keep filter argument lists free of spaces, as in `exclude_reactants(1,R(1!+))`. A filter examines the whole selected species, so in a multireceptor complex it may find a different receptor from the one transformed by the rule. Explicit local context is preferable when the requirement concerns the reacting receptor.

The parser also accepts `priority=value`. Rule priority is used in specialized workflows and should not be interpreted as a general override of overlapping kinetic rules across simulators.

B.7 Functions and rate-law conventions

A global function has the form `name() = expression`. It can use parameters, observables, other functions, and simulation time. Calling that function as a rule's rate law ordinarily supplies a rate coefficient: the generated reaction still multiplies the function by its reactant amounts and statistical factors.

```
begin functions
  rate() = k*Atotal/(K+Atotal)
end functions
begin reaction rules
  Coefficient: A() -> A() + B() rate()
  TotalFlux:   A() -> A() + C() rate() TotalRate
  SourceFlux:  0 -> D() rate()
end reaction rules
```

For **Coefficient**, the reaction flux is `rate()*Atotal`. Since the seed amount of **A** is fixed, the example has a constant coefficient and a constant source of **B**. **SourceFlux** has no reactant multiplier, so its flux is `rate()`.

i **TotalRate** depends on the simulation backend

In NFsim, **TotalFlux** uses `rate()` as a total rule rate. In the tested network export, **TotalFlux** and **Coefficient** receive the same functional coefficient in the `.net` file, and the ODE

simulation multiplies both by `Atotal`. The native ODE run therefore gives `Btotal = 4.5`, `Ctotal = 4.5`, and `Dtotal = 1.5` at its final time.

For a network-based source whose flux is a function of an observable, use `0 -> product function()`, as in `SourceFlux`. When a rule also consumes reactants, derive a rate coefficient that gives the intended flux and remains defined over the reachable states. The parser rejects `TotalRate` combined with a local function or Arrhenius law.

B.7.1 Local functions and tags

A local function takes a reference to a matched complex or molecule. Within that function, `Observable(x)` evaluates the observable with the scope or root specified by `x`.

Tag	Reference
<code>%x:R()</code>	The whole complex matched by this species pattern
<code>%x@membrane:R()</code>	The whole complex, with a species-level compartment constraint
<code>R%x()</code>	The particular matched receptor molecule

```
begin functions
  local_rate(x) = k*(1+Bound(x))
end functions
begin reaction rules
  Phos: %x:R(Y~u) -> %x:R(Y~p) local_rate(x)
end reaction rules
```

```
Phos: R%x(Y~u) -> R%x(Y~p) local_rate(x)
```

In both files, `Bound` is the motif `R(1!1).L(r!1)` and the seed complex contains a dimer with ligand on one receptor. The species-level tag lets each reacting receptor see the bound motif anywhere in that complex. The molecule-level tag roots the observable match at the tagged receptor, so the unbound receptor receives the base rate and the bound receptor receives the enhanced rate. A molecule reference therefore imposes more than a restriction to the containing complex; the observable's root must match that molecule.

`FunctionProduct("f(x)", "g(y)")` provides a specialized product of local functions on separate reactants for network-free models. Keep each factor dependent on its own reactant reference and check support in the selected simulator. For ordinary network generation, many local expressions expand to reaction-specific coefficients once each reactant species is known.

B.7.2 Specialized network rate laws

The parser retains built-in rate forms used by older models:

```

begin reaction rules
  Saturate: A() -> P() Sat(v,K)
  HillLaw: B() -> Q() Hill(v,K,n)
  Enzyme: S() + E() -> T() + E() MM(kcat,K)
end reaction rules

```

Form	Convention for the illustrated unit-stoichiometry reaction
Sat(v,K)	$v[A]/(K + [A])$ for the first reactant; additional reactants multiply the rate
Hill(v,K,n)	$v[B]^n/(K^n + [B]^n)$ for the first reactant; additional reactants multiply the rate
MM(kcat,K)	Substrate first, enzyme second; uses a free-substrate estimate from total substrate and enzyme

MM has a specific total-substrate convention, so substituting a familiar Michaelis–Menten formula without checking the represented pools can change the kinetics. A **Sat** law can have further saturation constants associated with subsequent reactants. These forms refer to explicit generated reactant species, which can differ from a saturation function of a pooled observable. They are also less portable across simulators and exporters than explicit functions. State the intended flux mathematically before translating a specialized form to a functional coefficient.

B.8 Compartments and transport

Each compartment line has the form **name dimension size [outside]**. The dimension is 3 for a volume or 2 for a surface compartment; **outside** names its enclosing compartment. BioNetGen uses the containment tree to check bonds and transport between neighboring locations.

```

begin compartments
  outside 3 Voutside
  membrane 2 Vmem outside
  inside 3 Vinside membrane
end compartments

```

```

begin seed species
  @outside:A() 1
  @inside:A() 0
end seed species
begin observables
  Molecules Aout A()@outside
  Molecules Ain A()@inside
end observables

```

```

begin reaction rules
  Import: @outside:A() -> @inside:A() kin
  Export: @inside:A() -> @outside:A() kout
end reaction rules

```

`@inside:A()` sets a species-level compartment, while `A()@inside` specifies the molecule's compartment. A pattern can also specify molecule locations within a complex that spans adjacent compartments. The containment tree restricts which cross-compartment bonds are allowed.

Compartmental rates can acquire volume conversion factors, and a surface compartment's size is used with BioNetGen's compartment convention. Record whether a membrane size represents an effective volume, for example surface area times an assigned width, and keep its units consistent with the amount convention. Dimension 2 alone does not supply that width.

For transport, a species-level location change can move a matched complex according to the compartment rules. A molecule-level location change selects the named molecule; `MoveConnected` specifies how unmentioned connected molecules in its original compartment accompany it. Check the generated products for complexes spanning several compartments, since movement can change the allowed topology.

B.9 Energy patterns

An energy-pattern line has the form `pattern energy_expression`. Its contribution is multiplied by the number of occurrences of the pattern, with pattern symmetry accounted for, and added to the species energy. The energies used by `Arrhenius` are dimensionless free energies.

```

begin energy patterns
  R(l!1).L(r!1) gbind
end energy patterns
begin reaction rules
  Bind: R(l) + L(r) <-> R(l!1).L(r!1) Arrhenius(phi,Ea)
end reaction rules

```

`Arrhenius(phi,Ea)` supplies both directions of a reversible rule, so it has no second comma-separated reverse rate law. For an energy difference Δg , the numerical coefficients are

$$k_+ = e^{-E_a - \phi \Delta g}, \quad k_- = e^{-E_a + (1-\phi) \Delta g}.$$

The energy difference fixes their ratio, while `Ea` sets the kinetic scale and `phi` allocates the energy change between the directions. Association energies require a concentration reference, and the rates require a time reference. In this example `gbind = ln(KD/Cref)` uses the micromolar reference recorded in the file. The parameter `Cref` is part of that expression; it does not independently request a unit conversion from BioNetGen. The older `setOption("energyBNG",1)` command is deprecated because energy-pattern syntax is enabled by default in the tested release.

B.10 Network generation

`generate_network({options})` iteratively applies the rules to the seed species and newly generated species. It writes an enumerated `.net` file when network generation succeeds.

Option	Purpose
<code>overwrite=>1</code>	Allow an existing output network to be replaced
<code>max_iter=>value</code>	Limit successive rounds of rule application
<code>max_agg=>value</code>	Limit the total number of molecules in a generated species
<code>max_stoich=>{R=>value,L=>value}</code>	Limit the number of each named molecule type in a generated species
<code>check_iso=>1</code>	Check molecular graph isomorphism when identifying species
<code>prefix=>"name", suffix=>"name"</code>	Control output names

An aggregate or stoichiometry limit changes which species and reactions are included. It therefore defines a truncated model whose predictions require a convergence check as the limit is increased. An iteration limit can also stop enumeration before the network closes. For a model with unbounded polymerization, network-free simulation avoids enumeration of the complete species set.

The generated network contains indexed parameters, functions, species, reactions, and observable groups. A reaction's comment identifies its originating rule. Its coefficient may contain a statistical factor for equivalent sites or identical reactants. Compare the generated elementary rate with the intended event rate before interpreting a rule parameter as a macroscopic association or dissociation constant.

B.11 Simulation actions

An action uses the form `simulate({method=>"method",options})`.

Method	Simulation
"ode"	Deterministic ODE integration of an enumerated network
"ssa"	Stochastic simulation of an enumerated network
"pla"	Partitioned-leaping approximation for a network
"psa"	Partial-scaling approximate stochastic network simulation
"nf"	NFsim simulation directly from the rule-based model

For network methods, generate the network before simulating. NFsim uses the molecular types, rules, and seeds exported to XML and can run without `generate_network()`. An NFsim input must satisfy the simulator's feature requirements; accepting a file in the BNGL parser does not establish support for every rate law.

Option	Meaning and applicability
<code>t_start, t_end</code>	Requested time interval; use an explicit start in multistage protocols
<code>n_steps</code>	Number of output intervals, with the initial point also reported
<code>sample_times=>[times]</code>	Requested output times; use this instead of <code>n_steps</code>
<code>atol, rtol</code>	Absolute and relative tolerances for ODE integration
<code>sparse=>1</code>	Sparse linear algebra for the ODE solver
<code>seed=>integer</code>	Random seed for stochastic methods
<code>suffix=>"name"</code>	Distinguish output files from separate runs
<code>continue=>1</code>	Append a network-simulation continuation using its current state and time; retain the same output prefix
<code>print_functions=>1</code>	Request global-function output
<code>steady_state=>1</code>	Enable the network ODE solver's steady-state stopping criterion
<code>get_final_state=>1</code>	Request the final NFsim state, including molecular species output

Output intervals are observation intervals, not the internal integration steps or stochastic event times. Specifying `n_steps` together with `sample_times` gives precedence to `n_steps` and produces a warning. Use distinct suffixes for independent runs and reset the molecular amounts between them. Set `t_start` explicitly when switching to NFsim after a previous simulation; NFsim otherwise uses the model's current time. NFsim reports elapsed time and rejects `continue=>1` in the tested release.

For reproducible stochastic runs, record the simulator version and seed. The same seed in SSA and NFsim does not imply the same trajectory, since the algorithms sample different event representations.

B.12 Parameter changes, protocols, and scans

Action	Effect
<code>setParameter("name","expression")</code>	Change a defined parameter
<code>setConcentration("complete_species","expression")</code>	Set the amount of an existing species
<code>addConcentration("complete_species","expression")</code>	Add to its current amount
<code>saveParameters("label")</code>	Save current parameter definitions under a label
<code>resetParameters("label")</code>	Restore those saved definitions
<code>saveConcentrations("label")</code>	Save the current species amounts
<code>resetConcentrations("label")</code>	Restore the saved amounts
<code>resetConcentrations()</code>	Restore default seed amounts
<code>setVolume("compartment","expression")</code>	Change a declared compartment size

Species addressed by concentration actions must already exist in the species list; an action does not create a missing species definition. Changing a seed parameter after simulation and restoring seed amounts is distinct from directly setting the current amount. Parameter and concentration saves are separate, so restore both when an experiment changes both. A concentration reset also leaves time management to the simulation action; specify the intended start explicitly.

```
saveConcentrations("start")
saveParameters("rates")
simulate({method=>"ode",t_end=>1,n_steps=>4,suffix=>"ode"})
setParameter("kp","0.4")
setConcentration("A(Y~u)","6")
addConcentration("A(Y~p)","1")
simulate({method=>"ode",t_start=>1,t_end=>2,n_steps=>4,continue=>1,suffix=>
  ↪ "ode"})
resetConcentrations("start")
resetParameters("rates")
```

`parameter_scan` runs a sequence of parameter values and records the final observables for each value:

```
parameter_scan({parameter=>"kp",par_min=>0.1,par_max=>0.3,n_scan_pts=>3,
  ↪ ,method=>"ode",t_end=>1,n_steps=>4,suffix=>"scan"})
```

Scan option	Purpose
<code>parameter=>"name"</code>	Defined parameter to vary
<code>par_min, par_max, n_scan_pts</code>	Endpoints and number of scan values
<code>log_scale=>1</code>	Logarithmic spacing; endpoints must be positive
<code>par_scan_vals=>[values]</code>	Explicit values instead of a regularly spaced range
<code>reset_conc=>1</code>	Start scan points from the saved starting amounts
<code>reset_conc=>0</code>	Carry final amounts into the next point
<code>method, t_end, n_steps</code>	Simulation options used at each scan point

Independent dose-response scans and scans that carry state between points answer different questions, particularly in models with multiple stable states. Restore parameters after a scan if subsequent actions require the original values. `bifurcate` automates increasing and decreasing scans with carried state; its result still depends on sufficient relaxation at each point.

`readFile({file=>"model.bngl"})` reads another BNGL file; `readFile({file=>"network.net"})` reads a generated network. Relative paths are interpreted within the execution environment, so keep supporting files with a documented run directory. SBML import is a separate translation workflow and can require additional Atomizer dependencies.

B.13 Exports, visualization, and output files

```
writeModel({suffix=>"model"})
writeNetwork({suffix=>"network"})
writeXML({suffix=>"xml"})
writeSBML({suffix=>"sbml"})
writeMfile({suffix=>"mfile"})
visualize({type=>"contactmap",suffix=>"contact"})
```

Action	Representation written
<code>writeModel()</code>	BNGL model definition
<code>writeNetwork()</code>	Enumerated <code>.net</code> reaction network
<code>writeXML()</code>	BioNetGen's rule-based XML, including NFsim input
<code>writeSBML()</code>	SBML representation of the reaction network
<code>writeSBMLMulti()</code>	Structured SBML Multi export, with feature-dependent support
<code>writeMfile()</code>	MATLAB simulation code
<code>writeMexfile()</code>	MATLAB interface and C source for a compiled solver
<code>writeMDL()</code>	MCell-oriented export for supported models
<code>visualize({type=>"contactmap"})</code>	Molecule types, components, and possible bonds
<code>visualize({type=>"ruleviz_pattern"})</code>	Reactant and product patterns of rules
<code>visualize({type=>"ruleviz_operation"})</code>	Graph operations performed by rules
<code>visualize({type=>"regulatory"})</code>	Relationships between rules and their structural requirements

Most writers accept output-name options such as `suffix`. Visualization writes graph files for a separate layout program. Regulatory graphs can use options including `groups`, `collapse`, and an `opts` file defining user groupings. An exporter may require network generation or reject constructs it cannot represent. `writeXML` and `writeSBML` both write XML files but encode different model representations; use distinct suffixes when requesting both.

Extension	Contents
<code>.bngl</code>	Rule-based model and requested actions
<code>.net</code>	Enumerated species, reactions, functions, and observable weights
<code>.gdat</code>	Time and observable values, usually with a <code>#</code> header
<code>.cdat</code>	Time and individual network-species amounts
<code>.fdat</code>	Global function values when written separately by a network backend; some backends append them to <code>.gdat</code>
<code>.species</code>	Final molecular species and counts from NFsim

Extension	Contents
<code>.scan</code>	Parameter values and final observables from a scan
<code>.graphml</code> , <code>.gml</code>	Graph representations produced by visualization; the tested contact-map export uses GraphML

Read the header to identify output columns; species indices can change when a model is edited or its network is regenerated. Use observables with stable names for analyses that compare model versions.

B.14 Diagnosing syntax and execution errors

Symptom	Check
Undefined parameter or function	Spelling, case, definition order, and parentheses on function calls
Missing component or internal state in a seed	Write a complete species consistent with its molecule type
Illegal bond wildcard in a seed	Replace pattern wildcards with a specified bond or an unbound component
Unmatched bond endpoint	Each explicit bond label must identify both endpoints within the expression
Missing reverse rate law	Supply both coefficients for <code><-></code> , except when using Arrhenius
Species not found in a concentration action	Generate the species or include it as a seed before addressing it
Unexpected receptor or complex count	Compare Molecules with Species and check overlapping patterns
Unexpected rate multiplicity	Inspect equivalent sites, repeated reactants, local-function scope, and generated statistical factors
No network closure	Check for unbounded aggregation or an iteration limit
Simulation interval error	Check the current time and supply explicit t_start and t_end
Unrecognized action option	Check the spelling and the options accepted by the selected method

Parser errors identify the file and line being read. Reduce the failing rule or pattern to a small executable model before changing the full model; the examples in this appendix provide starting points for checking matching, counting, and rate semantics.

B.15 Additional language constructs and source locations

Population molecule types, `population maps`, and `generate_hybrid_model` support hybrid particle/population workflows. They require a deliberate mapping between particle structures and population variables. Time-indexed tabular functions, including `tfun` in newer source snapshots, also have version-dependent syntax and backend support. Those specialized workflows should be checked against the distribution being used rather than inferred from a parser keyword.

The implementation files provide the detailed reference for less common constructs. All links below point to the source snapshot used for this draft.

- [Model blocks and core actions](#) define file processing, network generation, and parameter changes.
- [Molecular graphs](#) and [components](#) define pattern syntax and graph attributes.
- [Observables](#) define counting and quantifiers.
- [Reaction rules](#) and [rate laws](#) define transformations and rate modifiers.
- [Expressions](#) define mathematical operators and built-in functions.
- [Simulation actions](#) define method options, protocols, and scans.

The [upstream quick-reference guide](#) provides earlier examples of the language. Its installation instructions and some simulator details reflect the release available when it was written.