Computational Systems Biology in MATLAB and Python

An open primer

Karthik Raman

2026-11-02

Table of contents

Preface 4
How the examples are checked . . . 4
Licence . . . 4
Acknowledgements . . . 4

1 Getting started 5
1.1 Getting the code . . . 5
1.2 Python . . . 5
1.3 MATLAB . . . 6
1.4 What the examples write . . . 6

2 Networks 7
2.1 Network parameters . . . 7
2.2 Random and small-world networks . . . 12
2.3 Scale-free networks . . . 19
2.4 Exercises . . . 26
2.5 Going further . . . 26

3 Dynamic models with ODEs 27
3.1 Solving an ODE: Euler's method . . . 27
3.2 Enzyme kinetics: mass action vs Michaelis–Menten . . . 33
3.3 Stiff equations . . . 37
3.4 Exercises . . . 42
3.5 Going further . . . 42

4 Parameter estimation 43
4.1 The model and the data . . . 43
4.2 Fitting by least squares . . . 44
4.3 The error landscape, and grid search . . . 50
4.4 Evolutionary algorithms . . . 54
4.5 Exercises . . . 60
4.6 Going further . . . 61

5 Boolean networks 62
5.1 Transfer functions, states and attractors . . . 62
5.2 The mammalian cell cycle . . . 68
5.3 Exercises . . . 75
5.4 Going further . . . 75

6 Stochastic simulation **76**
6.1 Gillespie's algorithm . . . 76
6.2 Production and degradation of an mRNA . . . 77
6.3 Noise in gene expression . . . 82
6.4 Exercises . . . 87
6.5 Going further . . . 87

7 Constraint-based modelling **88**
7.1 The toy network . . . 88
7.2 Flux balance analysis . . . 89
7.3 Flux variability analysis . . . 93
7.4 A genome-scale example: *E. coli* core . . . 99
7.5 Exercises . . . 104
7.6 Going further . . . 104

8 Perturbations to metabolic networks **105**
8.1 Single deletions . . . 105
8.2 Synthetic lethals . . . 110
8.3 Exercises . . . 117
8.4 Going further . . . 117

References **118**

Preface

This primer is a hands-on companion to *An Introduction to Computational Systems Biology: Systems-Level Modelling of Cellular Networks* (Raman, 2021, CRC Press) and to the course BT5240 at IIT Madras. It can also be read on its own.

Each chapter takes one family of methods, explains the idea in a few paragraphs, and then works through examples. **Every example is given in both MATLAB and Python**, and the two versions produce the same numbers and the same figures. Pick your language in any code tab; the choice carries over to every other tab on the page.

The text is deliberately short. For the theory behind each method, every chapter points to the relevant chapter and section of the book.

How the examples are checked

Each example is a pair of standalone scripts, one `.m` and one `.py`, in the `examples/` folder of the repository. Every time the primer changes, both versions are run automatically and their key results are compared with each other, and with the values reported in the book where there are any. So the code you see on these pages is the code that was run to produce the figures beside it.

Licence

The text is available under CC BY 4.0 and the code under the MIT licence.

Acknowledgements

To be added.

1 Getting started

You need only one of the two languages to follow this primer. The examples are self-contained scripts that you can run as they are, and then modify.

1.1 Getting the code

Download the repository as a zip file from https://github.com/RamanLab/sysbio-primer, or clone it with Git:

```
git clone https://github.com/RamanLab/sysbio-primer.git
cd sysbio-primer
```

1.2 Python

Python 3.10 or newer is needed. From the repository folder, create a virtual environment and install the packages the examples use (NumPy, SciPy, Matplotlib, NetworkX and COBRApy), plus a tiny helper package, `sbp`, that the examples use to save their results and figures:

```
python -m venv .venv
source .venv/bin/activate        # on Windows: .venv\Scripts\activate
pip install -r requirements.txt
pip install -e .
```

Then run any example, for instance:

```
python examples/constraint-based/toy_fba/toy_fba.py
```

1.3 MATLAB

The examples use core MATLAB (R2021a or newer) and, where noted at the top of a script, standard MathWorks toolboxes such as the Optimization Toolbox. Some constraint-based examples need the COBRA Toolbox; each chapter says so where this applies.

Start MATLAB in the repository folder and run, once per session:

```
setup_primer      % adds the small +sbp helper package to the path
```

Then run any example, for instance `run examples/constraint-based/toy_fba/toy_fba.m`.

1.4 What the examples write

Each script prints its main results and writes two kinds of output next to itself, under `out/matlab/` or `out/python/`:

- figures (`fig-*.png`), which are the figures shown in this primer;
- `results.json`, with the key numbers. These files are what the automated checks compare between the two languages.

2 Networks

Many biological systems are naturally described as networks: proteins that bind one another, genes that regulate one another, metabolites linked by reactions. Before building detailed dynamic models of such systems, a great deal can be learnt from the structure of the network alone. This chapter computes the standard network parameters for a small example, and then generates the three classic network models, random, small-world and scale-free, to see how they differ.

> **In the book**
>
> Chapter 2, *Introduction to Graph Theory*: graphs, their representation as adjacency matrices and edge lists, and paths. Chapter 3, *Structure of Networks*: network parameters such as degree, clustering, path length and centrality (§3.1), and the canonical network models of Erdős and Rényi, Watts and Strogatz, and Barabási and Albert (§3.2). Communities, motifs and perturbations (§3.3–3.5) are left as exercises here.

2.1 Network parameters

A network, or graph, is a set of nodes joined by edges. For an undirected network of n nodes, the **adjacency matrix A** has $a_{ij} = 1$ if nodes i and j are joined, and 0 otherwise. Most network parameters follow directly from it:

- the **degree** $k_i = \sum_j a_{ij}$, the number of neighbours of node i;
- the **clustering coefficient** C_i, the fraction of pairs of neighbours of i that are themselves joined. Because $(\mathbf{A}^3)_{ii}$ counts the closed walks of length three from i, which is twice the number of triangles through i, $C_i = (\mathbf{A}^3)_{ii}/k_i(k_i - 1)$;
- the **shortest-path length** d_{ij} between every pair of nodes, from which follow the **diameter** (the largest d_{ij}) and the **characteristic path length** (the mean d_{ij} over all pairs);
- two measures of how central a node is: **closeness**, $(n-1)/\sum_j d_{ij}$, the reciprocal of the mean distance to all other nodes, and **betweenness**, the number of shortest paths between other pairs of nodes that pass through the node (a path shared by several equally short alternatives counts fractionally).

In MATLAB, networks are `graph` objects (or `digraph` for directed networks), part of core MATLAB. In Python, the standard package is NetworkX. Both provide shortest paths

and centrality measures; we compute the clustering coefficient and closeness from their definitions, so that both languages use exactly the same formulas as the book.

Table 2.1: Working with networks in MATLAB and NetworkX.

	MATLAB	Python (NetworkX)
Build a network	`G = graph(s, t)`	`G = nx.Graph(); G.add_edges_from(edges)`
Adjacency matrix	`adjacency(G)` (sparse)	`nx.to_numpy_array(G)`
Degrees	`degree(G)`	`dict(G.degree())`
Shortest-path lengths	`distances(G)`	`nx.floyd_warshall_numpy(G)`
Betweenness	`centrality(G, 'betweenness')`	`nx.betweenness_centrality(G, normalized=False)`

Our example is a small network of ten proteins, two tightly knit groups joined through protein E.

MATLAB

Core MATLAB only.

```
%% Basic network parameters for a small undirected network
% Ten proteins, A to J, in two tightly knit groups joined through E.
% We compute degrees, clustering coefficients, shortest-path lengths and
% two centrality measures, using the definitions in the book (Ch. 3).
% Requires: core MATLAB only (graph objects, R2015b or newer).

nodes = cellstr(('A':'J')');
edges = {'A' 'B'; 'A' 'C'; 'A' 'D'; 'B' 'C'; 'B' 'D'; 'C' 'D'; 'D' 'E';
         'E' 'F'; 'F' 'G'; 'F' 'H'; 'G' 'H'; 'G' 'I'; 'H' 'I'; 'I' 'J'};
G = graph(edges(:, 1), edges(:, 2));
G = reordernodes(G, nodes);   % keep nodes in the order A, B, ..., J
n = numnodes(G);

% Adjacency matrix and degrees (book Eq. 3.1)
A = full(adjacency(G));
deg = sum(A, 2);

% Clustering coefficient (Eq. 3.6): the fraction of pairs of neighbours
% that are themselves connected; (A^3)_ii counts each triangle twice
triangles = diag(A^3) / 2;
pairs = deg .* (deg - 1) / 2;
clustering = zeros(n, 1);
clustering(pairs > 0) = triangles(pairs > 0) ./ pairs(pairs > 0);

% Shortest-path lengths between all pairs of nodes
```

```
D = distances(G);
diameter = max(D(:));                                   % Eq. 3.11
char_path_length = mean(D(triu(true(n), 1)));           % Eq. 3.12
closeness = (n - 1) ./ sum(D, 2);                       % Eq. 3.14

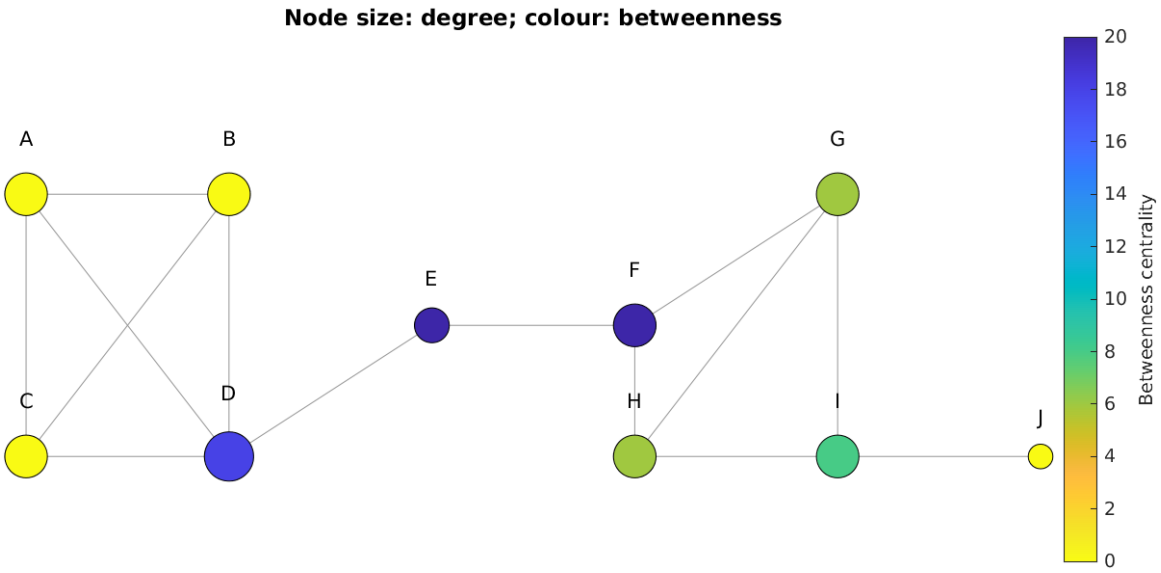
% Betweenness centrality (Eq. 3.17), counting each pair of nodes once
betweenness = centrality(G, 'betweenness');

fprintf('%d nodes, %d edges, mean degree %.2f\n', n, numedges(G), ...
    mean(deg));
fprintf(['mean clustering coefficient %.4f, diameter %g, ' ...
    'characteristic path length %.4f\n'], mean(clustering), diameter, ...
    char_path_length);
fprintf('node  degree  clustering  closeness  betweenness\n');
for i = 1:n
    fprintf('  %s   %4g    %8.3f   %8.3f   %9.2f\n', nodes{i}, deg(i), ...
        clustering(i), closeness(i), betweenness(i));
end

%% Plot: node size by degree, colour by betweenness, fixed positions
x = [0 1 0 1 2 3 4 3 4 5];
y = [1 1 0 0 0.5 0.5 1 0 0 0];
fig = figure('Position', [100 100 700 320]);
hold on
for e = 1:size(edges, 1)
    i = find(strcmp(nodes, edges{e, 1}));
    j = find(strcmp(nodes, edges{e, 2}));
    plot([x(i) x(j)], [y(i) y(j)], '-', 'Color', [0.6 0.6 0.6]);
end
scatter(x, y, 150 * deg, betweenness, 'filled', 'MarkerEdgeColor', 'k');
text(x, y + 0.12 + 0.03 * deg', nodes, 'HorizontalAlignment', 'center');
colormap(flipud(parula));
cb = colorbar;
cb.Label.String = 'Betweenness centrality';
title('Node size: degree; colour: betweenness');
axis off
ylim([-0.4 1.6]);
sbp.save_figure(fig, mfilename('fullpath'), 'fig-graph-basics');

results.degree = deg';
results.clustering = clustering';
results.mean_clustering = mean(clustering);
results.diameter = diameter;
results.char_path_length = char_path_length;
results.closeness = closeness';
results.betweenness = betweenness';
```

```
sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs NetworkX.

```
"""Basic network parameters for a small undirected network.

Ten proteins, A to J, in two tightly knit groups joined through E.
We compute degrees, clustering coefficients, shortest-path lengths and
two centrality measures, using the definitions in the book (Ch. 3).
"""
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import sbp

nodes = list("ABCDEFGHIJ")
edges = [("A", "B"), ("A", "C"), ("A", "D"), ("B", "C"), ("B", "D"),
         ("C", "D"), ("D", "E"), ("E", "F"), ("F", "G"), ("F", "H"),
         ("G", "H"), ("G", "I"), ("H", "I"), ("I", "J")]
G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
n = G.number_of_nodes()

# Adjacency matrix and degrees (book Eq. 3.1)
A = nx.to_numpy_array(G, nodelist=nodes)
degree = A.sum(axis=1)
```

```
# Clustering coefficient (Eq. 3.6): the fraction of pairs of neighbours
# that are themselves connected; (A^3)_ii counts each triangle twice
triangles = np.diag(A @ A @ A) / 2
pairs = degree * (degree - 1) / 2
clustering = np.divide(triangles, pairs, out=np.zeros(n), where=pairs > 0)

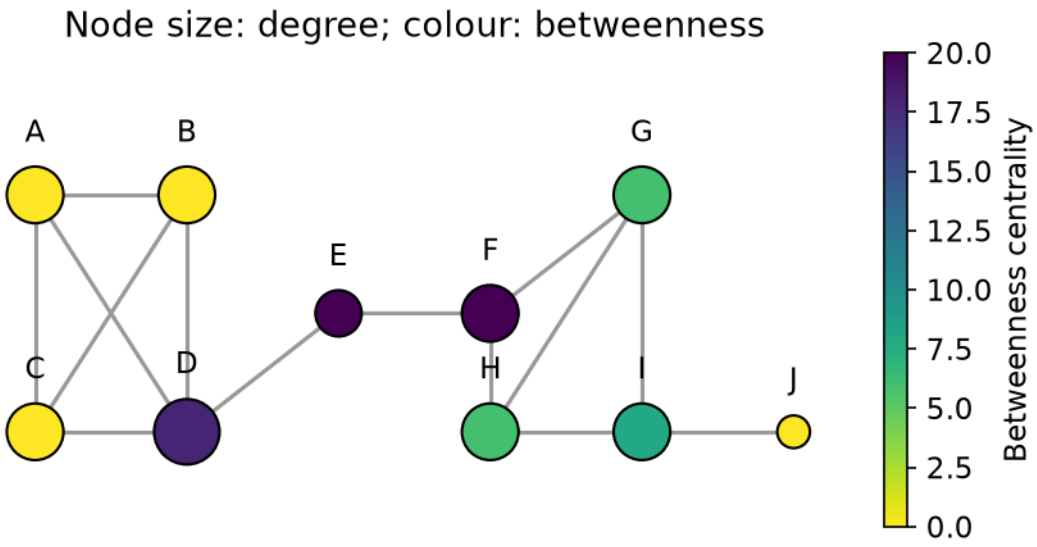
# Shortest-path lengths between all pairs of nodes
D = nx.floyd_warshall_numpy(G, nodelist=nodes)
diameter = D.max()                                          # Eq. 3.11
char_path_length = D[np.triu_indices(n, k=1)].mean()       # Eq. 3.12
closeness = (n - 1) / D.sum(axis=1)                         # Eq. 3.14

# Betweenness centrality (Eq. 3.17), counting each pair of nodes once
bc = nx.betweenness_centrality(G, normalized=False)
betweenness = np.array([bc[v] for v in nodes])

print(f"{n} nodes, {G.number_of_edges()} edges, mean degree "
      f"{degree.mean():.2f}")
print(f"mean clustering coefficient {clustering.mean():.4f}, diameter "
      f"{diameter:g}, characteristic path length {char_path_length:.4f}")
print("node  degree  clustering  closeness  betweenness")
for i, v in enumerate(nodes):
    print(f"  {v}   {degree[i]:4g}    {clustering[i]:8.3f}   "
          f"{closeness[i]:8.3f}   {betweenness[i]:9.2f}")

# --- Plot: node size by degree, colour by betweenness, fixed positions
x = [0.0, 1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 3.0, 4.0, 5.0]
y = [1.0, 1.0, 0.0, 0.0, 0.5, 0.5, 1.0, 0.0, 0.0, 0.0]
sbp.style()
fig, ax = plt.subplots(figsize=(7, 3.2))
idx = {v: i for i, v in enumerate(nodes)}
for u, w in edges:
    ax.plot([x[idx[u]], x[idx[w]]], [y[idx[u]], y[idx[w]]], "-",
            color="#999999", zorder=1)
sc = ax.scatter(x, y, s=150 * degree, c=betweenness, cmap="viridis_r",
                edgecolors="k", zorder=2)
for i, v in enumerate(nodes):
    ax.annotate(v, (x[i], y[i]), xytext=(0, 14 + 2 * degree[i]),
                textcoords="offset points", ha="center")
fig.colorbar(sc, ax=ax, label="Betweenness centrality")
ax.set_title("Node size: degree; colour: betweenness")
ax.set_axis_off()
ax.set_ylim(-0.4, 1.6)
sbp.save_figure(fig, __file__, "fig-graph-basics")
```

```
sbp.save_results(__file__, {
    "degree": degree, "clustering": clustering,
    "mean_clustering": clustering.mean(), "diameter": diameter,
    "char_path_length": char_path_length, "closeness": closeness,
    "betweenness": betweenness,
})
sbp.show()
```



The different measures pick out different nodes. D has the largest degree, but E and F, with only two and three neighbours, carry the most shortest paths: every path between the two groups passes through both of them, so removing either would split the network in two. A, B and C, inside a complete group of four, have the highest clustering coefficient and no betweenness at all. Which measure matters depends on the question being asked; for example, betweenness, more than degree, often marks nodes whose removal disrupts communication across a network (book, §3.1.3 and §3.5).

2.2 Random and small-world networks

To judge whether a property of a real network is remarkable, we need something to compare it with. The simplest null model is the **Erdős–Rényi** random network, in which each of the $\binom{n}{2}$ possible edges is present independently with probability p. Its degrees follow a binomial distribution, close to a Poisson distribution for large n; its clustering coefficient is about p, which is very small for a sparse network; and its paths are short, growing only as $\ln n$.

Real networks often combine short paths with high clustering. Watts and Strogatz showed how this can arise (Watts and Strogatz 1998). Start from a **ring lattice**, in which each node is joined to its k nearest neighbours on either side: clustering is high, but paths are

long, since getting to the far side of the ring takes many steps. Then rewire each edge, with probability p, to a randomly chosen node. The code below computes the mean clustering coefficient $C(p)$ and characteristic path length $L(p)$ as p rises from 10^{-4} to 1, for networks of 500 nodes with ten neighbours each, averaging over ten networks at each p. This is also the first problem of the BT5240 network assignment.

MATLAB

Core MATLAB only.

```
%% Small-world networks (Watts and Strogatz, 1998)
% Start from a ring lattice of n nodes, each linked to its k nearest
% neighbours on either side, and rewire each edge with probability p. A
% little rewiring makes paths short, as in a random network, while keeping
% the high clustering of the lattice. We also compare with Erdős-Rényi
% random networks with the same numbers of nodes and edges.
% Requires: core MATLAB only.

n = 500; k = 5;                    % 500 nodes, 2k = 10 neighbours each
probs = logspace(-4, 0, 13);       % rewiring probabilities
reps = 10;                         % networks averaged for each p
rng(1);

[C0, L0] = clustering_and_path_length(ring_lattice(n, k));
C = zeros(reps, numel(probs));
L = zeros(reps, numel(probs));
for r = 1:reps
    for i = 1:numel(probs)
        [C(r, i), L(r, i)] = clustering_and_path_length( ...
            watts_strogatz(n, k, probs(i)));
    end
end
C_ratio = mean(C, 1) / C0;
L_ratio = mean(L, 1) / L0;

p_er = n * k / (n * (n - 1) / 2);  % same expected number of edges, n*k
er = zeros(reps, 2);
for r = 1:reps
    [er(r, 1), er(r, 2)] = clustering_and_path_length(erdos_renyi(n,
 ↪  p_er));
end
er = mean(er, 1);

fprintf('Ring lattice: C = %.4f, L = %.4f\n', C0, L0);
for i = 1:numel(probs)
    fprintf('  p = %8.4f: C(p)/C(0) = %.3f, L(p)/L(0) = %.3f\n', ...
        probs(i), C_ratio(i), L_ratio(i));
```

```
end
fprintf('Erdős-Rényi, same n and m: C = %.4f, L = %.4f\n', er(1), er(2));

%% Plot
fig = figure;
semilogx(probs, C_ratio, 'o-', 'Color', [0.122 0.467 0.706]);
hold on
semilogx(probs, L_ratio, 's-', 'Color', [1.000 0.498 0.055]);
xlabel('Rewiring probability p'); ylabel('Relative to the ring lattice');
title(sprintf('Watts-Strogatz networks, n = %d, %d neighbours', n, 2 * k));
legend({'C(p) / C(0)', 'L(p) / L(0)'}, 'Location', 'southwest', ...
    'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-small-world');

results.lattice = struct('C', C0, 'L', L0);
for p = [0.1 1]
    i = find(abs(probs - p) < 1e-12);
    name = strrep(sprintf('p_%g', p), '.', '_');
    results.ws.(name) = struct('C_ratio', C_ratio(i), ...
        'L_ratio', L_ratio(i));
end
results.er = struct('C', er(1), 'L', er(2));
results.probs = probs;
results.C_ratio = C_ratio;
results.L_ratio = L_ratio;
sbp.save_results(mfilename('fullpath'), results);

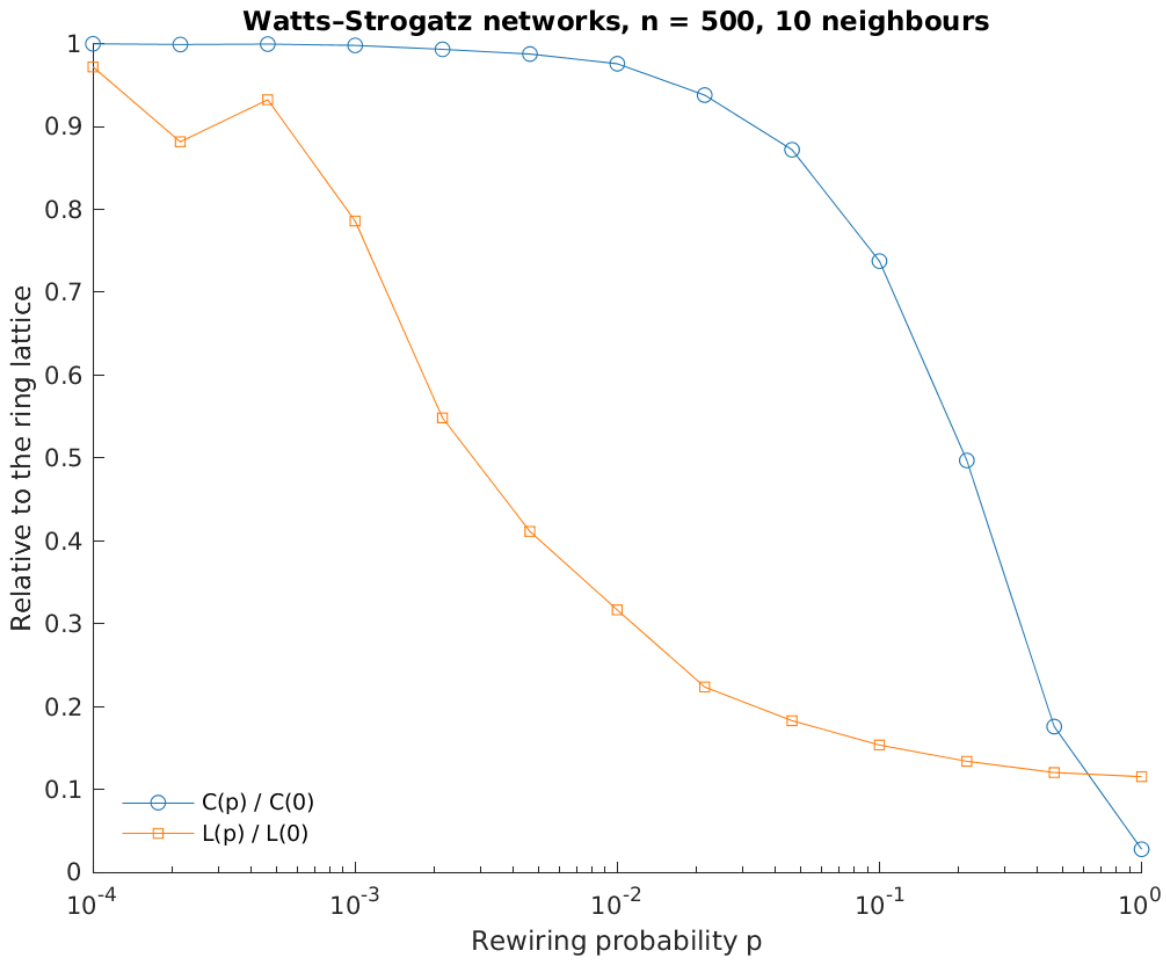
%% Local functions
function A = ring_lattice(n, k)
A = false(n);
idx = (1:n)';
for j = 1:k
    nb = mod(idx - 1 + j, n) + 1;
    A(sub2ind([n n], idx, nb)) = true;
    A(sub2ind([n n], nb, idx)) = true;
end
end

function A = watts_strogatz(n, k, p)
% Rewire each lattice edge (i, i+j) with probability p, moving its far
% end to a random node, avoiding self-loops and duplicate edges.
A = ring_lattice(n, k);
for j = 1:k
    for i = 1:n
        old = mod(i - 1 + j, n) + 1;
```

```
        if A(i, old) && rand < p
            new = randi(n);                 % draw until the new end is
            while new == i || A(i, new)     % neither i nor a neighbour
                new = randi(n);
            end
            A(i, old) = false; A(old, i) = false;
            A(i, new) = true;  A(new, i) = true;
        end
    end
end
end

function A = erdos_renyi(n, p)
upper = triu(rand(n) < p, 1);
A = upper | upper';
end

function [C, L] = clustering_and_path_length(A)
% Mean clustering coefficient, and mean shortest-path length over all
% pairs of nodes that are connected by some path.
A = double(A);
deg = sum(A, 2);
tri = sum((A * A) .* A, 2) / 2;
pairs = deg .* (deg - 1) / 2;
c = zeros(size(deg));
c(pairs > 0) = tri(pairs > 0) ./ pairs(pairs > 0);
C = mean(c);
D = distances(graph(A));
d = D(triu(true(size(A)), 1));
L = mean(d(isfinite(d)));
end
```



Python

Needs NumPy and SciPy.

```
"""Small-world networks (Watts and Strogatz, 1998).

Start from a ring lattice of n nodes, each linked to its k nearest
neighbours on either side, and rewire each edge with probability p. A
little rewiring makes paths short, as in a random network, while keeping
the high clustering of the lattice. We also compare with Erdős-Rényi
random networks with the same numbers of nodes and edges.
"""
import numpy as np
from scipy.sparse.csgraph import shortest_path
import matplotlib.pyplot as plt
import sbp

n, k = 500, 5                       # 500 nodes, 2k = 10 neighbours each
probs = np.logspace(-4, 0, 13)      # rewiring probabilities
```

```
reps = 10                          # networks averaged for each p
rng = np.random.default_rng(1)


def ring_lattice(n, k):
    A = np.zeros((n, n), dtype=bool)
    for j in range(1, k + 1):
        idx = np.arange(n)
        A[idx, (idx + j) % n] = A[(idx + j) % n, idx] = True
    return A


def watts_strogatz(n, k, p):
    """Rewire each lattice edge (i, i+j) with probability p, moving its far
    end to a random node, avoiding self-loops and duplicate edges."""
    A = ring_lattice(n, k)
    for j in range(1, k + 1):
        for i in range(n):
            old = (i + j) % n
            if A[i, old] and rng.random() < p:
                new = rng.integers(n)          # draw until the new end is
                while new == i or A[i, new]:   # neither i nor a neighbour
                    new = rng.integers(n)
                A[i, old] = A[old, i] = False
                A[i, new] = A[new, i] = True
    return A


def erdos_renyi(n, p):
    upper = np.triu(rng.random((n, n)) < p, 1)
    return upper | upper.T


def clustering_and_path_length(A):
    """Mean clustering coefficient, and mean shortest-path length over all
    pairs of nodes that are connected by some path."""
    A = A.astype(float)
    deg = A.sum(axis=1)
    tri = ((A @ A) * A).sum(axis=1) / 2
    pairs = deg * (deg - 1) / 2
    C = np.divide(tri, pairs, out=np.zeros(len(A)), where=pairs > 0).mean()
    D = shortest_path(A, unweighted=True, directed=False)
    d = D[np.triu_indices(len(A), k=1)]
    return C, d[np.isfinite(d)].mean()
```

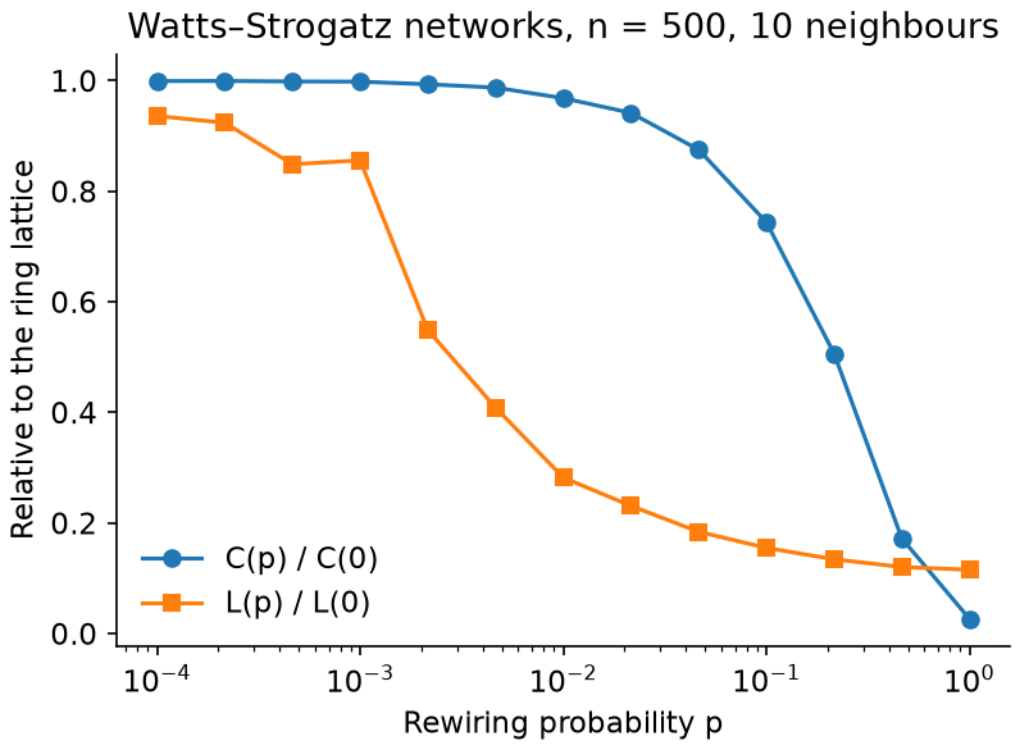
```
C0, L0 = clustering_and_path_length(ring_lattice(n, k))
C, L = np.zeros((reps, probs.size)), np.zeros((reps, probs.size))
for r in range(reps):
    for i, p in enumerate(probs):
        C[r, i], L[r, i] = clustering_and_path_length(watts_strogatz(n, k,
 ↪  p))
C_ratio, L_ratio = C.mean(axis=0) / C0, L.mean(axis=0) / L0

p_er = n * k / (n * (n - 1) / 2)  # same expected number of edges, n*k
er = np.array([clustering_and_path_length(erdos_renyi(n, p_er))
               for _ in range(reps)]).mean(axis=0)

print(f"Ring lattice: C = {C0:.4f}, L = {L0:.4f}")
for p, c, l in zip(probs, C_ratio, L_ratio):
    print(f"  p = {p:8.4f}: C(p)/C(0) = {c:.3f}, L(p)/L(0) = {l:.3f}")
print(f"Erdős-Rényi, same n and m: C = {er[0]:.4f}, L = {er[1]:.4f}")

# --- Plot
sbp.style()
fig, ax = plt.subplots()
ax.semilogx(probs, C_ratio, "o-", color="#1f77b4", label="C(p) / C(0)")
ax.semilogx(probs, L_ratio, "s-", color="#ff7f0e", label="L(p) / L(0)")
ax.set_xlabel("Rewiring probability p")
ax.set_ylabel("Relative to the ring lattice")
ax.set_title(f"Watts-Strogatz networks, n = {n}, {2 * k} neighbours")
ax.legend(frameon=False, loc="lower left")
sbp.save_figure(fig, __file__, "fig-small-world")

at = {f"p_{p:g}".replace(".", "_"): i
      for i, p in enumerate(probs) if p in (0.1, 1.0)}
sbp.save_results(__file__, {
    "lattice": {"C": C0, "L": L0},
    "ws": {name: {"C_ratio": C_ratio[i], "L_ratio": L_ratio[i]}
           for name, i in at.items()},
    "er": {"C": er[0], "L": er[1]},
    "probs": probs, "C_ratio": C_ratio, "L_ratio": L_ratio,
})
sbp.show()
```



The two curves fall at very different rewiring probabilities. Rewiring 1% of the edges already shortens the typical path by about 70%, because each rewired edge is a shortcut across the ring, while clustering hardly changes, because each rewired edge removes only a few of the many triangles around it. Between $p \approx 0.001$ and $p \approx 0.1$, the networks are **small worlds**: clustered like a lattice, with paths almost as short as in a random network. At $p = 1$, they are essentially random, and their clustering coefficient and path length are close to those of Erdős–Rényi networks with the same numbers of nodes and edges (about 0.02 and 2.9).

Since these networks are random, MATLAB and Python, which draw different random numbers, give slightly different results; the automated checks compare the averages, within 10%.

2.3 Scale-free networks

Erdős–Rényi and small-world networks share one feature that many biological networks do not: most nodes have roughly the same degree. In protein interaction and metabolic networks, by contrast, most nodes have very few neighbours while a few **hubs** have very many, and the degree distribution has a long tail, close to a power law, $P(k) \propto k^{-\gamma}$. Such networks are called **scale-free**.

Barabási and Albert proposed a simple mechanism (Barabási and Albert 1999). The network grows one node at a time, and each new node attaches to m existing nodes, chosen with

probability proportional to their degree: **preferential attachment**, or "the rich get richer". For large networks this gives $P(k) = 2m(m+1)/k(k+1)(k+2)$, which falls as k^{-3}.

There is a neat way to choose a node with probability proportional to its degree: keep a list of the two ends of every edge, so that each node appears in it as many times as it has links, and pick an entry of that list uniformly at random.

MATLAB

Core MATLAB only.

```
%% Scale-free networks by preferential attachment (Barabási and Albert)
% Nodes are added one at a time; each new node links to m existing nodes,
% chosen with probability proportional to their degree. The resulting
% degree distribution has a heavy tail, P(k) ~ k^-3, with a few highly
% connected hubs. We compare it with an Erdős-Rényi network of the same
% size and mean degree.
% Requires: core MATLAB only.

n = 2000; m = 2;
reps = 10;
rng(1);

k_max = 200;
P_ba = zeros(1, k_max + 1); P_er = zeros(1, k_max + 1);
max_deg_ba = zeros(1, reps); max_deg_er = zeros(1, reps);
for r = 1:reps
    d = degrees(barabasi_albert(n, m), n);
    P_ba = P_ba + histcounts(min(d, k_max), -0.5:k_max + 0.5) / n / reps;
    max_deg_ba(r) = max(d);
    % Erdős-Rényi with the same expected mean degree
    mean_k = mean(d);
    upper = triu(rand(n) < mean_k / (n - 1), 1);
    d = sum(upper | upper', 2)';
    P_er = P_er + histcounts(min(d, k_max), -0.5:k_max + 0.5) / n / reps;
    max_deg_er(r) = max(d);
end

k = 0:k_max;
fprintf('Mean degree %.4f\n', mean_k);
fprintf(['Largest degree, averaged over %d networks: preferential ' ...
    'attachment %.1f, Erdős-Rényi %.1f\n'], reps, mean(max_deg_ba), ...
    mean(max_deg_er));
fprintf(['Fraction of nodes with degree >= 10: %.4f (theory for large ' ...
    'n: %.4f); Erdős-Rényi %.4f\n'], sum(P_ba(11:end)), ...
    m * (m + 1) / (10 * 11), sum(P_er(11:end)));
```

```
%% Plot. Right: the fraction of nodes with degree at least k, which is
% much less noisy in the tail than P(k) itself
ccdf_ba = fliplr(cumsum(fliplr(P_ba)));
ccdf_er = fliplr(cumsum(fliplr(P_er)));
blue = [0.122 0.467 0.706]; orange = [1.000 0.498 0.055];
fig = figure('Position', [100 100 1000 400]);
subplot(1, 2, 1)
plot(k, P_ba, 'o', 'Color', blue, 'MarkerFaceColor', blue, 'MarkerSize',
 ↪  4);
hold on
plot(k, P_er, 's', 'Color', orange, 'MarkerFaceColor', orange, ...
    'MarkerSize', 4);
xlim([-0.5 30]);
xlabel('Degree k'); ylabel('Fraction of nodes with degree k');
title('Degree distribution');
legend({'preferential attachment', 'Erdős-Rényi'}, 'Box', 'off');
box off

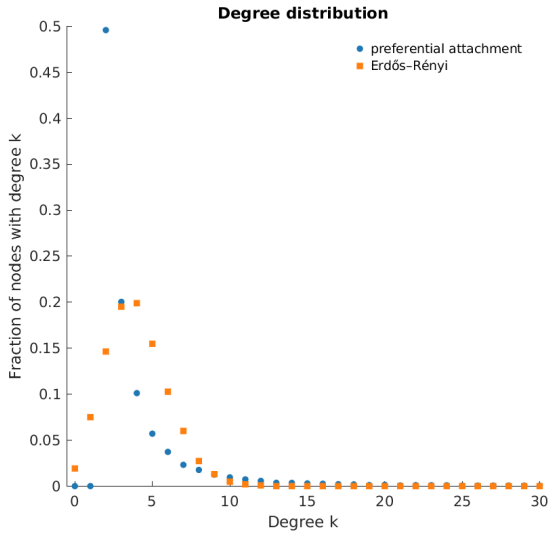
subplot(1, 2, 2)
ok = ccdf_ba > 0 & k > 0;
loglog(k(ok), ccdf_ba(ok), 'o', 'Color', blue, 'MarkerFaceColor', blue, ...
    'MarkerSize', 4);
hold on
ok = ccdf_er > 0 & k > 0;
loglog(k(ok), ccdf_er(ok), 's', 'Color', orange, ...
    'MarkerFaceColor', orange, 'MarkerSize', 4);
kk = k(m + 1:end);
loglog(kk, m * (m + 1) ./ (kk .* (kk + 1)), '-', 'Color', [0.6 0.6 0.6]);
ylim([1e-5 1.5]);
xlabel('Degree k'); ylabel('Fraction of nodes with degree \geq k');
title('Tail, on logarithmic axes');
legend({'preferential attachment', 'Erdős-Rényi', ...
    'theory, m(m+1)/k(k+1)'}, 'Location', 'southwest', 'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-scale-free');

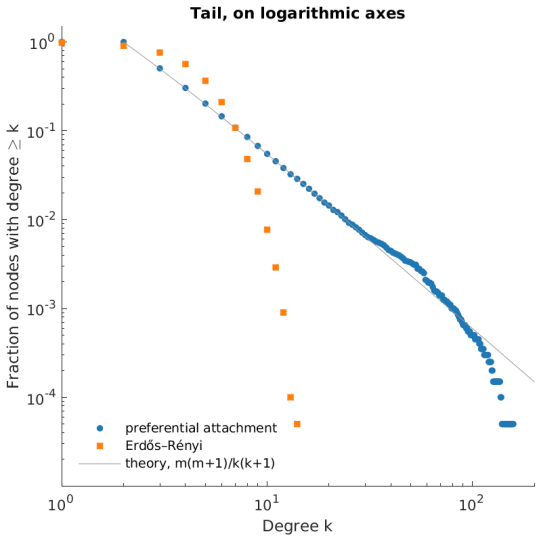
results.mean_degree = mean_k;
results.ba = struct('P2', P_ba(3), 'P3', P_ba(4), ...
    'P_ge_10', sum(P_ba(11:end)), 'mean_max_degree', mean(max_deg_ba));
results.er = struct('mean_max_degree', mean(max_deg_er));
results.er_P_ge_10 = sum(P_er(11:end));
results.P_ba = P_ba(1:51);
results.P_er = P_er(1:51);
sbp.save_results(mfilename('fullpath'), results);

%% Local functions
```

```
function edges = barabasi_albert(n, m)
% Edge list of a preferential-attachment network. 'ends' lists every
% edge end, so a node appears in it once per link: picking a uniformly
% random element of 'ends' picks a node with probability proportional
% to its degree. Nodes are numbered from 0, as in the Python version.
n_edges = m * (m + 1) / 2 + m * (n - m - 1);
edges = zeros(n_edges, 2);
e = 0;
for i = 0:m                                 % start from a clique
    for j = 0:i - 1
        e = e + 1;
        edges(e, :) = [i j];
    end
end
ends = reshape(edges(1:e, :)', 1, []);
ends = [ends, zeros(1, 2 * n_edges - numel(ends))];
n_ends = 2 * e;
for new = m + 1:n - 1
    targets = [];
    while numel(targets) < m
        t = ends(randi(n_ends));
        if ~ismember(t, targets)
            targets(end + 1) = t; %#ok<AGROW>
        end
    end
    for t = targets
        e = e + 1;
        edges(e, :) = [new t];
        ends(n_ends + 1:n_ends + 2) = [new t];
        n_ends = n_ends + 2;
    end
end
end

function d = degrees(edges, n)
d = accumarray(edges(:) + 1, 1, [n 1])';
end
```





Python

Needs NumPy.

```
"""Scale-free networks by preferential attachment (Barabási and Albert).

Nodes are added one at a time; each new node links to m existing nodes,
chosen with probability proportional to their degree. The resulting
degree distribution has a heavy tail, P(k) ~ k^-3, with a few highly
connected hubs. We compare it with an Erdős–Rényi network of the same
size and mean degree.
"""
import numpy as np
import matplotlib.pyplot as plt
import sbp

n, m = 2000, 2
reps = 10
rng = np.random.default_rng(1)


def barabasi_albert(n, m):
    """Edge list of a preferential-attachment network. 'ends' lists every
    edge end, so a node appears in it once per link: picking a uniformly
    random element of 'ends' picks a node with probability proportional
    to its degree."""
    edges = [(i, j) for i in range(m + 1) for j in range(i)]  # a clique
    ends = [v for e in edges for v in e]
    for new in range(m + 1, n):
        targets = set()
        while len(targets) < m:
```

```
            targets.add(ends[rng.integers(len(ends))])
        for t in targets:
            edges.append((new, t))
            ends += [new, t]
    return edges


def degrees(edges, n):
    return np.bincount(np.ravel(edges), minlength=n)

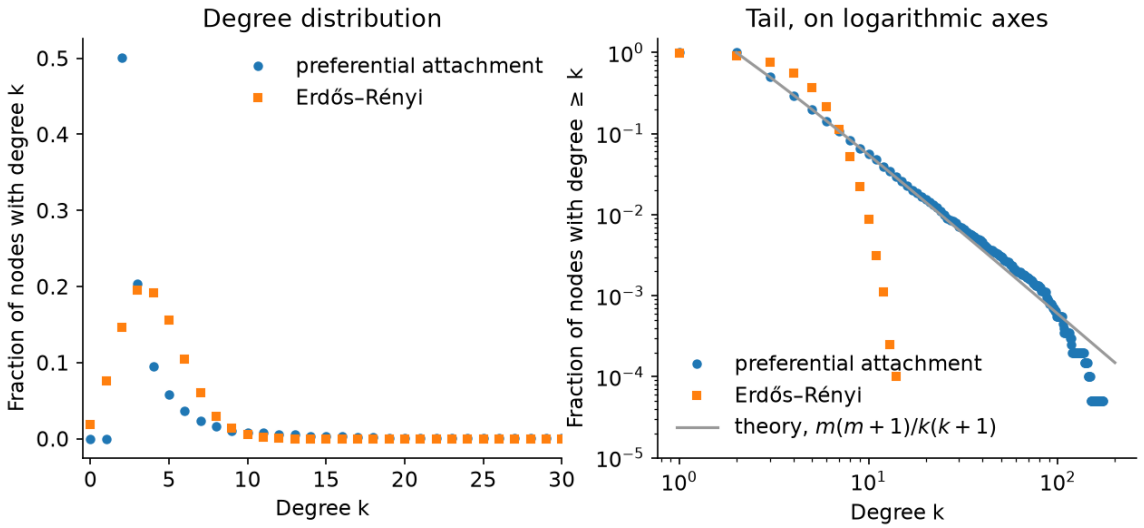

k_max = 200
P_ba, P_er = np.zeros(k_max + 1), np.zeros(k_max + 1)
max_deg_ba, max_deg_er = [], []
for _ in range(reps):
    d = degrees(barabasi_albert(n, m), n)
    P_ba += np.bincount(np.minimum(d, k_max), minlength=k_max + 1) / n /
 ↪  reps
    max_deg_ba.append(d.max())
    # Erdős-Rényi with the same expected mean degree
    mean_k = d.mean()
    upper = np.triu(rng.random((n, n)) < mean_k / (n - 1), 1)
    d = (upper | upper.T).sum(axis=1)
    P_er += np.bincount(np.minimum(d, k_max), minlength=k_max + 1) / n /
 ↪  reps
    max_deg_er.append(d.max())

k = np.arange(k_max + 1)
print(f"Mean degree {mean_k:.4f}")
print(f"Largest degree, averaged over {reps} networks: preferential "
      f"attachment {np.mean(max_deg_ba):.1f}, Erdős-Rényi "
      f"{np.mean(max_deg_er):.1f}")
print(f"Fraction of nodes with degree >= 10: {P_ba[10:].sum():.4f} "
      f"(theory for large n: {m * (m + 1) / (10 * 11):.4f}); "
      f"Erdős-Rényi {P_er[10:].sum():.4f}")

# --- Plot. Right: the fraction of nodes with degree at least k, which is
# much less noisy in the tail than P(k) itself
ccdf_ba, ccdf_er = P_ba[::-1].cumsum()[::-1], P_er[::-1].cumsum()[::-1]
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
ax1.plot(k, P_ba, "o", color="#1f77b4", ms=4, label="preferential
 ↪  attachment")
ax1.plot(k, P_er, "s", color="#ff7f0e", ms=4, label="Erdős-Rényi")
ax1.set_xlim(-0.5, 30)
ax1.set_xlabel("Degree k")
```

```
ax1.set_ylabel("Fraction of nodes with degree k")
ax1.set_title("Degree distribution")
ax1.legend(frameon=False)
ax2.loglog(k[1:][ccdf_ba[1:] > 0], ccdf_ba[1:][ccdf_ba[1:] > 0], "o",
           color="#1f77b4", ms=4, label="preferential attachment")
ax2.loglog(k[1:][ccdf_er[1:] > 0], ccdf_er[1:][ccdf_er[1:] > 0], "s",
           color="#ff7f0e", ms=4, label="Erdős-Rényi")
ax2.loglog(k[m:], m * (m + 1) / (k[m:] * (k[m:] + 1.0)), "-",
           color="#999999", label="theory, $m(m+1)/k(k+1)$")
ax2.set_ylim(1e-5, 1.5)
ax2.set_xlabel("Degree k")
ax2.set_ylabel("Fraction of nodes with degree $\\geq$ k")
ax2.set_title("Tail, on logarithmic axes")
ax2.legend(frameon=False, loc="lower left")
sbp.save_figure(fig, __file__, "fig-scale-free")

sbp.save_results(__file__, {
    "mean_degree": mean_k,
    "ba": {"P2": P_ba[2], "P3": P_ba[3], "P_ge_10": P_ba[10:].sum(),
           "mean_max_degree": np.mean(max_deg_ba)},
    "er": {"mean_max_degree": np.mean(max_deg_er)},
    "er_P_ge_10": P_er[10:].sum(),
    "P_ba": P_ba[:51], "P_er": P_er[:51],
})
sbp.show()
```



Both networks have 2000 nodes and a mean degree of about four. In the random network, no node has more than about a dozen neighbours. With preferential attachment, half of the nodes have only the minimum two links, and the most connected node has, on average

over ten networks, more than a hundred. On logarithmic axes the tail of the distribution follows the predicted straight line until it runs out of nodes.

Plotting the fraction of nodes with degree *at least* k (right panel), in place of the fraction with degree exactly k, avoids most of the noise in the tail, where only one or two nodes have each degree. Even so, deciding whether a real network is truly scale-free from a straight-looking line on a log–log plot is risky; the book discusses better tests (§3.6.1).

2.4 Exercises

1. Remove node E from the example network. Recompute all the parameters. What happens to the diameter, and why do the shortest-path functions report infinite distances?
2. Generate Erdős–Rényi networks with $n = 1000$ and mean degree from 0.2 to 3. Plot the size of the largest connected component (`conncomp` in MATLAB, `nx.connected_components` in Python) against the mean degree. What happens at a mean degree of 1?
3. For the scale-free network, remove nodes one at a time, either at random or in order of decreasing degree, and track the size of the largest connected component. Repeat for the Erdős–Rényi network. Which network is more robust to random failures, and which to targeted attack (book, §3.5.2)?
4. Download a biological network, for example a protein interaction network from https://networkrepository.com/, and compute its degree distribution, mean clustering coefficient and characteristic path length. Compare them with an Erdős–Rényi network with the same numbers of nodes and edges. (This is the second problem of the BT5240 network assignment.)
5. The book's code repository has a function `barabasi_albert.m` for $m = 1$. What kind of network does it always produce, and what is its clustering coefficient?

2.5 Going further

Detecting communities and network motifs (book, §3.3–3.4) builds on the same tools, together with randomised networks that preserve each node's degree. Chapter 4 of the book applies these ideas to biological questions, from predicting essential genes to finding disease modules.

3 Dynamic models with ODEs

A dynamic model describes how the levels of molecules, such as metabolites, mRNAs or proteins, change over time. Most such models are systems of ordinary differential equations (ODEs),

$$\frac{dx_i}{dt} = f_i(x_1, x_2, \dots, x_n;\ \Theta), \qquad i = 1, 2, \dots, n,$$

where x_i is the concentration of species i, f_i is its rate law, and Θ is the set of parameters, such as rate constants. This chapter covers the two halves of working with such models: writing the equations down, and solving them numerically, including what can go wrong.

> **In the book**
>
> Chapter 5, *Introduction to Dynamic Modelling*: mass-action kinetics (§5.2), enzyme kinetics and the Michaelis–Menten model (§5.3), solving ODEs (§5.5) and stiff equations (§5.6.1). Appendix E covers ODE solvers in MATLAB.

3.1 Solving an ODE: Euler's method

Few ODE models can be solved on paper, so we solve them numerically. The simplest method, due to Euler, replaces the derivative by a finite difference over a small time step Δt:

$$x(t + \Delta t) \approx x(t) + f(x(t))\,\Delta t.$$

Starting from the initial value x_0, we step forward in time, one Δt at a time. This is the **forward** (or explicit) Euler method. The **backward** (or implicit) Euler method evaluates f at the *end* of the step instead, $x(t + \Delta t) = x(t) + f(x(t + \Delta t))\,\Delta t$, which must be solved for $x(t + \Delta t)$ at every step.

We try both on an ODE with a known answer,

$$\frac{dx}{dt} = a - bx, \qquad x(0) = c, \qquad x(t) = \frac{a}{b} - \left(\frac{a}{b} - c\right) e^{-bt},$$

with $a = 20$, $b = 2$ and $c = 5$, so x rises from 5 towards its steady state $a/b = 10$.

MATLAB

Core MATLAB only.

```
%% Solving dx/dt = a - b*x with the forward and backward Euler methods
% This simple ODE has an exact solution, so we can see how the step size
% dt affects accuracy and, for forward Euler, stability.
% Requires: core MATLAB only.

a = 20; b = 2; c = 5;   % dx/dt = a - b*x,  x(0) = c
t_end = 4;
exact = @(t) a/b - (a/b - c) * exp(-b*t);

%% Forward Euler with different step sizes
steps = [0.05 0.25 0.9 1.1];
max_error = zeros(size(steps));
for k = 1:numel(steps)
    [t, x] = forward_euler(a, b, c, t_end, steps(k));
    max_error(k) = max(abs(x - exact(t)));
    fprintf('forward Euler, dt = %4g: max error = %.4g\n', ...
        steps(k), max_error(k));
end
fprintf('Forward Euler is stable only for dt < 2/b = %g\n', 2/b);

%% Forward vs backward Euler at a step size beyond the stability limit
[t_f, x_f] = forward_euler(a, b, c, t_end, 1.1);
[t_b, x_b] = backward_euler(a, b, c, t_end, 1.1);

%% Plot
colors = [0.122 0.467 0.706; 0.173 0.627 0.173;
          1.000 0.498 0.055; 0.839 0.153 0.157];
t_fine = linspace(0, 4.4, 441);
fig = figure('Position', [100 100 1000 400]);

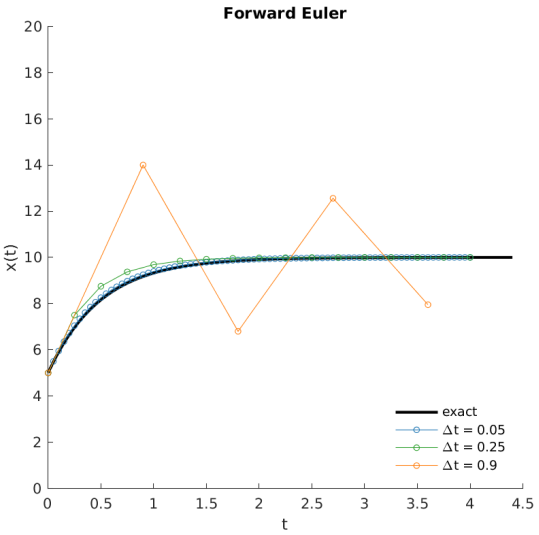
subplot(1, 2, 1)
plot(t_fine, exact(t_fine), 'k-', 'LineWidth', 2);
hold on
for k = 1:3                     % dt = 1.1 is in the right panel
    [t, x] = forward_euler(a, b, c, t_end, steps(k));
    plot(t, x, 'o-', 'Color', colors(k, :), 'MarkerSize', 4);
end
ylim([0 20]);
xlabel('t'); ylabel('x(t)'); title('Forward Euler');
legend(['exact', compose('\\Deltat = %g', steps(1:3))], ...
    'Location', 'southeast', 'Box', 'off');
box off
```

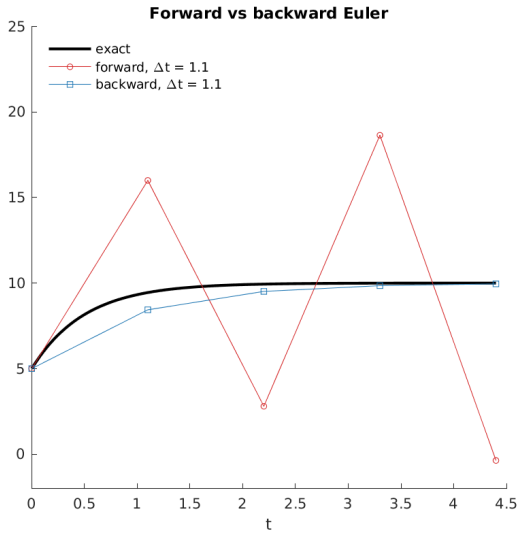
```
subplot(1, 2, 2)
plot(t_fine, exact(t_fine), 'k-', 'LineWidth', 2);
hold on
plot(t_f, x_f, 'o-', 'Color', colors(4, :), 'MarkerSize', 4);
plot(t_b, x_b, 's-', 'Color', colors(1, :), 'MarkerSize', 4);
ylim([-2 25]);
xlabel('t'); title('Forward vs backward Euler');
legend({'exact', 'forward, \Deltat = 1.1', 'backward, \Deltat = 1.1'}, ...
    'Location', 'northwest', 'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-euler');

results.steps = steps;
results.max_error_forward = max_error;
results.forward_dt_1_1 = x_f;
results.backward_dt_1_1 = x_b;
sbp.save_results(mfilename('fullpath'), results);

%% Local functions
function [t, x] = forward_euler(a, b, c, t_end, dt)
% x(t + dt) = x(t) + f(x(t)) * dt
n = round(t_end / dt);
t = dt * (0:n);
x = zeros(1, n + 1);
x(1) = c;
for i = 1:n
    x(i + 1) = x(i) + (a - b * x(i)) * dt;
end
end

function [t, x] = backward_euler(a, b, c, t_end, dt)
% x(t + dt) = x(t) + f(x(t + dt)) * dt, solved for x(t + dt).
% For a linear f this can be done by hand:
% x(t + dt) = (x(t) + a*dt) / (1 + b*dt).
n = round(t_end / dt);
t = dt * (0:n);
x = zeros(1, n + 1);
x(1) = c;
for i = 1:n
    x(i + 1) = (x(i) + a * dt) / (1 + b * dt);
end
end
```





Python

Needs NumPy and Matplotlib.

```
"""Solving dx/dt = a - b*x with the forward and backward Euler methods.

This simple ODE has an exact solution, so we can see how the step size
dt affects accuracy and, for forward Euler, stability.
"""
import numpy as np
import matplotlib.pyplot as plt
import sbp

a, b, c = 20, 2, 5      # dx/dt = a - b*x,  x(0) = c
t_end = 4


def exact(t):
    return a / b - (a / b - c) * np.exp(-b * t)


def forward_euler(dt):
    """x(t + dt) = x(t) + f(x(t)) * dt"""
    n = round(t_end / dt)
    t = dt * np.arange(n + 1)
    x = np.zeros(n + 1)
    x[0] = c
    for i in range(n):
        x[i + 1] = x[i] + (a - b * x[i]) * dt
    return t, x
```

```
def backward_euler(dt):
    """x(t + dt) = x(t) + f(x(t + dt)) * dt, solved for x(t + dt).

    For a linear f this can be done by hand:
    x(t + dt) = (x(t) + a*dt) / (1 + b*dt).
    """
    n = round(t_end / dt)
    t = dt * np.arange(n + 1)
    x = np.zeros(n + 1)
    x[0] = c
    for i in range(n):
        x[i + 1] = (x[i] + a * dt) / (1 + b * dt)
    return t, x


# --- Forward Euler with different step sizes
steps = [0.05, 0.25, 0.9, 1.1]
max_error = []
for dt in steps:
    t, x = forward_euler(dt)
    max_error.append(np.abs(x - exact(t)).max())
    print(f"forward Euler, dt = {dt:4}: max error = {max_error[-1]:.4g}")
print(f"Forward Euler is stable only for dt < 2/b = {2 / b}")

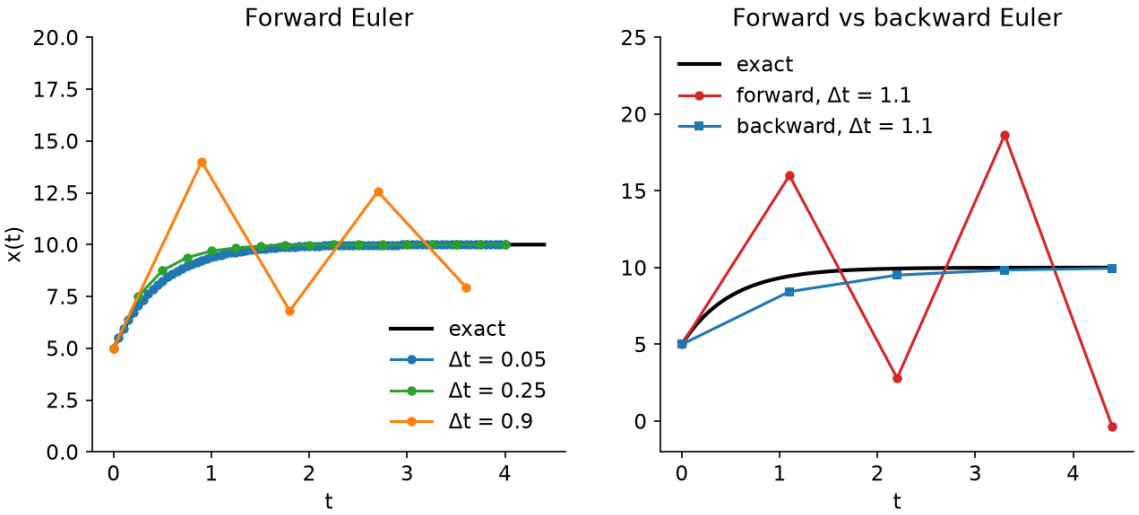
# --- Forward vs backward Euler at a step size beyond the stability limit
t_f, x_f = forward_euler(1.1)
t_b, x_b = backward_euler(1.1)

# --- Plot
sbp.style()
colors = ["#1f77b4", "#2ca02c", "#ff7f0e"]
t_fine = np.linspace(0, 4.4, 441)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
ax1.plot(t_fine, exact(t_fine), "k-", lw=2, label="exact")
for dt, col in zip(steps[:3], colors):  # dt = 1.1 is in the right panel
    t, x = forward_euler(dt)
    ax1.plot(t, x, "o-", color=col, ms=4, label=f"Δt = {dt}")
ax1.set_ylim(0, 20)
ax1.set_xlabel("t")
ax1.set_ylabel("x(t)")
ax1.set_title("Forward Euler")
ax1.legend(frameon=False, loc="lower right")

ax2.plot(t_fine, exact(t_fine), "k-", lw=2, label="exact")
ax2.plot(t_f, x_f, "o-", color="#d62728", ms=4, label="forward, Δt = 1.1")
```

```
ax2.plot(t_b, x_b, "s-", color="#1f77b4", ms=4, label="backward, Δt = 1.1")
ax2.set_ylim(-2, 25)
ax2.set_xlabel("t")
ax2.set_title("Forward vs backward Euler")
ax2.legend(frameon=False, loc="upper left")
sbp.save_figure(fig, __file__, "fig-euler")

sbp.save_results(__file__, {
    "steps": steps,
    "max_error_forward": max_error,
    "forward_dt_1_1": x_f,
    "backward_dt_1_1": x_b,
})
sbp.show()
```



Small steps track the exact solution well, and the error shrinks roughly in proportion to Δt. Large steps are a different matter. For this ODE, each forward Euler step multiplies the distance from the steady state by $(1 - b\,\Delta t)$. If Δt is between $1/b$ and $2/b$, this factor is negative but smaller than 1 in size, so the solution overshoots and oscillates, but still settles ($\Delta t = 0.9$). Beyond $2/b = 1$ it is larger than 1 in size, and the solution oscillates with growing amplitude ($\Delta t = 1.1$): the method is **unstable**. Backward Euler multiplies the distance by $1/(1 + b\,\Delta t)$ instead, which is less than 1 for any step size, so it stays stable even at $\Delta t = 1.1$; it is merely inaccurate.

3.1.1 In practice: use a library solver

You will rarely write your own solver. MATLAB and SciPy provide well-tested solvers that choose the step size automatically to keep the estimated error within a tolerance you set. The main ones correspond closely:

Table 3.1: Common ODE solvers in MATLAB and SciPy. Both use the same default tolerances (relative 10^{-3}, absolute 10^{-6}), which are often too loose for careful work.

	MATLAB	Python (SciPy)
Call	`[t, y] = ode45(f, tspan, y0, opts)`	`sol = solve_ivp(f, t_span, y0, ...)`
Right-hand side	`f(t, y)`, returns a column vector	`f(t, y)`, returns a list or array
General-purpose (non-stiff)	`ode45`	`method="RK45"` (the default)
Lower accuracy, cheaper	`ode23`	`method="RK23"`
Stiff problems	`ode15s`, `ode23s`	`method="BDF"`, `"Radau"`, or `"LSODA"` (switches automatically)
Tolerances	`odeset('RelTol', 1e-6, 'AbsTol', 1e-9)`	`rtol=1e-6, atol=1e-9`
Output at chosen times	pass a vector as `tspan`	pass `t_eval=...`
Shape of the solution	`y` is time × species	`sol.y` is species × time

3.2 Enzyme kinetics: mass action vs Michaelis–Menten

For an enzyme E converting a substrate S into a product P through a complex ES,

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \xrightarrow{k_2} E + P,$$

mass-action kinetics gives one ODE per species, with $[E] = E_0 - [ES]$ because the total amount of enzyme is fixed (book, Eqs. 5.4 and 5.6). Assuming in addition that $[ES]$ changes slowly compared to $[S]$ (the quasi-steady-state approximation of Briggs and Haldane) collapses all of this into the familiar Michaelis–Menten rate law

$$v = \frac{v_{\max}[S]}{K_M + [S]}, \qquad v_{\max} = k_2 E_0, \qquad K_M = \frac{k_{-1} + k_2}{k_1}.$$

The rate law is so widely used that it is easy to forget it is an approximation. The code below solves both models, for a small and a large amount of enzyme, with $K_M = 2$ mM and $S_0 = 10$ mM.

MATLAB

Core MATLAB only.

```
%% Full mass-action enzyme model vs the Michaelis-Menten rate law
%     E + S <-> ES -> E + P      (rate constants k1, k_1, k2)
% The full model tracks S, ES and P; the enzyme is E = E0 - ES. The
% Michaelis-Menten (Briggs-Haldane) approximation replaces all of this
% with a single rate law, v = vmax*S/(KM + S). How good is it?
% Requires: core MATLAB only.

k1 = 1.0; k_1 = 1.0; k2 = 1.0;     % 1/(mM s), 1/s, 1/s
KM = (k_1 + k2) / k1;              % Michaelis constant: 2 mM
S0 = 10.0;                         % initial substrate (mM)
opts = odeset('RelTol', 1e-8, 'AbsTol', 1e-10);  % tight tolerances

% Two cases: little enzyme (E0 << S0) and a lot of enzyme (E0 ~ S0)
names = {'low_enzyme', 'high_enzyme'};
E0s = [0.1 5.0];
t_ends = [150 6];

fig = figure('Position', [100 100 1000 400]);
for k = 1:2
    E0 = E0s(k);
    t = linspace(0, t_ends(k), 51);

    % Full model: y = [S; ES; P]
    full_model = @(t, y) [-k1*(E0 - y(2))*y(1) + k_1*y(2)
                          k1*(E0 - y(2))*y(1) - (k_1 + k2)*y(2)
                          k2*y(2)];
    [~, y] = ode45(full_model, t, [S0; 0; 0], opts);
    P_full = y(:, 3)';

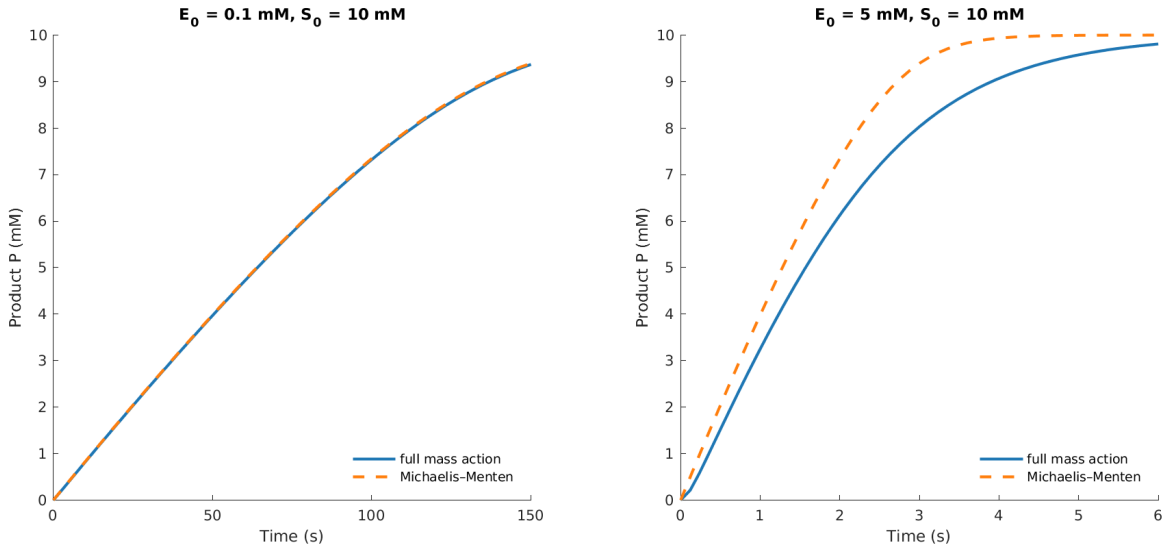
    % Michaelis-Menten model: a single ODE for S
    vmax = k2 * E0;
    mm_model = @(t, S) -vmax * S / (KM + S);
    [~, S] = ode45(mm_model, t, S0, opts);
    P_mm = S0 - S';                % in the MM model, P = S0 - S

    err = max(abs(P_full - P_mm)) / S0;
    fprintf('E0 = %g mM: max |P_full - P_MM| = %.1f%% of S0\n', E0,
     ↪  100*err);
    results.(names{k}) = struct('E0', E0, 't', t, 'P_full', P_full, ...
        'P_mm', P_mm, 'max_rel_difference', err);

    subplot(1, 2, k)
    plot(t, P_full, '-', 'Color', [0.122 0.467 0.706], 'LineWidth', 2);
    hold on
    plot(t, P_mm, '--', 'Color', [1.000 0.498 0.055], 'LineWidth', 2);
    xlabel('Time (s)'); ylabel('Product P (mM)');
```

```
    title(sprintf('E_0 = %g mM, S_0 = %g mM', E0, S0));
    legend({'full mass action', 'Michaelis-Menten'}, ...
        'Location', 'southeast', 'Box', 'off');
    box off
end
sbp.save_figure(fig, mfilename('fullpath'), 'fig-michaelis-menten');

sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs SciPy.

```
"""Full mass-action enzyme model vs the Michaelis-Menten rate law.

    E + S <-> ES -> E + P      (rate constants k1, k_1, k2)

The full model tracks S, ES and P; the enzyme is E = E0 - ES. The
Michaelis-Menten (Briggs-Haldane) approximation replaces all of this
with a single rate law, v = vmax*S/(KM + S). How good is it?
"""
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import sbp

k1, k_1, k2 = 1.0, 1.0, 1.0       # 1/(mM s), 1/s, 1/s
KM = (k_1 + k2) / k1               # Michaelis constant: 2 mM
S0 = 10.0                          # initial substrate (mM)
tol = dict(rtol=1e-8, atol=1e-10)  # tight tolerances for both solvers
```

```
def full_model(t, y, E0):
    """Mass-action ODEs for S, ES and P (book Eqs. 5.4)."""
    S, ES, P = y
    E = E0 - ES                    # enzyme conservation (Eq. 5.6)
    return [-k1 * E * S + k_1 * ES,
            k1 * E * S - (k_1 + k2) * ES,
            k2 * ES]

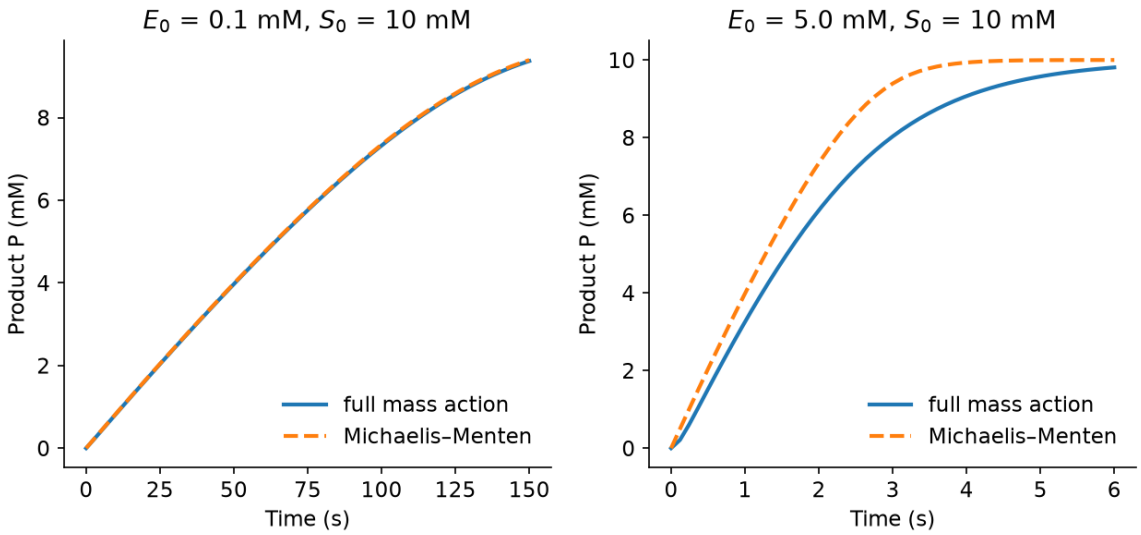

def mm_model(t, y, E0):
    """A single ODE for S, with the Michaelis-Menten rate law."""
    S = y[0]
    vmax = k2 * E0
    return [-vmax * S / (KM + S)]


def simulate(E0, t_end):
    t = np.linspace(0, t_end, 51)
    full = solve_ivp(full_model, [0, t_end], [S0, 0, 0], t_eval=t,
                     args=(E0,), **tol)
    mm = solve_ivp(mm_model, [0, t_end], [S0], t_eval=t, args=(E0,), **tol)
    P_full = full.y[2]
    P_mm = S0 - mm.y[0]            # in the MM model, P = S0 - S
    return t, P_full, P_mm


# Two cases: little enzyme (E0 << S0) and a lot of enzyme (E0 ~ S0)
cases = {"low_enzyme": (0.1, 150), "high_enzyme": (5.0, 6)}  # E0, t_end
results = {}
sbp.style()
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for ax, (name, (E0, t_end)) in zip(axes, cases.items()):
    t, P_full, P_mm = simulate(E0, t_end)
    err = np.abs(P_full - P_mm).max() / S0
    print(f"E0 = {E0} mM: max |P_full - P_MM| = {100 * err:.1f}% of S0")
    results[name] = {"E0": E0, "t": t, "P_full": P_full, "P_mm": P_mm,
                     "max_rel_difference": err}
    ax.plot(t, P_full, "-", color="#1f77b4", lw=2, label="full mass
↪   action")
    ax.plot(t, P_mm, "--", color="#ff7f0e", lw=2, label="Michaelis-Menten")
    ax.set_xlabel("Time (s)")
    ax.set_ylabel("Product P (mM)")
    ax.set_title(f"$E_0$ = {E0} mM, $S_0$ = {S0:g} mM")
    ax.legend(frameon=False, loc="lower right")
```

```
sbp.save_figure(fig, __file__, "fig-michaelis-menten")

sbp.save_results(__file__, results)
sbp.show()
```



With little enzyme, the two models are indistinguishable: they never differ by more than 0.3% of S_0. With a lot of enzyme, the Michaelis–Menten model runs ahead of the full model by up to 14% of S_0. The reason is that a substantial share of the substrate is held in the complex ES, which the rate law ignores. A useful rule of thumb is that the approximation is good when $E_0/(S_0 + K_M) \ll 1$ (Segel and Slemrod 1989); here that ratio is 0.008 in the first case and 0.42 in the second. Know thy assumptions!

3.3 Stiff equations

The quasi-steady-state approximation works because binding is fast and catalysis is slow. That same separation of time scales causes trouble for numerical solvers. Below, the enzyme model is solved again with $E_0 \ll S_0$, while binding and unbinding ($k_1 = k_{-1}$) are made faster and faster.

MATLAB

Core MATLAB only.

```
%% Stiffness: when fast binding makes a simple model expensive to solve
% The same enzyme model as before, but now binding and unbinding (k1, k_1)
% are much faster than catalysis (k2). The fast and slow time scales make
% the ODEs 'stiff'. We compare a non-stiff solver (ode45) with a stiff
% one (ode15s) by counting how often each one evaluates the right-hand
```

```
% side of the ODEs.
% Requires: core MATLAB only.

k2 = 1.0; E0 = 0.1; S0 = 10.0; t_end = 50.0;
opts = odeset('RelTol', 1e-6, 'AbsTol', 1e-9);

% y = [S; ES; P]
enzyme = @(t, y, k1, k_1) [-k1*(E0 - y(2))*y(1) + k_1*y(2)
                           k1*(E0 - y(2))*y(1) - (k_1 + k2)*y(2)
                           k2*y(2)];

%% How the cost grows as binding gets faster (k1 = k_1 = k)
speeds = [1 10 100 1000];
cost_nonstiff = zeros(size(speeds));
cost_stiff = zeros(size(speeds));
final_P = zeros(size(speeds));
for i = 1:numel(speeds)
    k = speeds(i);
    f = @(t, y) enzyme(t, y, k, k);
    nonstiff = ode45(f, [0 t_end], [S0; 0; 0], opts);
    stiff = ode15s(f, [0 t_end], [S0; 0; 0], opts);
    cost_nonstiff(i) = nonstiff.stats.nfevals;
    cost_stiff(i) = stiff.stats.nfevals;
    final_P(i) = stiff.y(3, end);
    fprintf(['k = %5g: ode45 used %7d evaluations, ode15s %5d;  ' ...
        'P(%g) = %.4f mM\n'], k, cost_nonstiff(i), cost_stiff(i), ...
        t_end, final_P(i));
end

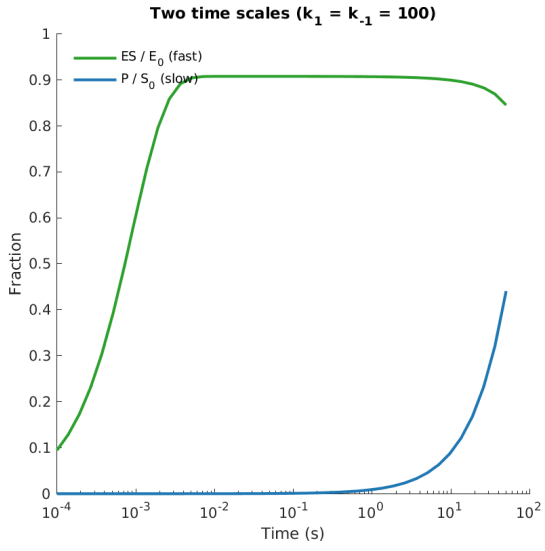
%% The fast transient, for k = 100, on a logarithmic time axis
t = logspace(-4, log10(t_end), 41);
[~, y] = ode15s(@(t, y) enzyme(t, y, 100, 100), [0 t], [S0; 0; 0], opts);
y = y(2:end, :);                % drop t = 0, which we added as the start

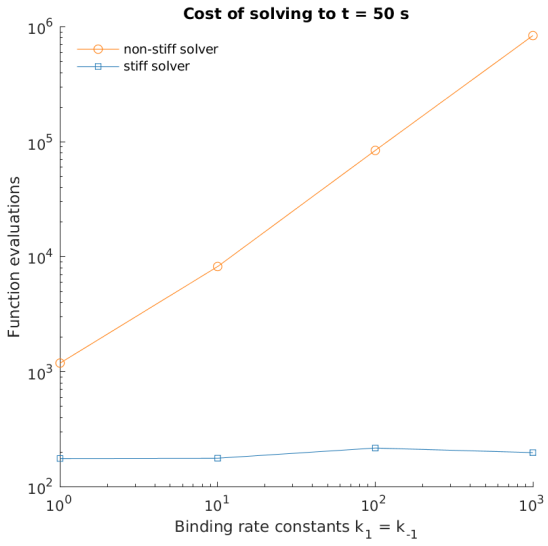
%% Plot
fig = figure('Position', [100 100 1000 400]);
subplot(1, 2, 1)
semilogx(t, y(:, 2) / E0, '-', 'Color', [0.173 0.627 0.173], 'LineWidth',
 ↪  2);
hold on
semilogx(t, y(:, 3) / S0, '-', 'Color', [0.122 0.467 0.706], 'LineWidth',
 ↪  2);
xlabel('Time (s)'); ylabel('Fraction');
title('Two time scales (k_1 = k_{-1} = 100)');
legend({'ES / E_0 (fast)', 'P / S_0 (slow)'}, 'Location', 'northwest', ...
    'Box', 'off');
```

```
box off

subplot(1, 2, 2)
loglog(speeds, cost_nonstiff, 'o-', 'Color', [1.000 0.498 0.055]);
hold on
loglog(speeds, cost_stiff, 's-', 'Color', [0.122 0.467 0.706]);
xlabel('Binding rate constants k_1 = k_{-1}');
ylabel('Function evaluations');
title('Cost of solving to t = 50 s');
legend({'non-stiff solver', 'stiff solver'}, 'Location', 'northwest', ...
    'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-stiff');

results.speeds = speeds;
results.final_P = final_P;
results.t = t;
results.ES_k100 = y(:, 2)';
results.P_k100 = y(:, 3)';
results.evaluations_nonstiff = cost_nonstiff;
results.evaluations_stiff = cost_stiff;
sbp.save_results(mfilename('fullpath'), results);
```





Python

Needs SciPy.

```
"""Stiffness: when fast binding makes a simple model expensive to solve.

The same enzyme model as before, but now binding and unbinding (k1, k_1)
```

```
are much faster than catalysis (k2). The fast and slow time scales make
the ODEs 'stiff'. We compare a non-stiff solver (RK45, like ode45) with a
stiff one (BDF, like ode15s) by counting how often each one evaluates the
right-hand side of the ODEs.
"""
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import sbp

k2, E0, S0, t_end = 1.0, 0.1, 10.0, 50.0
tol = dict(rtol=1e-6, atol=1e-9)


def enzyme(t, y, k1, k_1):
    S, ES, P = y
    E = E0 - ES
    return [-k1 * E * S + k_1 * ES,
            k1 * E * S - (k_1 + k2) * ES,
            k2 * ES]


# --- How the cost grows as binding gets faster (k1 = k_1 = k)
speeds = [1, 10, 100, 1000]
cost = {"nonstiff": [], "stiff": []}
final_P = []
for k in speeds:
    nonstiff = solve_ivp(enzyme, [0, t_end], [S0, 0, 0], method="RK45",
                         args=(k, k), **tol)
    stiff = solve_ivp(enzyme, [0, t_end], [S0, 0, 0], method="BDF",
                      args=(k, k), **tol)
    cost["nonstiff"].append(nonstiff.nfev)
    cost["stiff"].append(stiff.nfev)
    final_P.append(stiff.y[2, -1])
    print(f"k = {k:5}: RK45 used {nonstiff.nfev:7} evaluations, "
          f"BDF {stiff.nfev:5};  P({t_end:g}) = {stiff.y[2, -1]:.4f} mM")

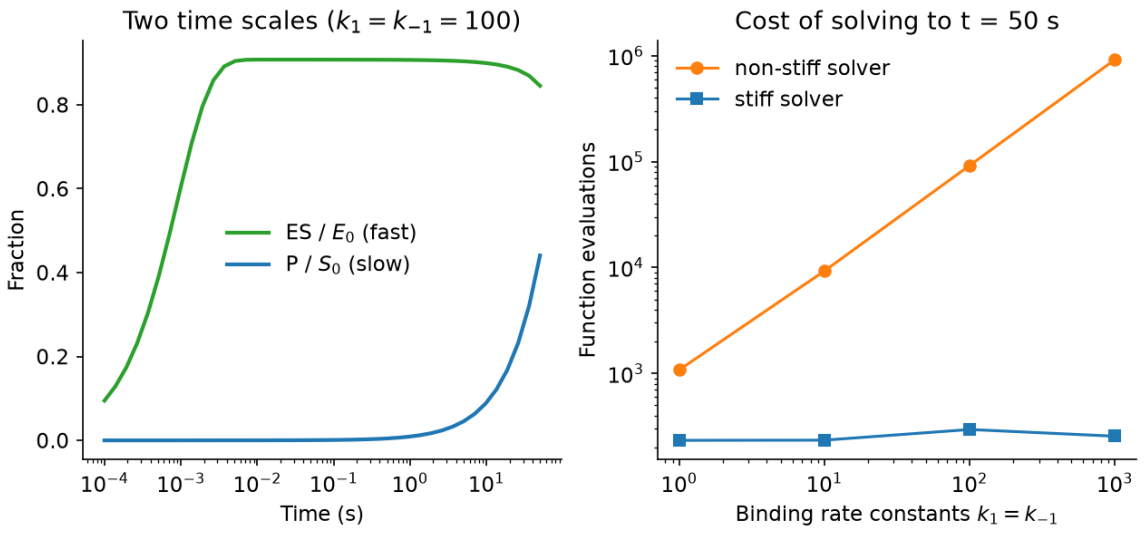
# --- The fast transient, for k = 100, on a logarithmic time axis
t = np.logspace(-4, np.log10(t_end), 41)
sol = solve_ivp(enzyme, [0, t_end], [S0, 0, 0], method="BDF", t_eval=t,
                args=(100, 100), **tol)

# --- Plot
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
ax1.semilogx(t, sol.y[1] / E0, "-", color="#2ca02c", lw=2,
```

```
            label="ES / $E_0$ (fast)")
ax1.semilogx(t, sol.y[2] / S0, "-", color="#1f77b4", lw=2,
            label="P / $S_0$ (slow)")
ax1.set_xlabel("Time (s)")
ax1.set_ylabel("Fraction")
ax1.set_title("Two time scales ($k_1 = k_{-1} = 100$)")
ax1.legend(frameon=False)

ax2.loglog(speeds, cost["nonstiff"], "o-", color="#ff7f0e",
           label="non-stiff solver")
ax2.loglog(speeds, cost["stiff"], "s-", color="#1f77b4",
           label="stiff solver")
ax2.set_xlabel("Binding rate constants $k_1 = k_{-1}$")
ax2.set_ylabel("Function evaluations")
ax2.set_title("Cost of solving to t = 50 s")
ax2.legend(frameon=False)
sbp.save_figure(fig, __file__, "fig-stiff")

sbp.save_results(__file__, {
    "speeds": speeds,
    "final_P": final_P,
    "t": t,
    "ES_k100": sol.y[1],
    "P_k100": sol.y[2],
    "evaluations_nonstiff": cost["nonstiff"],
    "evaluations_stiff": cost["stiff"],
})
sbp.show()
```



The left panel shows the two time scales: the complex ES forms within a few milliseconds,

while the product takes tens of seconds to accumulate. After the first few milliseconds, nothing fast is happening, yet a non-stiff solver such as `ode45` or `RK45` must keep its steps small enough to remain stable with respect to the fast process, just as forward Euler had to keep $\Delta t < 2/b$. Its cost therefore grows in proportion to the binding rate constants (right panel), reaching close to a million function evaluations. A stiff solver such as `ode15s` or `BDF` is implicit, like backward Euler, and can take large steps once the fast transient is over; its cost hardly changes. Both reach the same answer.

Such ODEs are called **stiff**: stability, rather than accuracy, limits the step size. Stiffness is common in biology, where binding, phosphorylation and transport often run much faster than gene expression or growth. A practical rule: start with `ode45` or `RK45`, and if it is unexpectedly slow, switch to `ode15s` or `BDF` (or `LSODA`).

The exact counts differ between MATLAB and Python, because the solvers differ in their internals, but the pattern is the same; our automated checks compare only the solutions.

3.4 Exercises

1. Run forward Euler with $\Delta t = 1/b = 0.5$ and $\Delta t = 2/b = 1$. Explain both results using the factor $(1 - b\,\Delta t)$.
2. For the Michaelis–Menten example, vary E_0 from 0.01 to 10 mM and plot the largest difference between the two models against $E_0/(S_0 + K_M)$.
3. Make the product step reversible ($E + P \to ES$ with rate constant k_{-2}; book Eq. 5.3). Does the Michaelis–Menten rate law still describe the time course? What happens at long times?
4. Solve the stiff example with $k_1 = k_{-1} = 1000$ using `ode45` and `ode15s` (or `RK45` and `BDF`), timing each with `tic`/`toc` (or `time.perf_counter`).
5. Replace the Michaelis–Menten rate law with a Hill rate law, $v = v_{\max}[S]^n/(K^n + [S]^n)$, and compare the time courses for $n = 1, 2, 4$.

3.5 Going further

The parameters here were simply chosen. In practice, they must be estimated from data, which is the subject of the next chapter (book, Chapter 6).

4 Parameter estimation

Every dynamic model carries parameters, such as rate constants, Michaelis constants and inhibition constants, and only a few of them can be looked up. The rest must be estimated from experimental data: we look for the parameter values that make the model's predictions agree best with the measurements. This chapter sets up that problem for a small enzyme-kinetic model, solves it in three different ways, and shows what the data can and cannot tell us about the parameters.

> **In the book**
>
> Chapter 6, *Parameter Estimation*: least squares and its weighted forms (§6.2.2), maximum likelihood (§6.2.3), algorithms for optimisation, from direct search to evolutionary algorithms (§6.3), post-regression diagnostics (§6.4) and troubleshooting, including identifiability and the curse of dimensionality (§6.5). Exercise 6.1 uses the same model.

4.1 The model and the data

An enzyme follows Michaelis–Menten kinetics with a competitive inhibitor I:

$$v = \frac{v_{\max} S}{K_M \left(1 + \dfrac{I}{K_I}\right) + S}.$$

We want $v_{\max}$, K_M and K_I. The initial rate v was measured at 20 substrate concentrations, from 1 to 20 mM, once with the inhibitor at $I = 10$ mM (dataset D1) and once at $I = 20$ mM (dataset D2). The data are in `data/competitive_inhibition.csv`.

Each measurement is assumed to carry an error of about 10% of its value, $\delta = 0.1$. A good fit should therefore be judged by *relative* errors, and the cost we minimise is the weighted sum of squared errors (book, Eq. 6.5):

$$\mathrm{SSE}(\Theta) = \sum_{i=1}^{n} \left(\frac{v_{\mathrm{pred},i}(\Theta) - v_i}{\delta\, v_i} \right)^2, \qquad \Theta = (v_{\max}, K_M, K_I).$$

If the measurement errors are independent and normally distributed with standard deviation δv_i, minimising this SSE is the same as finding the maximum-likelihood estimate (book, §6.2.3).

4.2 Fitting by least squares

Both MATLAB and SciPy have solvers made for least-squares problems. They take a function that returns the vector of weighted *residuals*, one per data point, and minimise the sum of their squares. They use the structure of the problem to converge quickly, but, like all local methods, they need a starting guess.

Table 4.1: Optimisers used for parameter estimation.

	MATLAB	Python (SciPy)
Least squares (residual vector)	`lsqnonlin` (Optimization Toolbox)	`scipy.optimize.least_squares`
General minimisation, no derivatives	`fminsearch` (core MATLAB)	`scipy.optimize.minimize(method="Nelder`
Global search	`ga`, `particleswarm` (Global Optimization Toolbox)	`scipy.optimize.differential_evolution`

We fit each dataset, D1, D2 and both together, from the same eight starting guesses.

MATLAB

Needs the Optimization Toolbox (for `lsqnonlin`).

```
%% Fitting a rate law to data by weighted least squares
% Competitive inhibition:  v = vmax*S / (KM*(1 + I/KI) + S).
% Rates were measured at 20 substrate concentrations, with the inhibitor
% at I = 10 mM (dataset D1) and at I = 20 mM (dataset D2). We estimate
% vmax, KM and KI from D1 alone, and from both datasets together.
% Requires: Optimization Toolbox (for lsqnonlin).

data = readmatrix(fullfile(sbp.data_dir(), 'competitive_inhibition.csv'));
S = data(:, 1); v10 = data(:, 2); v20 = data(:, 3);
delta = 0.1;                        % relative measurement error (10%)

rate = @(p, S, I) p(1) * S ./ (p(2) * (1 + I / p(3)) + S);
% Weighted residuals; their sum of squares is the SSE (book Eq. 6.5)
residuals = @(p, S, I, v) (rate(p, S, I) - v) ./ (delta * v);
opts = optimoptions('lsqnonlin', 'Display', 'off', ...
    'StepTolerance', 1e-12, 'FunctionTolerance', 1e-12, ...
    'OptimalityTolerance', 1e-12);

% The same eight starting guesses for every fit: vmax = 20 and a spread
% of values for KM and KI
KM_KI = [1 1; 2 8; 5 2; 5 20; 10 5; 20 40; 40 10; 40 80];
starts = [20 * ones(8, 1), KM_KI];
```

```
names = {'D1', 'D2', 'D1_D2'};
datasets = {{S, 10 * ones(20, 1), v10}, ...
            {S, 20 * ones(20, 1), v20}, ...
            {[S; S], [10 * ones(20, 1); 20 * ones(20, 1)], [v10; v20]}};

for d = 1:3
    [s, i, v] = datasets{d}{:};
    P = zeros(8, 3);
    SSE = zeros(8, 1);
    for k = 1:8
        [P(k, :), SSE(k)] = lsqnonlin(@(p) residuals(p, s, i, v), ...
            starts(k, :), [0 0 0], [Inf Inf Inf], opts);
    end
    [~, best] = min(SSE);
    % KM*(1 + I/KI) at the first inhibitor level of the dataset
    KM_app = P(:, 2) .* (1 + i(1) ./ P(:, 3));
    fprintf('%s: SSE = %.4f (spread over starts %.1e)\n', names{d}, ...
        SSE(best), max(SSE) - min(SSE));
    fprintf(['   vmax = %.4f;  KM ranges %.3g to %.3g;  ' ...
        'KI ranges %.3g to %.3g\n'], P(best, 1), min(P(:, 2)), ...
        max(P(:, 2)), min(P(:, 3)), max(P(:, 3)));
    % What every start agrees on; KM and KI only if they are pinned down
    results.(names{d}) = struct('SSE', SSE(best), 'vmax', P(best, 1), ...
        'KM_app', KM_app(best));
    if strcmp(names{d}, 'D1_D2')
        results.D1_D2.KM = P(best, 2);
        results.D1_D2.KI = P(best, 3);
    else
        results.([names{d} '_estimates']) = struct('KM', P(:, 2)', ...
            'KI', P(:, 3)');
    end
end

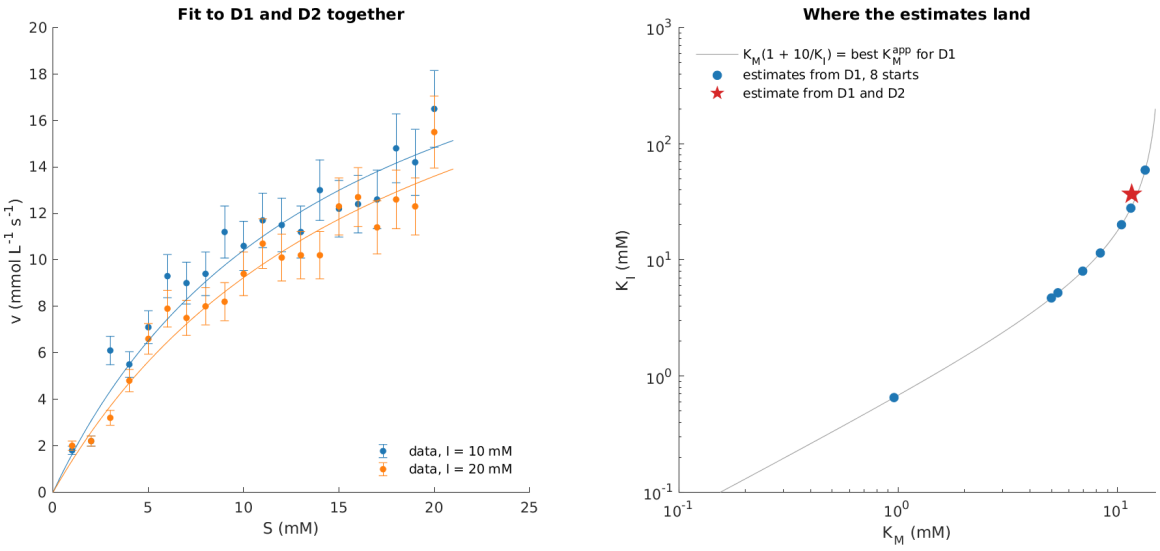
%% Plot
fig = figure('Position', [100 100 1000 400]);
blue = [0.122 0.467 0.706]; orange = [1.000 0.498 0.055];
r = results.D1_D2;
p = [r.vmax r.KM r.KI];
S_fine = linspace(0, 21, 200);

subplot(1, 2, 1)
hold on
errorbar(S, v10, delta * v10, 'o', 'Color', blue, 'MarkerSize', 4, ...
    'MarkerFaceColor', blue);
errorbar(S, v20, delta * v20, 'o', 'Color', orange, 'MarkerSize', 4, ...
```

```
    'MarkerFaceColor', orange);
plot(S_fine, rate(p, S_fine, 10), '-', 'Color', blue);
plot(S_fine, rate(p, S_fine, 20), '-', 'Color', orange);
xlabel('S (mM)'); ylabel('v (mmol L^{-1} s^{-1})');
title('Fit to D1 and D2 together');
legend({'data, I = 10 mM', 'data, I = 20 mM'}, 'Location', 'southeast', ...
    'Box', 'off');
box off

subplot(1, 2, 2)
KI_line = logspace(-1, 2.3, 200);
loglog(results.D1.KM_app ./ (1 + 10 ./ KI_line), KI_line, '-', ...
    'Color', [0.6 0.6 0.6]);
hold on
est = results.D1_estimates;
loglog(est.KM, est.KI, 'o', 'Color', blue, 'MarkerFaceColor', blue);
loglog(p(2), p(3), 'p', 'Color', [0.839 0.153 0.157], ...
    'MarkerFaceColor', [0.839 0.153 0.157], 'MarkerSize', 14);
xlabel('K_M (mM)'); ylabel('K_I (mM)');
title('Where the estimates land');
legend({'K_M(1 + 10/K_I) = best K_M^{app} for D1', ...
    'estimates from D1, 8 starts', 'estimate from D1 and D2'}, ...
    'Location', 'northwest', 'Box', 'off', 'FontSize', 9);
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-fit-inhibition');

sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs SciPy.

```
"""Fitting a rate law to data by weighted least squares.

Competitive inhibition:  v = vmax*S / (KM*(1 + I/KI) + S).
Rates were measured at 20 substrate concentrations, with the inhibitor
at I = 10 mM (dataset D1) and at I = 20 mM (dataset D2). We estimate
vmax, KM and KI from D1 alone, and from both datasets together.
"""
import numpy as np
from scipy.optimize import least_squares
import matplotlib.pyplot as plt
import sbp

data = np.loadtxt(sbp.DATA / "competitive_inhibition.csv", delimiter=",",
                  skiprows=1)
S, v10, v20 = data[:, 0], data[:, 1], data[:, 2]
delta = 0.1                          # relative measurement error (10%)


def rate(p, S, I):
    vmax, KM, KI = p
    return vmax * S / (KM * (1 + I / KI) + S)


def residuals(p, S, I, v):
    """Weighted residuals; their sum of squares is the SSE (book Eq.
     ↪  6.5)."""
    return (rate(p, S, I) - v) / (delta * v)


def fit(S, I, v, p0):
    sol = least_squares(residuals, p0, args=(S, I, v), bounds=(0, np.inf),
                        xtol=1e-12, ftol=1e-12, gtol=1e-12)
    return sol.x, np.sum(sol.fun ** 2)


# The same eight starting guesses for every fit: vmax = 20 and a spread
# of values for KM and KI
starts = [(20, km, ki) for km, ki in
          [(1, 1), (2, 8), (5, 2), (5, 20), (10, 5), (20, 40), (40, 10),
           (40, 80)]]

datasets = {
    "D1": (S, np.full_like(S, 10.0), v10),
    "D2": (S, np.full_like(S, 20.0), v20),
```

```
    "D1_D2": (np.r_[S, S], np.r_[np.full_like(S, 10.0),
                                 np.full_like(S, 20.0)], np.r_[v10, v20]),
}

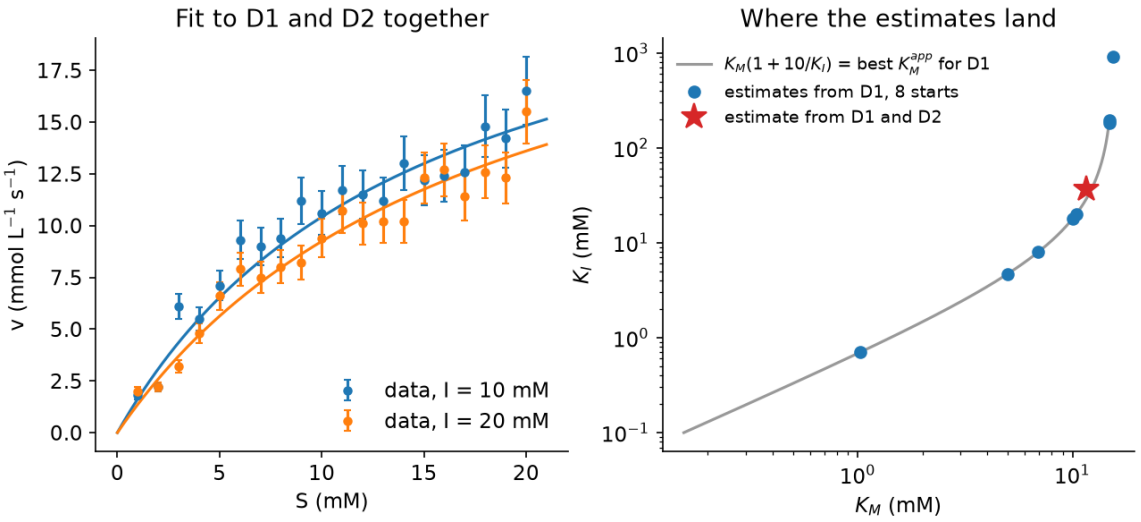
results = {}
for name, (s, i, v) in datasets.items():
    fits = [fit(s, i, v, p0) for p0 in starts]
    P = np.array([p for p, _ in fits])
    SSE = np.array([sse for _, sse in fits])
    best = np.argmin(SSE)
    # KM*(1 + I/KI) at the first inhibitor level of the dataset
    KM_app = P[:, 1] * (1 + i[0] / P[:, 2])
    print(f"{name}: SSE = {SSE[best]:.4f} (spread over starts "
          f"{np.ptp(SSE):.1e})")
    print(f"   vmax = {P[best, 0]:.4f};  KM ranges {P[:, 1].min():.3g} to "
          f"{P[:, 1].max():.3g};  KI ranges {P[:, 2].min():.3g} to "
          f"{P[:, 2].max():.3g}")
    # What every start agrees on; KM and KI only if they are pinned down
    results[name] = {"SSE": SSE[best], "vmax": P[best, 0],
                     "KM_app": KM_app[best]}
    if name == "D1_D2":
        results[name].update(KM=P[best, 1], KI=P[best, 2])
    else:
        results[name + "_estimates"] = {"KM": P[:, 1], "KI": P[:, 2]}

# --- Plot
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
p = [results["D1_D2"][k] for k in ("vmax", "KM", "KI")]
S_fine = np.linspace(0, 21, 200)
for I, v, col in [(10, v10, "#1f77b4"), (20, v20, "#ff7f0e")]:
    ax1.errorbar(S, v, yerr=delta * v, fmt="o", color=col, ms=4,
                 capsize=2, label=f"data, I = {I} mM")
    ax1.plot(S_fine, rate(p, S_fine, I), "-", color=col)
ax1.set_xlabel("S (mM)")
ax1.set_ylabel("v (mmol L$^{-1}$ s$^{-1}$)")
ax1.set_title("Fit to D1 and D2 together")
ax1.legend(frameon=False, loc="lower right")

KI_line = np.logspace(-1, 2.3, 200)
ax2.plot(results["D1"]["KM_app"] / (1 + 10 / KI_line), KI_line, "-",
         color="#999999", label="$K_M(1 + 10/K_I)$ = best $K_M^{app}$ for
 ↪  D1")
est = results["D1_estimates"]
ax2.plot(est["KM"], est["KI"], "o", color="#1f77b4", ms=6,
         label="estimates from D1, 8 starts")
```

```
ax2.plot(p[1], p[2], "*", color="#d62728", ms=14,
         label="estimate from D1 and D2")
ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_xlabel("$K_M$ (mM)")
ax2.set_ylabel("$K_I$ (mM)")
ax2.set_title("Where the estimates land")
ax2.legend(frameon=False, loc="upper left", fontsize=9)
sbp.save_figure(fig, __file__, "fig-fit-inhibition")

sbp.save_results(__file__, results)
sbp.show()
```



With both datasets, all eight starts arrive at the same answer, $v_{\max} = 25.8$ mmol L^{-1} s^{-1}, $K_M = 11.6$ mM and $K_I = 36.8$ mM, and the curves follow the data well (left panel).

With D1 alone, something quite different happens. All eight starts again reach exactly the same SSE and the same $v_{\max}$, and every one of them reports a successful fit. Yet K_M ranges from about 1 to 15 mM and K_I from below 1 to several hundred mM. The right panel shows why. At a single inhibitor concentration, K_M and K_I enter the rate law only through the combination

$$K_M^{\text{app}} = K_M \left(1 + \frac{I}{K_I}\right),$$

the apparent Michaelis constant. Every pair (K_M, K_I) on the grey curve gives the same K_M^{app}, and hence identical predictions for every data point. The data determine $v_{\max}$ and K_M^{app}, and nothing more: K_M and K_I are **structurally non-identifiable** from D1. More data of the same kind would not help. A second inhibitor concentration does, because K_M^{app} changes with I in a way that depends on K_I (red star).

Two lessons follow. First, a solver that reports success has found *a* minimum; it says nothing about whether the minimum is unique. Fitting from several starting points is a simple and effective check. Second, identifiability is a property of the model *and* the experiment together, and it is best examined before the experiments are done (book, §6.5.3).

4.3 The error landscape, and grid search

The simplest way to minimise a function is to evaluate it everywhere, on a grid, and keep the best point (book, §6.3.3). Grid search needs no starting guess and cannot be trapped by a local minimum, and it shows the whole error landscape, which makes it a good tool for seeing what is going on.

Here, one more trick keeps the grid small. The rate law is proportional to $v_{\max}$, so for given K_M and K_I the weighted residuals are linear in $v_{\max}$, and the best $v_{\max}$ follows from a one-line formula, just as in linear regression. The search is then over K_M and K_I only: whole numbers from 1 to 60 mM, 3600 points.

MATLAB

Core MATLAB only.

```
%% Grid search, and the shape of the error landscape
% The same competitive-inhibition model and data as in fit_inhibition.
% We evaluate the weighted SSE on a grid of whole-number values of KM and
% KI (1 to 60 mM). The rate is proportional to vmax, so for each (KM, KI)
% the best vmax can be computed exactly, as in linear regression, and the
% grid only needs two dimensions.
% Requires: core MATLAB only.

data = readmatrix(fullfile(sbp.data_dir(), 'competitive_inhibition.csv'));
S = data(:, 1); v10 = data(:, 2); v20 = data(:, 3);
delta = 0.1;
names = {'D1', 'D1_D2'};
datasets = {{S, 10 * ones(20, 1), v10}, ...
            {[S; S], [10 * ones(20, 1); 20 * ones(20, 1)], [v10; v20]}};
KM_grid = 1:60;
KI_grid = 1:60;

for d = 1:2
    [s, i, v] = datasets{d}{:};
    grid = zeros(numel(KI_grid), numel(KM_grid));  % rows: KI, columns: KM
    vmax = zeros(size(grid));
    for r = 1:numel(KI_grid)
        for c = 1:numel(KM_grid)
            [vmax(r, c), grid(r, c)] = best_vmax_and_sse( ...
                KM_grid(c), KI_grid(r), s, i, v, delta);
```

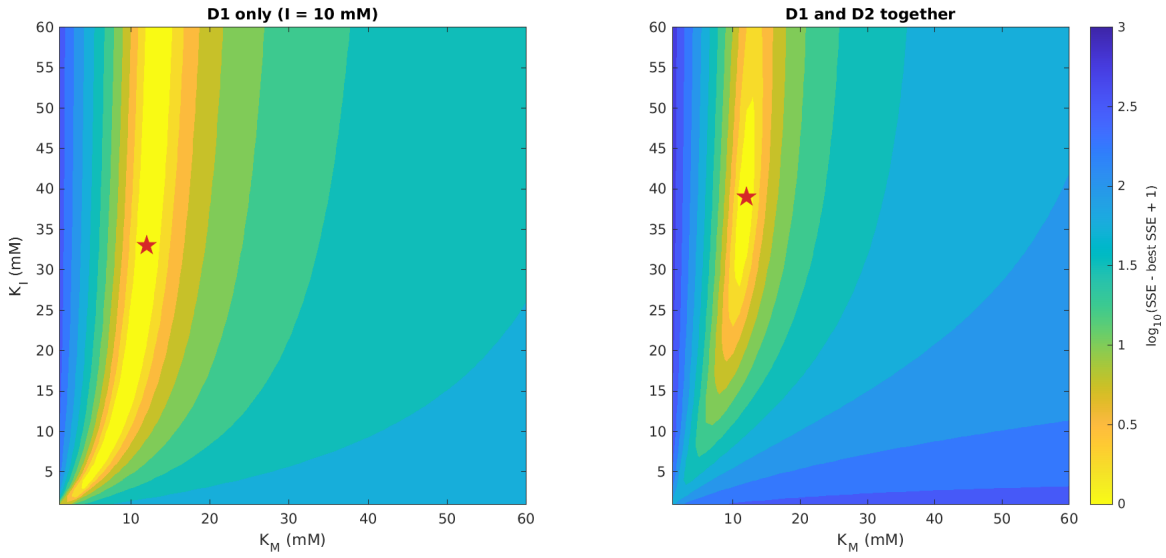
```
        end
    end
    [best, idx] = min(grid(:));
    [r, c] = ind2sub(size(grid), idx);
    % How many grid points fit almost as well as the best (within 1 unit)?
    n_close = sum(grid(:) < best + 1);
    fprintf(['%s: best on grid KM = %d, KI = %d, vmax = %.4f, ' ...
        'SSE = %.4f; %d grid points within 1 of the best SSE\n'], ...
        names{d}, KM_grid(c), KI_grid(r), vmax(r, c), best, n_close);
    SSE.(names{d}) = grid;
    results.(names{d}) = struct('KM', KM_grid(c), 'KI', KI_grid(r), ...
        'vmax', vmax(r, c), 'SSE', best, 'n_within_1', n_close, ...
        'SSE_at_KI_20', grid(KI_grid == 20, :));
end

%% Plot: log10(SSE - best SSE + 1), so that the best point is at 0
fig = figure('Position', [100 100 1000 420]);
titles = {'D1 only (I = 10 mM)', 'D1 and D2 together'};
for d = 1:2
    subplot(1, 2, d)
    Z = log10(SSE.(names{d}) - min(SSE.(names{d})(:)) + 1);
    contourf(KM_grid, KI_grid, min(Z, 3), linspace(0, 3, 13), ...
        'LineStyle', 'none');
    hold on
    plot(results.(names{d}).KM, results.(names{d}).KI, 'p', ...
        'Color', [0.839 0.153 0.157], ...
        'MarkerFaceColor', [0.839 0.153 0.157], 'MarkerSize', 14);
    colormap(flipud(parula));
    clim([0 3]);
    xlabel('K_M (mM)');
    if d == 1, ylabel('K_I (mM)'); end
    title(titles{d});
end
cb = colorbar;
cb.Label.String = 'log_{10}(SSE - best SSE + 1)';
sbp.save_figure(fig, mfilename('fullpath'), 'fig-sse-landscape');

sbp.save_results(mfilename('fullpath'), results);

%% Local function
function [vmax, sse] = best_vmax_and_sse(KM, KI, S, I, v, delta)
% Weighted residuals are vmax*a - b, so the best vmax is
 ↪  sum(a.*b)/sum(a.^2).
a = S ./ (KM * (1 + I / KI) + S) ./ (delta * v);
b = 1 / delta;
vmax = sum(a .* b) / sum(a .* a);
```

```
sse = sum((vmax * a - b) .^ 2);
end
```



Python

Needs NumPy and Matplotlib.

```
"""Grid search, and the shape of the error landscape.

The same competitive-inhibition model and data as in fit_inhibition.
We evaluate the weighted SSE on a grid of whole-number values of KM and
KI (1 to 60 mM). The rate is proportional to vmax, so for each (KM, KI)
the best vmax can be computed exactly, as in linear regression, and the
grid only needs two dimensions.
"""
import numpy as np
import matplotlib.pyplot as plt
import sbp

data = np.loadtxt(sbp.DATA / "competitive_inhibition.csv", delimiter=",",
                  skiprows=1)
S, v10, v20 = data[:, 0], data[:, 1], data[:, 2]
delta = 0.1
datasets = {
    "D1": (S, np.full_like(S, 10.0), v10),
    "D1_D2": (np.r_[S, S], np.r_[np.full_like(S, 10.0),
                                 np.full_like(S, 20.0)], np.r_[v10, v20]),
}
KM_grid = np.arange(1, 61)
```

```
KI_grid = np.arange(1, 61)

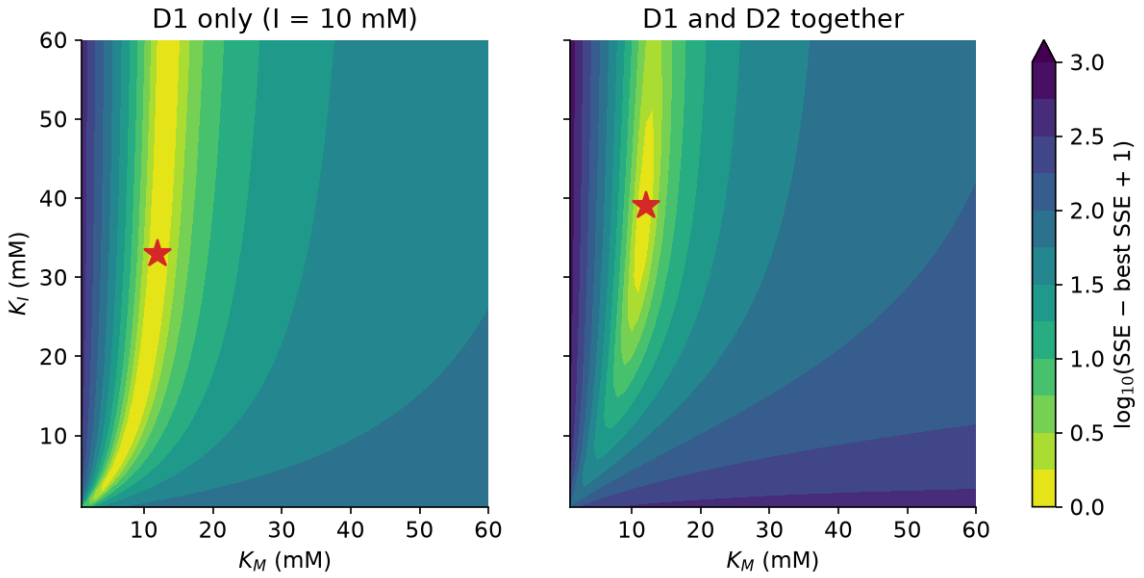

def best_vmax_and_sse(KM, KI, S, I, v):
    """Weighted residuals are vmax*a - b, so the best vmax is
    sum(a*b) / sum(a^2)."""
    a = S / (KM * (1 + I / KI) + S) / (delta * v)
    b = 1 / delta
    vmax = np.sum(a * b) / np.sum(a * a)
    return vmax, np.sum((vmax * a - b) ** 2)


results = {}
SSE = {}
for name, (s, i, v) in datasets.items():
    grid = np.zeros((len(KI_grid), len(KM_grid)))   # rows: KI, columns: KM
    vmax = np.zeros_like(grid)
    for r, KI in enumerate(KI_grid):
        for c, KM in enumerate(KM_grid):
            vmax[r, c], grid[r, c] = best_vmax_and_sse(KM, KI, s, i, v)
    r, c = np.unravel_index(np.argmin(grid), grid.shape)
    # How many grid points fit almost as well as the best (within 1 unit)?
    n_close = int(np.sum(grid < grid[r, c] + 1))
    print(f"{name}: best on grid KM = {KM_grid[c]}, KI = {KI_grid[r]}, "
          f"vmax = {vmax[r, c]:.4f}, SSE = {grid[r, c]:.4f}; "
          f"{n_close} grid points within 1 of the best SSE")
    SSE[name] = grid
    results[name] = {"KM": KM_grid[c], "KI": KI_grid[r],
                     "vmax": vmax[r, c], "SSE": grid[r, c],
                     "n_within_1": n_close,
                     "SSE_at_KI_20": grid[KI_grid == 20, :].ravel()}

# --- Plot: log10(SSE - best SSE + 1), so that the best point is at 0
sbp.style()
fig, axes = plt.subplots(1, 2, figsize=(10, 4.2), sharey=True)
levels = np.linspace(0, 3, 13)
titles = {"D1": "D1 only (I = 10 mM)", "D1_D2": "D1 and D2 together"}
for ax, name in zip(axes, SSE):
    Z = np.log10(SSE[name] - SSE[name].min() + 1)
    cs = ax.contourf(KM_grid, KI_grid, Z, levels=levels, cmap="viridis_r",
                     extend="max")
    ax.plot(results[name]["KM"], results[name]["KI"], "*", color="#d62728",
            ms=14)
    ax.set_xlabel("$K_M$ (mM)")
    ax.set_title(titles[name])
axes[0].set_ylabel("$K_I$ (mM)")
```

```
fig.colorbar(cs, ax=axes, label="log$_{10}$(SSE - best SSE + 1)")
sbp.save_figure(fig, __file__, "fig-sse-landscape")

sbp.save_results(__file__, results)
sbp.show()
```



For D1, the landscape has a long, curved valley with an almost flat floor: the curve of constant K_M^{app} from the previous section. The grid point that happens to be lowest ($K_M = 12$, $K_I = 33$) is no better than many others along the valley; 185 grid points come within one unit of the best SSE, with K_I anywhere from 2 to 60 mM. For D1 and D2 together, the valley closes into a basin with a single minimum, at $K_M = 12$, $K_I = 39$ on the grid, close to the least-squares estimate. The basin is still long in the K_I direction: grid points from 28 to 54 mM fit almost as well. K_I is identifiable from these data, although only loosely determined.

Grid search becomes hopeless quickly as parameters are added. Sixty values for each of two parameters makes 3600 evaluations; sixty values for each of ten parameters would make about 6×10^{17}. This is the curse of dimensionality (book, §6.5.5), and it is why grid search is used for small problems, for a first look at a landscape, or to choose starting points for a local method.

4.4 Evolutionary algorithms

Between local methods, which are fast but need a good starting guess, and grid search, which is thorough but expensive, lie stochastic global methods such as simulated annealing and evolutionary algorithms (book, §6.3.4). Evolutionary algorithms borrow the logic of

natural selection: a *population* of parameter sets is varied at random, and the better ones are more likely to survive into the next generation.

The algorithm below is one of the simplest of them, a (10 + 40) evolution strategy. It keeps 10 parent parameter sets. In each generation it makes 40 children by adding small random changes ("mutations") to copies of randomly chosen parents, and the best 10 of all 50 become the next parents. The mutations shrink slowly from one generation to the next, so the search moves from exploring widely to refining locally. The parameters are searched on a logarithmic scale, which suits parameters whose plausible values span orders of magnitude.

MATLAB

Core MATLAB only.

```
%% A simple evolutionary algorithm for parameter estimation
% The same model and data as in fit_inhibition (D1 and D2 together), but
% now estimated by a (10 + 40) evolution strategy, which needs no starting
% guess and no gradients:
%   1. start from 50 random parameter sets and keep the best 10 (parents);
%   2. each generation, make 40 children, each a copy of a random parent
%      with small random changes to its parameters ('mutation');
%   3. keep the best 10 of parents and children; shrink the mutations a
%      little; repeat.
% Parameters are searched on a log scale, from 0.1 to about 316.
% Requires: core MATLAB only.

data = readmatrix(fullfile(sbp.data_dir(), 'competitive_inhibition.csv'));
S = [data(:, 1); data(:, 1)];
I = [10 * ones(20, 1); 20 * ones(20, 1)];
v = [data(:, 2); data(:, 3)];
delta = 0.1;
sse = @(log_p) sum(((10^log_p(1) * S ./ ...
    (10^log_p(2) * (1 + I / 10^log_p(3)) + S) - v) ./ (delta * v)) .^ 2);

rng(1);                             % fixed seed: the same run every time
lo = -1.0; hi = 2.5;                % bounds on log10 of each parameter
n_parents = 10; n_children = 40; n_generations = 60;
sigma = 0.5;                        % initial mutation size (log10 units)

% 1. Random start; keep the best 10
pop = lo + (hi - lo) * rand(50, 3);
cost = arrayfun(@(k) sse(pop(k, :)), (1:50)');
[~, order] = sort(cost);
parents = pop(order(1:n_parents), :);
parent_cost = cost(order(1:n_parents));
snapshots = {parents};
```

```
snapshot_gens = 0;
best_per_generation = zeros(1, n_generations + 1);
best_per_generation(1) = parent_cost(1);
n_evaluations = 50;

for g = 1:n_generations
    % 2. Children: mutated copies of randomly chosen parents
    pick = randi(n_parents, n_children, 1);
    children = parents(pick, :) + sigma * randn(n_children, 3);
    children = min(max(children, lo), hi);
    child_cost = arrayfun(@(k) sse(children(k, :)), (1:n_children)');
    n_evaluations = n_evaluations + n_children;
    % 3. Survival of the fittest, among parents and children together
    everyone = [parents; children];
    everyone_cost = [parent_cost; child_cost];
    [~, order] = sort(everyone_cost);
    parents = everyone(order(1:n_parents), :);
    parent_cost = everyone_cost(order(1:n_parents));
    sigma = sigma * 0.93;
    best_per_generation(g + 1) = parent_cost(1);
    if ismember(g, [5 20 n_generations])
        snapshots{end + 1} = parents; %#ok<SAGROW>
        snapshot_gens(end + 1) = g; %#ok<SAGROW>
    end
end

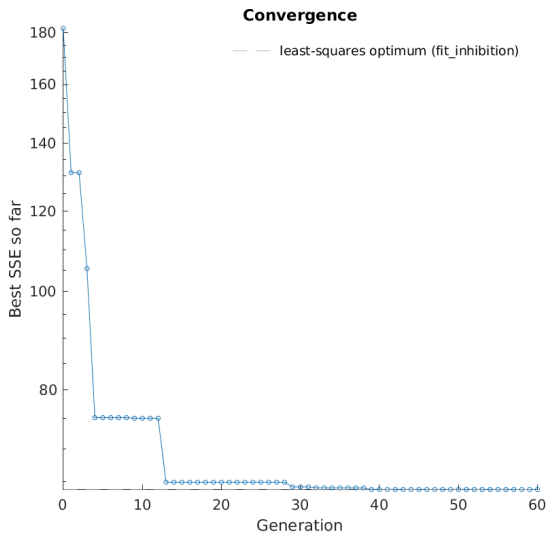
p = 10 .^ parents(1, :);
fprintf(['Best after %d generations (%d SSE evaluations): vmax = %.3f, '
 ↪  ...
    'KM = %.3f, KI = %.3f, SSE = %.4f\n'], n_generations, n_evaluations,
     ↪  ...
    p(1), p(2), p(3), parent_cost(1));

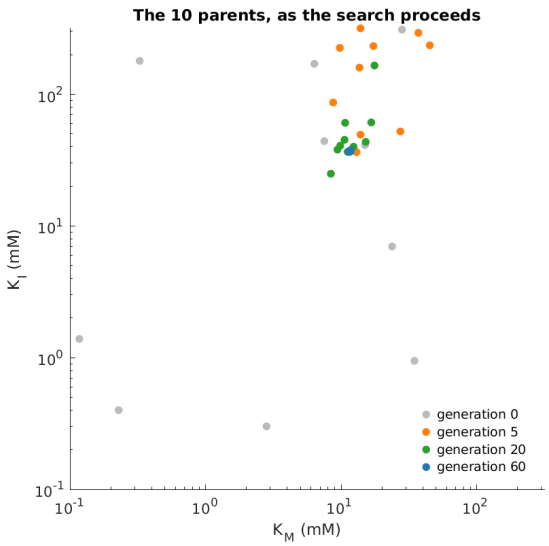
%% Plot
fig = figure('Position', [100 100 1000 400]);
subplot(1, 2, 1)
semilogy(0:n_generations, best_per_generation, 'o-', ...
    'Color', [0.122 0.467 0.706], 'MarkerSize', 3);
hold on
yline(63.7765, '--', 'Color', [0.6 0.6 0.6]);
xlabel('Generation'); ylabel('Best SSE so far'); title('Convergence');
legend({'', 'least-squares optimum (fit\_inhibition)'}, 'Box', 'off');
box off

subplot(1, 2, 2)
colors = [0.733 0.733 0.733; 1.000 0.498 0.055;
```

```
          0.173 0.627 0.173; 0.122 0.467 0.706];
for k = 1:numel(snapshots)
    P = 10 .^ snapshots{k};
    loglog(P(:, 2), P(:, 3), 'o', 'Color', colors(k, :), ...
        'MarkerFaceColor', colors(k, :), 'MarkerSize', 5);
    hold on
end
xlim([0.1 316]); ylim([0.1 316]);
xlabel('K_M (mM)'); ylabel('K_I (mM)');
title('The 10 parents, as the search proceeds');
legend(compose('generation %d', snapshot_gens), 'Location', 'southeast',
 ↪  ...
    'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-evolutionary');

results.best = struct('SSE', parent_cost(1), 'vmax', p(1), 'KM', p(2), ...
    'KI', p(3));
results.n_evaluations = n_evaluations;
results.best_per_generation = best_per_generation;
sbp.save_results(mfilename('fullpath'), results);
```





Python

Needs NumPy and Matplotlib.

```
"""A simple evolutionary algorithm for parameter estimation.

The same model and data as in fit_inhibition (D1 and D2 together), but
now estimated by a (10 + 40) evolution strategy, which needs no starting
```

```
guess and no gradients:
  1. start from 50 random parameter sets and keep the best 10 (parents);
  2. each generation, make 40 children, each a copy of a random parent
     with small random changes to its parameters ('mutation');
  3. keep the best 10 of parents and children; shrink the mutations a
     little; repeat.
Parameters are searched on a log scale, from 0.1 to about 316.
"""
import numpy as np
import matplotlib.pyplot as plt
import sbp

data = np.loadtxt(sbp.DATA / "competitive_inhibition.csv", delimiter=",",
                  skiprows=1)
S = np.r_[data[:, 0], data[:, 0]]
I = np.r_[np.full(20, 10.0), np.full(20, 20.0)]
v = np.r_[data[:, 1], data[:, 2]]
delta = 0.1


def sse(log_p):
    vmax, KM, KI = 10.0 ** log_p
    v_pred = vmax * S / (KM * (1 + I / KI) + S)
    return np.sum(((v_pred - v) / (delta * v)) ** 2)


rng = np.random.default_rng(1)     # fixed seed: the same run every time
lo, hi = -1.0, 2.5                 # bounds on log10 of each parameter
n_parents, n_children, n_generations = 10, 40, 60
sigma = 0.5                        # initial mutation size (log10 units)

# 1. Random start; keep the best 10
pop = rng.uniform(lo, hi, size=(50, 3))
cost = np.array([sse(p) for p in pop])
keep = np.argsort(cost)[:n_parents]
parents, parent_cost = pop[keep], cost[keep]
snapshots = {0: parents.copy()}
best_per_generation = [parent_cost[0]]
n_evaluations = 50

for g in range(1, n_generations + 1):
    # 2. Children: mutated copies of randomly chosen parents
    pick = rng.integers(n_parents, size=n_children)
    children = parents[pick] + sigma * rng.standard_normal((n_children, 3))
    children = np.clip(children, lo, hi)
    child_cost = np.array([sse(p) for p in children])
```

```
    n_evaluations += n_children
    # 3. Survival of the fittest, among parents and children together
    everyone = np.vstack([parents, children])
    everyone_cost = np.r_[parent_cost, child_cost]
    keep = np.argsort(everyone_cost)[:n_parents]
    parents, parent_cost = everyone[keep], everyone_cost[keep]
    sigma *= 0.93
    best_per_generation.append(parent_cost[0])
    if g in (5, 20):
        snapshots[g] = parents.copy()
snapshots[n_generations] = parents.copy()

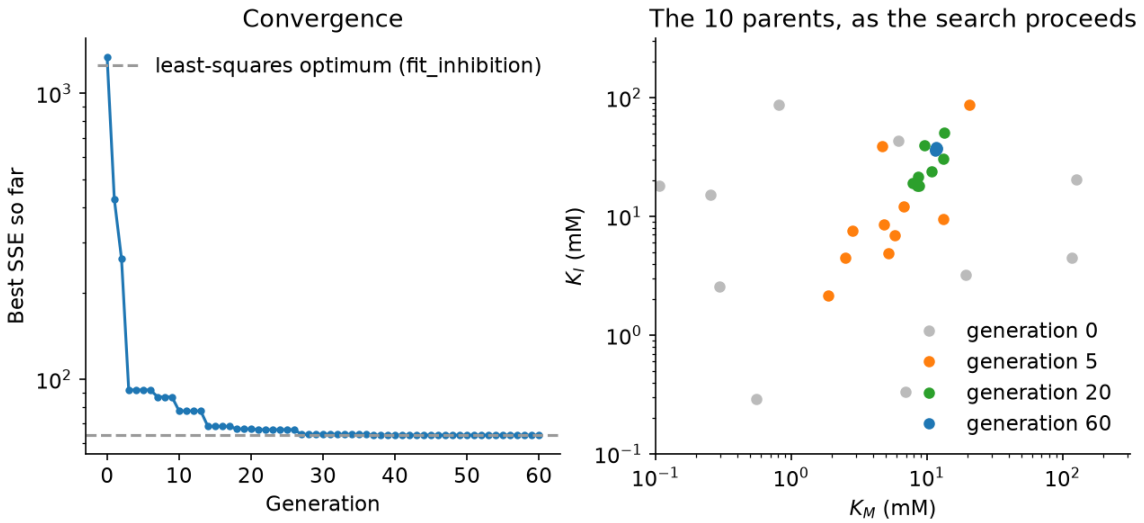
vmax, KM, KI = 10.0 ** parents[0]
print(f"Best after {n_generations} generations ({n_evaluations} SSE "
      f"evaluations): vmax = {vmax:.3f}, KM = {KM:.3f}, KI = {KI:.3f}, "
      f"SSE = {parent_cost[0]:.4f}")

# --- Plot
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
ax1.semilogy(best_per_generation, "o-", color="#1f77b4", ms=3)
ax1.axhline(63.7765, color="#999999", ls="--",
            label="least-squares optimum (fit_inhibition)")
ax1.set_xlabel("Generation")
ax1.set_ylabel("Best SSE so far")
ax1.set_title("Convergence")
ax1.legend(frameon=False)

colors = {0: "#bbbbbb", 5: "#ff7f0e", 20: "#2ca02c", n_generations:
 ↪  "#1f77b4"}
for g, P in snapshots.items():
    ax2.plot(10.0 ** P[:, 1], 10.0 ** P[:, 2], "o", color=colors[g], ms=5,
             label=f"generation {g}")
ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_xlim(0.1, 316)
ax2.set_ylim(0.1, 316)
ax2.set_xlabel("$K_M$ (mM)")
ax2.set_ylabel("$K_I$ (mM)")
ax2.set_title("The 10 parents, as the search proceeds")
ax2.legend(frameon=False, loc="lower right")
sbp.save_figure(fig, __file__, "fig-evolutionary")

sbp.save_results(__file__, {
    "best": {"SSE": parent_cost[0], "vmax": vmax, "KM": KM, "KI": KI},
    "n_evaluations": n_evaluations,
```

```
    "best_per_generation": best_per_generation,
})
sbp.show()
```



Starting from parameter sets scattered across more than three orders of magnitude, the population gathers into the basin we saw in the landscape and closes in on the least-squares estimate. After 60 generations, or 2450 evaluations of the SSE, the best parameter set is within a few per cent of the least-squares estimates, and its SSE is within 0.02% of the least-squares minimum. Notice that, on the way, the parents spread out along the same valley as before: the algorithm finds K_M^{app} well before it pins down K_M and K_I separately.

MATLAB and Python draw different random numbers, so the two versions follow different paths and end at slightly different points; the automated checks require both to land within 5% of the least-squares optimum. Run either version with a few different seeds to see how much the result varies.

The trade-off is clear from this small problem. The evolutionary algorithm needed no starting guess and no derivatives, but it used about 2450 evaluations to reach an approximate answer, whereas the least-squares solver reached the exact optimum in a few dozen evaluations, counting those used to estimate derivatives. For larger models, a common strategy is to combine the two: a global method to find the right basin, followed by a local method to finish the job.

4.5 Exercises

1. Fit D2 alone from the eight starting points. Which combination of K_M and K_I do all the fits agree on?
2. Fit D1 and D2 together using the *unweighted* SSE (book, Eq. 6.2). How do the estimates change, and which data points now dominate the fit?

3. Estimate the uncertainty of the parameters fitted to D1 and D2 together. Both least-squares solvers return the Jacobian J of the residuals at the solution; the covariance of the estimates is approximately $(J^\top J)^{-1}$. Compute the correlation between K_M and K_I (book, Eq. 6.11). Is it close to ± 1? Repeat for D1 alone: what happens to $J^\top J$?
4. Book Exercise 6.1: evaluate the SSE of the parameter sets θ_1, θ_2 and θ_3 given there. Which is best, and can the data tell them apart?
5. Run the evolutionary algorithm with ten different random seeds. How much do the final estimates vary? Then halve the number of generations, or start with a smaller mutation size, and repeat.

4.6 Going further

Here the model was a single algebraic rate law. Fitting a system of ODEs to time-course data works in exactly the same way, with the ODE solver of Chapter 3 called inside the residual function; the questions of identifiability and of local versus global search only become more pressing.

5 Boolean networks

ODE models need rate laws and parameters, and for gene regulatory and signalling networks these are rarely known. Often all we know is who activates or represses whom, and whether a gene is on or off. Boolean networks work at exactly that level: each gene is either ON (1) or OFF (0), time moves in discrete steps, and each gene's next state is a logical function of the current states of its regulators. Such models have no parameters to estimate, and yet they can reproduce the qualitative behaviour of quite complex systems.

> **In the book**
>
> Chapter 7, *Discrete Dynamic Models: Boolean Networks*: transfer functions (§7.2), state transitions and attractors (§7.2.1), synchronous and asynchronous updates (§7.2.2), and other formalisms, from probabilistic Boolean networks to Petri nets (§7.3).

5.1 Transfer functions, states and attractors

A Boolean network of n variables is defined by a **transfer function** for each variable, built from AND, OR and NOT:

$$X_i(t) = F_i(X_1(t-1), X_2(t-1), \ldots, X_n(t-1)).$$

With **synchronous** updating, as written here, all variables are updated at once from the previous state. The network then has 2^n possible states, and each state has exactly one successor. Starting anywhere, the network must eventually revisit a state, and from then on it repeats itself. The states it repeats form an **attractor**: a single state that maps to itself (a steady state, or point attractor), or a cycle of states. The states that lead to an attractor form its **basin**.

For small networks, we can simply examine every state. Our first example is the three-gene network of the book's Fig. 7.1:

$$g_1^* = g_2 \text{ AND NOT } g_3, \qquad g_2^* = \text{NOT } g_1, \qquad g_3^* = (\text{NOT } g_3) \text{ OR } g_2,$$

where g_i^* is the next state of g_i. States are written as three digits, $g_1 g_2 g_3$; state 011, for example, has g_1 OFF and g_2, g_3 ON.

MATLAB

Core MATLAB only.

```
%% A three-gene Boolean network (book Fig. 7.1, Eqs. 7.3)
%     g1* = g2 AND NOT g3
%     g2* = NOT g1
%     g3* = NOT g3 OR g2
% With synchronous updating, all genes are updated at once from the
% previous state. We list the successor of each of the 2^3 = 8 states,
% and find the attractors (the states the network ends up cycling
% through) and their basins (the states that lead to them).
% Requires: core MATLAB only.

n = 3;
update = @(x) [x(2) && ~x(3), ~x(1), ~x(3) || x(2)];
% State number <-> vector of 0/1, first gene as the leftmost digit
to_bits = @(s) bitget(s, n:-1:1);
to_number = @(x) sum(x .* 2.^(n-1:-1:0));
label = @(s) sprintf('%d', to_bits(s));

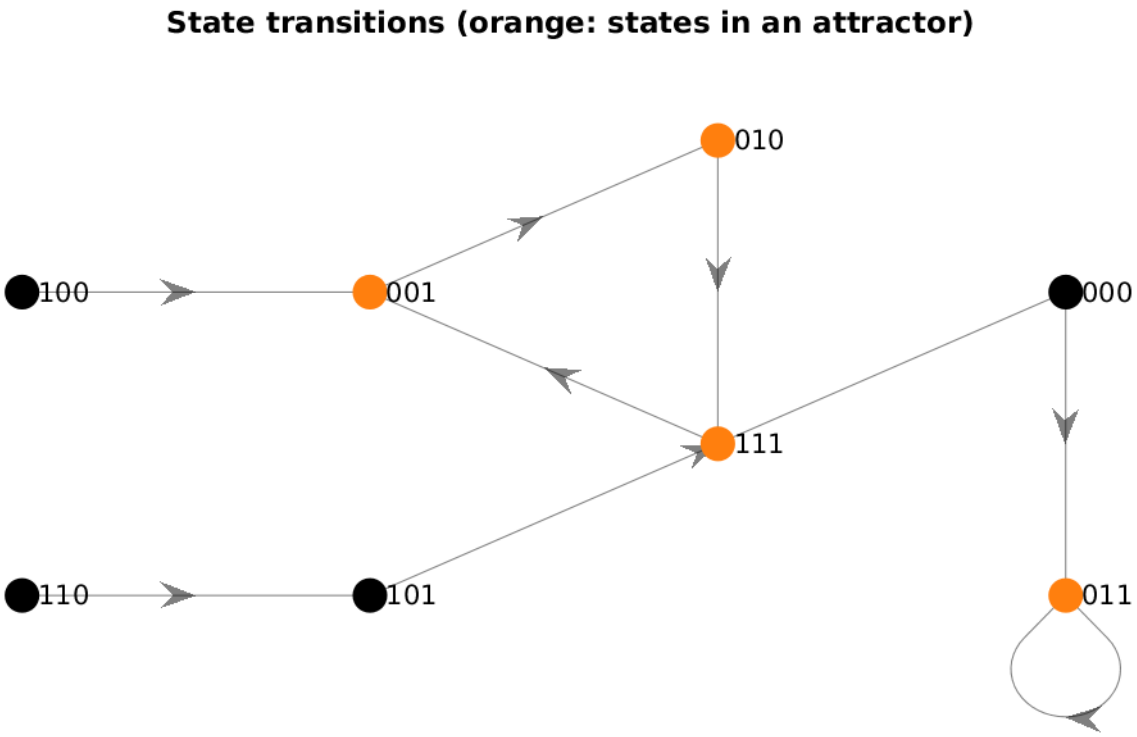
% Successor of every state (states are numbered 0 to 2^n - 1)
succ = zeros(1, 2^n);
for s = 0:2^n - 1
    succ(s + 1) = to_number(double(update(to_bits(s))));
    fprintf('%s => %s\n', label(s), label(succ(s + 1)));
end

% Attractors: follow each state until a state repeats; the repeating part
% is the attractor. Every state is assigned to the attractor it reaches.
attractors = {};
reaches = zeros(1, 2^n);
for s0 = 0:2^n - 1
    path = [];
    s = s0;
    while ~ismember(s, path)
        path(end + 1) = s; %#ok<AGROW>
        s = succ(s + 1);
    end
    cycle = path(find(path == s):end);
    key = min(cycle);
    if ~any(cellfun(@(a) min(a) == key, attractors))
        i = find(cycle == key);                    % start each cycle at
        attractors{end + 1} = cycle([i:end, 1:i - 1]); %#ok<SAGROW>
    end
    reaches(s0 + 1) = key;
end
```

```
basin_sizes = cellfun(@(a) sum(reaches == min(a)), attractors);
for k = 1:numel(attractors)
    a = attractors{k};
    fprintf('Attractor of period %d: %s; basin of %d states\n', numel(a),
    ↪  ...
        strjoin(arrayfun(label, a, 'UniformOutput', false), ' -> '), ...
        basin_sizes(k));
end

%% Plot the state transition map
labels = arrayfun(label, 0:2^n - 1, 'UniformOutput', false);
G = digraph(1:2^n, succ + 1, [], labels);   % MATLAB counts nodes from 1
% Same positions as the Python version, for states 000, 001, ..., 111
x = [3 1 2 3 0 1 0 2];
y = [1 1 1.5 0 1 0 0 0.5];
in_attractor = ismember(0:2^n - 1, [attractors{:}]);
fig = figure('Position', [100 100 700 350]);
h = plot(G, 'XData', x, 'YData', y, 'NodeLabel', labels, ...
    'MarkerSize', 12, 'NodeColor', 'k', 'EdgeColor', 'k', 'ArrowSize', 10);
highlight(h, find(in_attractor), 'NodeColor', [1.000 0.498 0.055]);
title('State transitions (orange: states in an attractor)');
axis off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-three-genes');

results.successor = arrayfun(label, succ, 'UniformOutput', false);
results.attractors = cellfun(@(a) arrayfun(label, a, ...
    'UniformOutput', false), attractors, 'UniformOutput', false);
results.basin_sizes = basin_sizes;
sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs NetworkX (for drawing the state transitions).

```
"""A three-gene Boolean network (book Fig. 7.1, Eqs. 7.3).

    g1* = g2 AND NOT g3
    g2* = NOT g1
    g3* = NOT g3 OR g2

With synchronous updating, all genes are updated at once from the
previous state. We list the successor of each of the 2^3 = 8 states,
and find the attractors (the states the network ends up cycling
through) and their basins (the states that lead to them).
"""
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import sbp

genes = ["g1", "g2", "g3"]
n = len(genes)


def update(x):
```

```
    g1, g2, g3 = x
    return (g2 and not g3, not g1, (not g3) or g2)


def to_bits(s):
    """State number -> tuple of 0/1, first gene as the leftmost digit."""
    return tuple((s >> (n - 1 - i)) & 1 for i in range(n))


def to_number(x):
    return sum(int(b) << (n - 1 - i) for i, b in enumerate(x))


def label(s):
    return "".join(map(str, to_bits(s)))

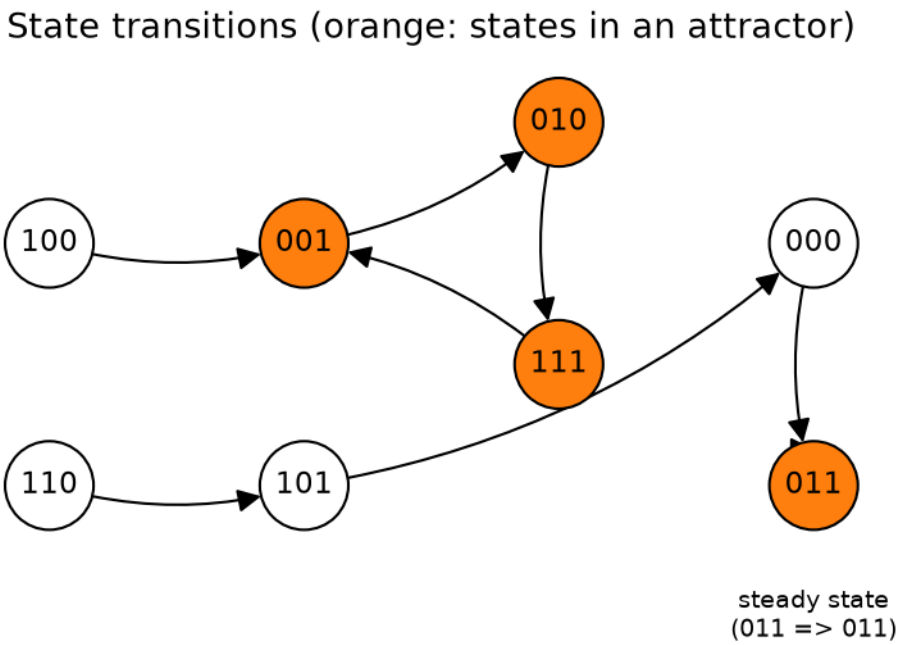

# Successor of every state
succ = [to_number(update(to_bits(s))) for s in range(2 ** n)]
for s in range(2 ** n):
    print(f"{label(s)} => {label(succ[s])}")

# Attractors: follow each state until a state repeats; the repeating part
# is the attractor. Every state is assigned to the attractor it reaches.
attractor_of = {}
attractors = []
for s in range(2 ** n):
    path = []
    while s not in path:
        path.append(s)
        s = succ[s]
    cycle = path[path.index(s):]
    key = min(cycle)
    if key not in [min(a) for a in attractors]:
        i = cycle.index(key)                    # start each cycle at its
        attractors.append(cycle[i:] + cycle[:i])  # smallest state
    attractor_of[path[0]] = key
basins = {min(a): sum(1 for v in attractor_of.values() if v == min(a))
          for a in attractors}
for a in attractors:
    print(f"Attractor of period {len(a)}: "
          f"{' -> '.join(label(s) for s in a)}; basin of {basins[min(a)]} "
          f"states")

# --- Plot the state transition map
G = nx.DiGraph([(label(s), label(succ[s])) for s in range(2 ** n)])
```

```
pos = {"100": (0, 1), "110": (0, 0), "001": (1, 1), "101": (1, 0),
       "010": (2, 1.5), "111": (2, 0.5), "000": (3, 1), "011": (3, 0)}
in_attractor = {label(s) for a in attractors for s in a}
sbp.style()
fig, ax = plt.subplots(figsize=(7, 3.5))
nx.draw_networkx_nodes(G, pos, ax=ax, node_size=1100, edgecolors="k",
                       node_color=["#ff7f0e" if v in in_attractor else
 ↪  "white"
                                   for v in G.nodes])
nx.draw_networkx_labels(G, pos, ax=ax, font_size=11)
nx.draw_networkx_edges(G, pos, ax=ax, node_size=1100, arrowsize=18,
                       connectionstyle="arc3,rad=0.15")
ax.annotate("steady state\n(011 => 011)", pos["011"], xytext=(0, -38),
            textcoords="offset points", ha="center", va="top", fontsize=9)
ax.set_title("State transitions (orange: states in an attractor)")
ax.set_axis_off()
ax.margins(0.12)
sbp.save_figure(fig, __file__, "fig-three-genes")

sbp.save_results(__file__, {
    "successor": [label(s) for s in succ],
    "attractors": [[label(s) for s in a] for a in attractors],
    "basin_sizes": [basins[min(a)] for a in attractors],
})
sbp.show()
```



The transitions are those listed in the book (§7.2.1). The network has two attractors, each with a basin of four states: a steady state, 011, in which g_1 stays off while g_2 and g_3 stay on; and a cycle of period three, $001 \to 010 \to 111 \to 001$, in which the genes switch on and off in a repeating pattern. Which of the two the network ends up in depends only on where it starts, and no state can reach both.

Following each state forward until it repeats, as the code does, finds every attractor of a synchronous network. With n variables there are 2^n states to examine, which is quick up to $n \approx 20$; beyond that, attractors must be sought by sampling starting states, or by specialised algorithms.

5.2 The mammalian cell cycle

The second example is a model of the mammalian cell cycle by Fauré and colleagues (Fauré et al. 2006), with ten variables: the cyclins CycD, CycE, CycA and CycB, the retinoblastoma protein Rb, the transcription factor E2F, the inhibitor p27, and the regulators Cdc20, Cdh1 and UbcH10 (Ubc). Its transfer functions are the book's Eqs. 7.4. CycD stands for the growth signal: it has no regulators in the model, so it simply keeps its initial value.

With ten variables there are 1024 states, still few enough to examine every one. We do so twice. The first time with synchronous updating. The second time with **asynchronous** updating, which drops the assumption that every process takes exactly one time step: at each step, any one of the variables whose transfer function calls for a change may change, in any order. A state can then have several possible successors, so the states and their possible transitions form a directed graph, the **state transition graph**. An attractor is now a set of states that, once entered, can never be left: a strongly connected component of that graph with no transitions leading out of it, which graph libraries find directly.

MATLAB

Core MATLAB only.

```
%% A Boolean model of the mammalian cell cycle (Fauré et al., 2006)
% Ten genes and proteins; the update rules are the book's Eqs. 7.4. With
% 2^10 = 1024 possible states, we can examine every one of them:
%   1. synchronous updating, as in the previous example;
%   2. asynchronous updating, where one variable changes at a time, in
%      any order. The states then form a 'state transition graph', and the
%      attractors are the groups of states that, once entered, are never
%      left: its terminal strongly connected components.
% Requires: core MATLAB only.

names = {'CycD', 'Rb', 'E2F', 'CycE', 'CycA', 'p27', 'Cdc20', 'Cdh1', ...
         'Ubc', 'CycB'};
n = numel(names);
to_bits = @(s) double(bitget(s, n:-1:1));
```

```
to_number = @(x) sum(x .* 2.^(n-1:-1:0));

N = 2^n;
target = zeros(1, N);
for s = 0:N - 1
    target(s + 1) = to_number(double(update(to_bits(s))));
end

%% 1. Synchronous updating: follow every state to its attractor
keys = []; periods = []; reaches = zeros(1, N);
for s0 = 0:N - 1
    path = [];
    s = s0;
    while ~ismember(s, path)
        path(end + 1) = s; %#ok<AGROW>
        s = target(s + 1);
    end
    cycle = path(find(path == s):end);
    if ~ismember(min(cycle), keys)
        keys(end + 1) = min(cycle); %#ok<AGROW>
        periods(end + 1) = numel(cycle); %#ok<AGROW>
    end
    reaches(s0 + 1) = min(cycle);
end
[keys, order] = sort(keys);
periods = periods(order);
fprintf('Synchronous updating:\n');
for k = 1:numel(keys)
    b = to_bits(keys(k));
    sync_summary(k) = struct('period', periods(k), ...
        'basin', sum(reaches == keys(k)), 'CycD', b(1)); %#ok<SAGROW>
    fprintf('  attractor of period %d (CycD = %d), basin of %d states\n',
     ↪  ...
        periods(k), b(1), sync_summary(k).basin);
end

%% 2. Asynchronous updating: from each state, any one variable whose
% rule says it should change may change next
src = []; dst = [];
for s = 0:N - 1
    x = to_bits(s);
    fx = to_bits(target(s + 1));
    for i = find(x ~= fx)
        y = x;
        y(i) = fx(i);
        src(end + 1) = s + 1; %#ok<AGROW>
```

```
        dst(end + 1) = to_number(y) + 1; %#ok<AGROW>
    end
end
G = digraph(src, dst, [], N);
comp = conncomp(G, 'Type', 'strong');  % component of each state
C = condensation(G);                  % one node per component
terminal = find(outdegree(C) == 0)';
sizes = zeros(size(terminal)); cycd = zeros(size(terminal));
for k = 1:numel(terminal)
    members = find(comp == terminal(k)) - 1;  % back to states 0..N-1
    sizes(k) = numel(members);
    b = to_bits(min(members));
    cycd(k) = b(1);
end
[~, order] = sortrows([cycd' sizes']);
fprintf('Asynchronous updating:\n');
for k = 1:numel(order)
    async_summary(k) = struct('size', sizes(order(k)), ...
        'CycD', cycd(order(k))); %#ok<SAGROW>
    fprintf('  attractor of %d states (CycD = %d)\n', sizes(order(k)), ...
        cycd(order(k)));
end

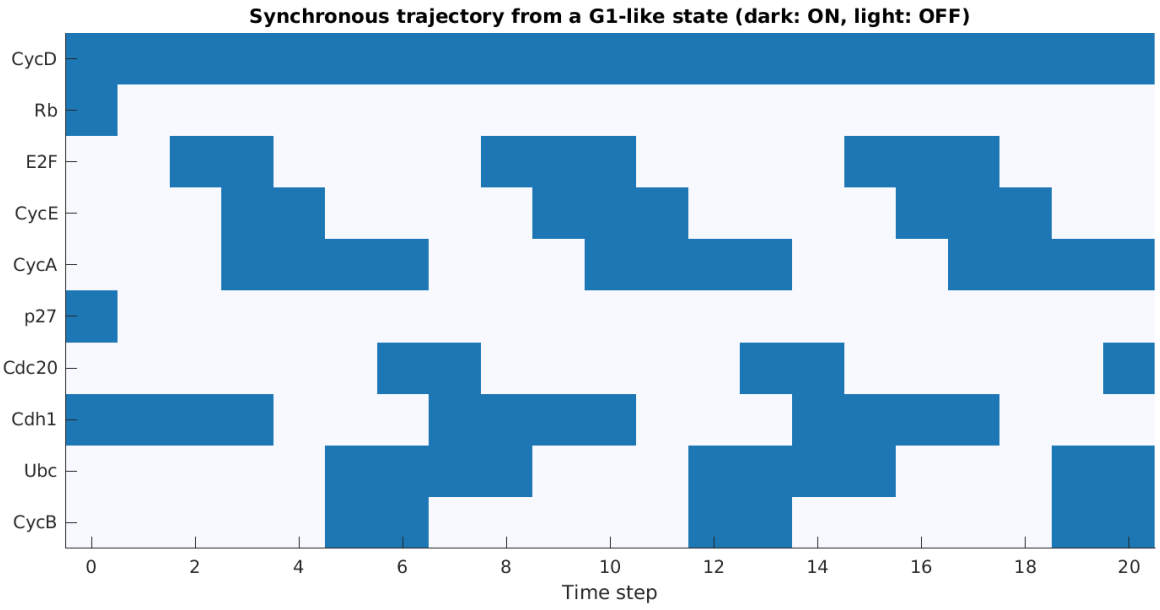
%% A synchronous trajectory: start from a G1-like state, with CycD on
x0 = double(ismember(names, {'CycD', 'Rb', 'p27', 'Cdh1'}));
traj = zeros(21, n);
traj(1, :) = x0;
for t = 2:21
    traj(t, :) = double(update(traj(t - 1, :)));
end

fig = figure('Position', [100 100 800 360]);
imagesc(0:20, 1:n, traj', [0 1]);
colormap([0.97 0.98 1; 0.12 0.47 0.71]);   % OFF: light, ON: dark
set(gca, 'YTick', 1:n, 'YTickLabel', names, 'XTick', 0:2:20, 'Box', 'off');
xlabel('Time step');
title('Synchronous trajectory from a G1-like state (dark: ON, light:
 ↪  OFF)');
sbp.save_figure(fig, mfilename('fullpath'), 'fig-cell-cycle');

results.synchronous = sync_summary;
results.asynchronous = async_summary;
results.trajectory = traj;
sbp.save_results(mfilename('fullpath'), results);

%% Local function
```

```
function y = update(x)
% Book Eqs. 7.4 (Fauré et al., 2006)
x = num2cell(logical(x));
[CycD, Rb, E2F, CycE, CycA, p27, Cdc20, Cdh1, Ubc, CycB] = x{:};
y = [CycD, ...
     (~CycA && ~CycB && ~CycD && ~CycE) || (p27 && ~CycB && ~CycD), ...
     (~Rb && ~CycA && ~CycB) || (p27 && ~Rb && ~CycB), ...
     E2F && ~Rb, ...
     (E2F && ~Rb && ~Cdc20 && ~(Cdh1 && Ubc)) || ...
         (CycA && ~Rb && ~Cdc20 && ~(Cdh1 && Ubc)), ...
     (~CycD && ~CycE && ~CycA && ~CycB) || ...
         (p27 && ~(CycE && CycA) && ~CycB && ~CycD), ...
     CycB, ...
     (~CycA && ~CycB) || Cdc20 || (p27 && ~CycB), ...
     ~Cdh1 || (Cdh1 && Ubc && (Cdc20 || CycA || CycB)), ...
     ~Cdc20 && ~Cdh1];
end
```



Python

Needs NetworkX.

```
"""A Boolean model of the mammalian cell cycle (Fauré et al., 2006).

Ten genes and proteins; the update rules are the book's Eqs. 7.4. With
2^10 = 1024 possible states, we can examine every one of them:
  1. synchronous updating, as in the previous example;
  2. asynchronous updating, where one variable changes at a time, in
```

```
     any order. The states then form a 'state transition graph', and the
     attractors are the groups of states that, once entered, are never
     left: its terminal strongly connected components.
"""
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import sbp

names = ["CycD", "Rb", "E2F", "CycE", "CycA", "p27", "Cdc20", "Cdh1",
         "Ubc", "CycB"]
n = len(names)


def update(x):
    CycD, Rb, E2F, CycE, CycA, p27, Cdc20, Cdh1, Ubc, CycB = x
    return (
        CycD,
        (not CycA and not CycB and not CycD and not CycE)
        or (p27 and not CycB and not CycD),
        (not Rb and not CycA and not CycB) or (p27 and not Rb and not
 ↪  CycB),
        E2F and not Rb,
        (E2F and not Rb and not Cdc20 and not (Cdh1 and Ubc))
        or (CycA and not Rb and not Cdc20 and not (Cdh1 and Ubc)),
        (not CycD and not CycE and not CycA and not CycB)
        or (p27 and not (CycE and CycA) and not CycB and not CycD),
        CycB,
        (not CycA and not CycB) or Cdc20 or (p27 and not CycB),
        (not Cdh1) or (Cdh1 and Ubc and (Cdc20 or CycA or CycB)),
        not Cdc20 and not Cdh1,
    )


def to_bits(s):
    return tuple((s >> (n - 1 - i)) & 1 for i in range(n))


def to_number(x):
    return sum(int(b) << (n - 1 - i) for i, b in enumerate(x))


states = range(2 ** n)
target = [to_number(update(to_bits(s))) for s in states]

# --- 1. Synchronous updating: follow every state to its attractor
```

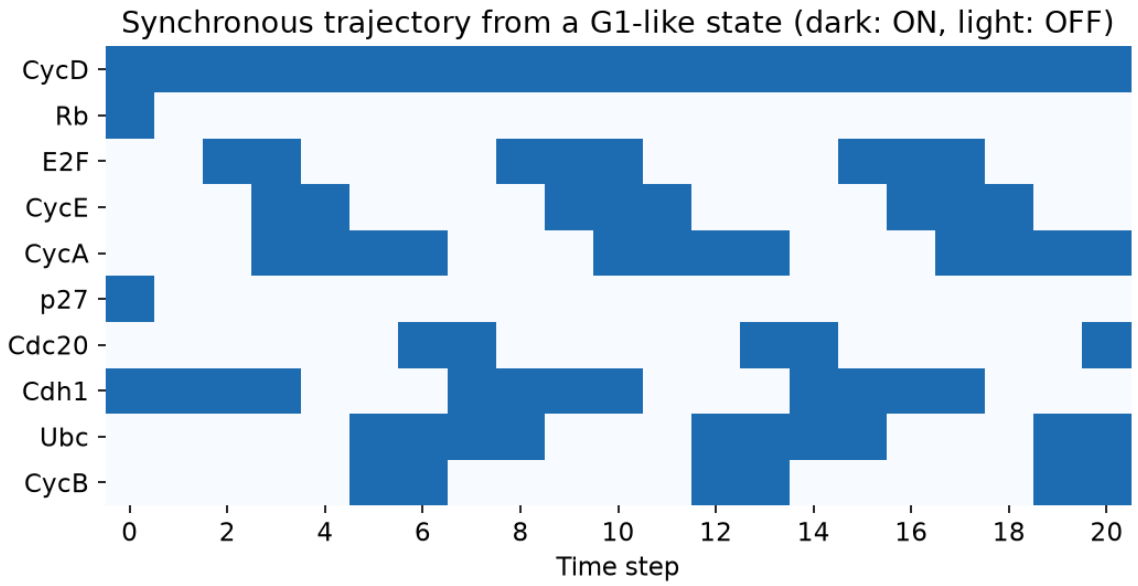
```
sync_attractors = {}           # smallest state of the cycle -> cycle
reaches = []
for s in states:
    path = []
    while s not in path:
        path.append(s)
        s = target[s]
    cycle = path[path.index(s):]
    sync_attractors.setdefault(min(cycle), cycle)
    reaches.append(min(cycle))
sync_summary = [{"period": len(c), "basin": reaches.count(k),
                 "CycD": to_bits(k)[0]}
                for k, c in sorted(sync_attractors.items())]
print("Synchronous updating:")
for a in sync_summary:
    print(f"  attractor of period {a['period']} (CycD = {a['CycD']}), "
          f"basin of {a['basin']} states")

# --- 2. Asynchronous updating: from each state, any one variable whose
# rule says it should change may change next
G = nx.DiGraph()
G.add_nodes_from(states)
for s in states:
    x, fx = to_bits(s), to_bits(target[s])
    for i in range(n):
        if x[i] != fx[i]:
            y = list(x)
            y[i] = fx[i]
            G.add_edge(s, to_number(y))
C = nx.condensation(G)       # one node per strongly connected component
terminal = [c for c in C.nodes if C.out_degree(c) == 0]
async_summary = sorted(
    ({"size": len(C.nodes[c]["members"]),
      "CycD": to_bits(min(C.nodes[c]["members"]))[0]} for c in terminal),
    key=lambda a: (a["CycD"], a["size"]))
print("Asynchronous updating:")
for a in async_summary:
    print(f"  attractor of {a['size']} states (CycD = {a['CycD']})")

# --- A synchronous trajectory: start from a G1-like state, with CycD on
x0 = dict.fromkeys(names, 0)
x0.update(CycD=1, Rb=1, p27=1, Cdh1=1)
traj = [tuple(x0[g] for g in names)]
for _ in range(20):
    traj.append(tuple(int(v) for v in update(traj[-1])))
traj = np.array(traj)
```

```
sbp.style()
fig, ax = plt.subplots(figsize=(8, 3.6))
ax.imshow(traj.T, cmap="Blues", aspect="auto", vmin=0, vmax=1.3)
ax.set_yticks(range(n), names)
ax.set_xticks(range(0, 21, 2))
ax.set_xlabel("Time step")
ax.set_title("Synchronous trajectory from a G1-like state "
             "(dark: ON, light: OFF)")
for spine in ax.spines.values():
    spine.set_visible(False)
sbp.save_figure(fig, __file__, "fig-cell-cycle")

sbp.save_results(__file__, {
    "synchronous": sync_summary,
    "asynchronous": async_summary,
    "trajectory": traj,
})
sbp.show()
```



With synchronous updating, each half of the state space (CycD off, or CycD on) leads to a single attractor. Without the growth signal, the network settles into a steady state, a quiescent cell with Rb, p27 and Cdh1 active. With the growth signal, it settles into a cycle of seven states, in which E2F, then cyclin E, cyclin A and cyclin B rise and fall in turn, much as they do through the phases of a real cell cycle (figure). These are exactly the attractors reported by Fauré and colleagues.

Asynchronous updating keeps the quiescent steady state, but the seven-state cycle becomes

a **complex attractor** of 112 states: the same sequence of events can now happen with many different relative timings, and the network wanders among all of them without ever leaving. Synchronous updating is easy to simulate and to reason about, but its fixed timing can create or destroy cycles that would not survive in a real cell; comparing both, as here, is a useful check on any conclusion drawn from a Boolean model (book, §7.5).

5.3 Exercises

1. In the three-gene network, change the rule for g_3 to $g_3^* = (\text{NOT } g_3)$ AND g_2. Find the new attractors and their basins.
2. Simulate a **knock-out** in the cell-cycle model: fix one variable at 0 (for example, Cdh1) and find the attractors again, with both kinds of updating. Which knock-outs abolish the cycle?
3. Start the cell-cycle model in its quiescent steady state, switch CycD on, and update synchronously. Through which states does the network pass before it joins the seven-state cycle, and how many steps does that take?
4. Write a function that generates a random Boolean network (Kauffman's "NK" network): n genes, each with K randomly chosen regulators and a random truth table. For $n = 12$ and $K = 1, 2, 3, 4$, find the number of attractors and their periods, averaged over many networks. How does the behaviour change with K?

5.4 Going further

Boolean networks lie at one end of a spectrum of models. Between them and ODE models sit multi-valued logical models and Petri nets (book, §7.3). The next chapter returns to continuous quantities, but treats the small numbers of molecules inside a single cell as random events.

6 Stochastic simulation

The ODE models of Chapter 3 treat concentrations as smooth, continuous quantities. Inside a single cell, however, many molecules are present in tiny numbers: a gene has one or two copies, and a given mRNA may be present as a handful of molecules. Each reaction is then a discrete, random event, and two genetically identical cells in the same environment can hold quite different numbers of a protein. This chapter simulates such systems with Gillespie's algorithm, and shows how the random fluctuations, or **noise**, depend on the reactions that produce them.

> **In the book**
>
> The book deals with deterministic models, and notes that stochastic models, widely simulated with Gillespie's algorithm, are important when molecule numbers are small (§1.4.3, *Modelling paradigms*). This chapter fills in that topic. For a fuller treatment, see the reviews by Gillespie (2007) and Raj and Oudenaarden (2008).

6.1 Gillespie's algorithm

Consider a set of reactions, each with a **propensity** $a_j(\mathbf{x})$: the probability per unit time that reaction j occurs, given the current numbers of molecules $\mathbf{x}$. For a reaction $\mathrm{M} \to \emptyset$ with rate constant γ, for instance, the propensity is γM, since each of the M molecules can be degraded. Gillespie's **stochastic simulation algorithm** (SSA) generates exact sample paths of such a system (Gillespie 1977):

1. Compute all propensities and their sum, $a_0 = \sum_j a_j(\mathbf{x})$.
2. Draw the time to the next reaction from an exponential distribution with mean $1/a_0$: $\tau = -\ln(r_1)/a_0$, with r_1 uniform on $(0, 1)$.
3. Choose which reaction occurs: reaction j with probability a_j/a_0.
4. Advance time by τ, update $\mathbf{x}$ for that reaction, and repeat.

Both languages have a short implementation of these four steps in the primer's helper package (`sbp.gillespie`), which returns the state at a chosen set of times. Here is the Python version; the MATLAB one, in `matlab/+sbp/gillespie.m`, follows it line by line.

```
"""Gillespie's stochastic simulation algorithm (SSA), shared by
 ↪  examples."""
import numpy as np
```

```
def gillespie(x0, changes, propensities, t_grid, rng):
    """Simulate one trajectory and return the state at the times t_grid.

    x0            initial numbers of molecules (one per species)
    changes       one row per reaction: how each species changes when the
                  reaction fires
    propensities  function of the state x, returning one rate per reaction
    t_grid        increasing times at which to record the state
    rng           a numpy random generator
    """
    changes = np.asarray(changes)
    x = np.array(x0, dtype=float)
    t = 0.0
    out = np.zeros((len(t_grid), len(x)))
    k = 0
    while k < len(t_grid):
        a = propensities(x)
        a0 = a.sum()
        # Time to the next reaction: exponential with mean 1/a0
        tau = rng.exponential(1 / a0) if a0 > 0 else np.inf
        # Record the current state at every grid time before the event
        while k < len(t_grid) and t_grid[k] < t + tau:
            out[k] = x
            k += 1
        t += tau
        if k < len(t_grid):
            # Which reaction: reaction j with probability a_j / a0
            j = np.searchsorted(np.cumsum(a), rng.random() * a0,
↪   side="right")
            x += changes[j]
    return out
```

6.2 Production and degradation of an mRNA

The simplest example is an mRNA made at a constant rate k and degraded at rate γ per molecule:

$$\emptyset \xrightarrow{k} \mathrm{M}, \qquad \mathrm{M} \xrightarrow{\gamma M} \emptyset.$$

The ODE model, $dM/dt = k - \gamma M$, rises smoothly to $M = k/\gamma$. The stochastic model has a steady state too, but it is a steady *distribution*: the number of molecules keeps fluctuating,

and for this model it follows a Poisson distribution with mean k/γ, whose variance is equal to its mean. We take $k = 10$ per minute and $\gamma = 1$ per minute, and run the SSA 2000 times.

MATLAB

Core MATLAB only.

```
%% Stochastic production and degradation of an mRNA
%     0 -> M     at rate k          (transcription)
%     M -> 0     at rate gamma * M  (degradation)
% The deterministic (ODE) model, dM/dt = k - gamma*M, predicts a smooth
% rise to M = k/gamma. With few molecules, each reaction is a random
% event, simulated here with Gillespie's algorithm. At steady state the
% number of molecules follows a Poisson distribution with mean k/gamma.
% Requires: core MATLAB only.

k = 10; gamma = 1;                  % per minute
t_grid = linspace(0, 10, 201);      % minutes
n_runs = 2000;
rng(1);

changes = [+1; -1];
propensities = @(x) [k, gamma * x(1)];

runs = zeros(n_runs, numel(t_grid));
for r = 1:n_runs
    runs(r, :) = sbp.gillespie(0, changes, propensities, t_grid)';
end
final = runs(:, end);
ode = k / gamma * (1 - exp(-gamma * t_grid));

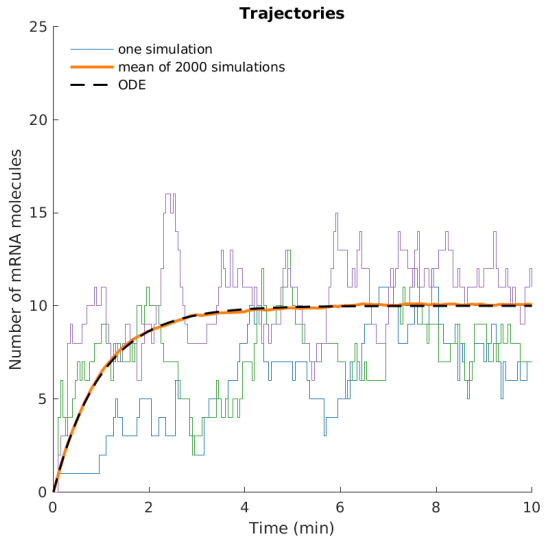
fprintf(['At t = %g min, over %d runs: mean %.3f, variance %.3f ' ...
    '(Poisson: both %g)\n'], t_grid(end), n_runs, mean(final), ...
    var(final, 1), k / gamma);

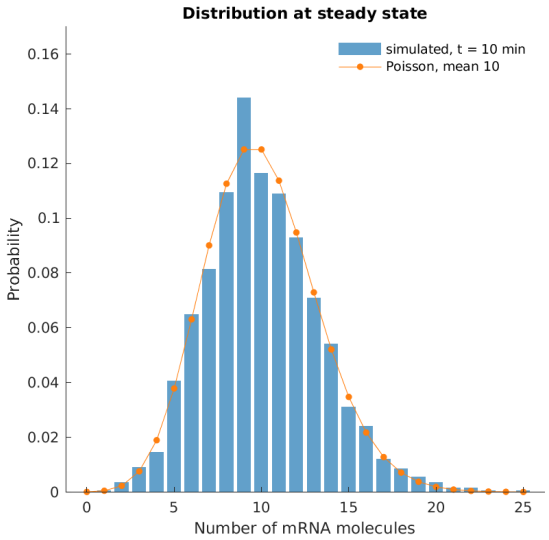
%% Plot
fig = figure('Position', [100 100 1000 400]);
subplot(1, 2, 1)
colors = [0.122 0.467 0.706; 0.173 0.627 0.173; 0.580 0.404 0.741];
for r = 1:3
    stairs(t_grid, runs(r, :), 'Color', colors(r, :));
    hold on
end
plot(t_grid, mean(runs, 1), '-', 'Color', [1.000 0.498 0.055], ...
    'LineWidth', 2);
plot(t_grid, ode, 'k--', 'LineWidth', 1.5);
```

```
ylim([0 25]);
xlabel('Time (min)'); ylabel('Number of mRNA molecules');
legend({'one simulation', '', '', ...
    sprintf('mean of %d simulations', n_runs), 'ODE'}, ...
    'Location', 'northwest', 'Box', 'off');
title('Trajectories');
box off

subplot(1, 2, 2)
m = 0:25;
counts = histcounts(final, -0.5:25.5) / n_runs;
bar(m, counts, 'FaceColor', [0.122 0.467 0.706], 'FaceAlpha', 0.7, ...
    'EdgeColor', 'none');
hold on
lambda = k / gamma;
plot(m, exp(-lambda) * lambda.^m ./ factorial(m), 'o-', ...
    'Color', [1.000 0.498 0.055], 'MarkerSize', 4, ...
    'MarkerFaceColor', [1.000 0.498 0.055]);
ylim([0 0.17]);
xlabel('Number of mRNA molecules'); ylabel('Probability');
legend({sprintf('simulated, t = %g min', t_grid(end)), ...
    sprintf('Poisson, mean %g', lambda)}, 'Box', 'off');
title('Distribution at steady state');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-birth-death');

results.steady_state = struct('mean', mean(final), ...
    'variance', var(final, 1));
mean_traj = mean(runs, 1);
results.mean_trajectory = mean_traj(1:20:end);
sbp.save_results(mfilename('fullpath'), results);
```





Python

Needs SciPy (for the Poisson distribution).

```
"""Stochastic production and degradation of an mRNA.

    0 -> M     at rate k          (transcription)
    M -> 0     at rate gamma * M  (degradation)

The deterministic (ODE) model, dM/dt = k - gamma*M, predicts a smooth
rise to M = k/gamma. With few molecules, each reaction is a random
event, simulated here with Gillespie's algorithm. At steady state the
number of molecules follows a Poisson distribution with mean k/gamma.
"""
import numpy as np
from scipy.stats import poisson
import matplotlib.pyplot as plt
import sbp
from sbp.ssa import gillespie

k, gamma = 10.0, 1.0              # per minute
t_grid = np.linspace(0, 10, 201)  # minutes
n_runs = 2000
rng = np.random.default_rng(1)

changes = [[+1], [-1]]


def propensities(x):
    return np.array([k, gamma * x[0]])
```

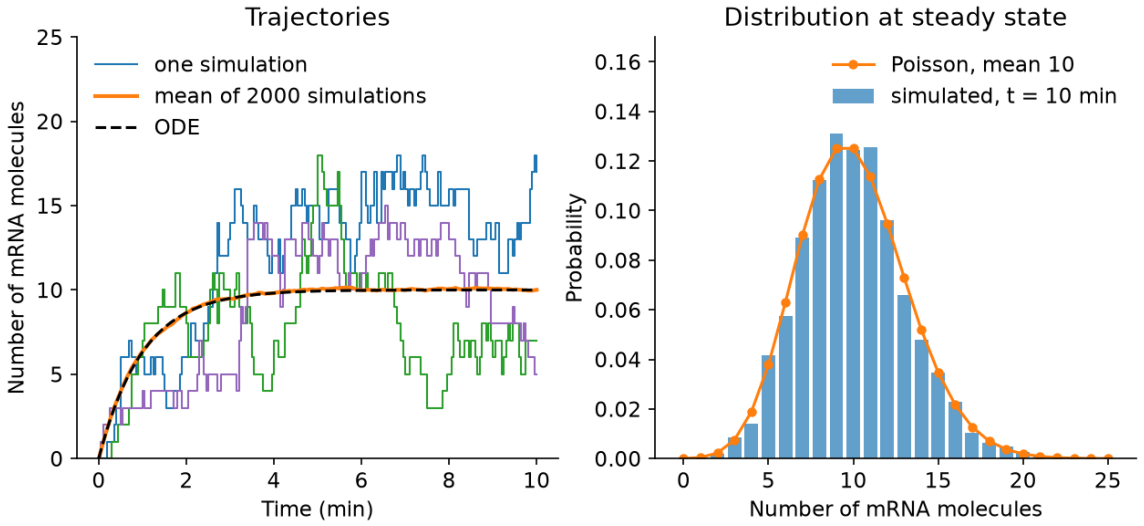
```
runs = np.array([gillespie([0], changes, propensities, t_grid, rng)[:, 0]
                 for _ in range(n_runs)])
final = runs[:, -1]
ode = k / gamma * (1 - np.exp(-gamma * t_grid))

print(f"At t = {t_grid[-1]:g} min, over {n_runs} runs: mean
 ↪  {final.mean():.3f}"
      f", variance {final.var():.3f} (Poisson: both {k / gamma:g})")

# --- Plot
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
colors = ["#1f77b4", "#2ca02c", "#9467bd"]
for r, col in zip(range(3), colors):
    ax1.step(t_grid, runs[r], where="post", color=col, lw=1,
             label="one simulation" if r == 0 else None)
ax1.plot(t_grid, runs.mean(axis=0), "-", color="#ff7f0e", lw=2,
         label=f"mean of {n_runs} simulations")
ax1.plot(t_grid, ode, "k--", lw=1.5, label="ODE")
ax1.set_xlabel("Time (min)")
ax1.set_ylabel("Number of mRNA molecules")
ax1.set_ylim(0, 25)
ax1.legend(frameon=False, loc="upper left")
ax1.set_title("Trajectories")

m = np.arange(0, 26)
counts = np.bincount(final.astype(int), minlength=26)[:26] / n_runs
ax2.bar(m, counts, color="#1f77b4", alpha=0.7,
        label=f"simulated, t = {t_grid[-1]:g} min")
ax2.plot(m, poisson.pmf(m, k / gamma), "o-", color="#ff7f0e", ms=4,
         label=f"Poisson, mean {k / gamma:g}")
ax2.set_xlabel("Number of mRNA molecules")
ax2.set_ylabel("Probability")
ax2.set_ylim(0, 0.17)
ax2.legend(frameon=False, loc="upper right")
ax2.set_title("Distribution at steady state")
sbp.save_figure(fig, __file__, "fig-birth-death")

sbp.save_results(__file__, {
    "steady_state": {"mean": final.mean(), "variance": final.var()},
    "mean_trajectory": runs.mean(axis=0)[::20],
})
sbp.show()
```



Individual runs (left) are jagged and differ from one another; at steady state, any one of them may hold anywhere from about 3 to 18 molecules. Their *average*, on the other hand, follows the ODE closely. For reactions such as these, where every propensity is at most linear in the numbers of molecules, the ODE gives the mean behaviour exactly; for nonlinear reactions, such as dimerisation, it is only an approximation. The distribution across runs (right) matches the Poisson distribution.

A useful way to describe the size of fluctuations is the **Fano factor**, the variance divided by the mean. It is 1 for a Poisson distribution, whatever the mean. Relative fluctuations, measured by the coefficient of variation $\sigma/\mu = 1/\sqrt{\mu}$, shrink as numbers grow: with 10 molecules they are about 32%, with 10,000 about 1%. That is why ODEs work well for abundant molecules and stochastic models matter for scarce ones.

6.3 Noise in gene expression

Proteins are made from mRNAs, and each mRNA is translated many times before it is degraded. Adding these two steps,

$$\emptyset \xrightarrow{k_m} \mathrm{M}, \qquad \mathrm{M} \xrightarrow{\gamma_m M} \emptyset, \qquad \mathrm{M} \xrightarrow{k_p M} \mathrm{M} + \mathrm{P}, \qquad \mathrm{P} \xrightarrow{\gamma_p P} \emptyset,$$

the mean number of proteins at steady state is $k_m k_p/\gamma_m\gamma_p$, and the Fano factor of the protein is approximately $1 + k_p/(\gamma_m + \gamma_p)$ (Thattai and Oudenaarden 2001). The ratio k_p/γ_m is the mean number of proteins made from one mRNA, the **burst size**. So two genes can make the same average amount of protein with very different noise: gene A is transcribed often, with small bursts of 2.5 proteins per mRNA, while gene B is transcribed ten times less often, with bursts of 25. Here we simulate each gene for 40,000 minutes and sample the protein number once a minute.

MATLAB

Core MATLAB only.

```
%% Noise in gene expression: transcription and translation
%     0 -> M       at rate km           (transcription)
%     M -> 0       at rate gm * M       (mRNA degradation)
%     M -> M + P   at rate kp * M       (translation)
%     P -> 0       at rate gp * P       (protein degradation)
% Two genes make the same mean number of proteins, 50. Gene A is
% transcribed often and each mRNA is translated a few times; gene B is
% transcribed rarely and each mRNA is translated many times, so its
% proteins are made in bursts. Theory (for example, Thattai and van
% Oudenaarden, 2001) gives, at steady state,
%     mean P = km*kp/(gm*gp),   Fano factor = var/mean = 1 + kp/(gm + gp).
% Requires: core MATLAB only.

gm = 1.0; gp = 0.1;                     % per minute
names = {'A', 'B'};
km = [2.0 0.2];                         % B: rare transcription,
kp = [2.5 25.0];                        %    big bursts
t_grid = 0:40000;                       % one long run, sampled each minute
burn_in = 200;                          % discard the approach to steady
 ↪ state
rng(1);

% Changes in (M, P) for each of the four reactions
changes = [1 0; -1 0; 0 1; 0 -1];

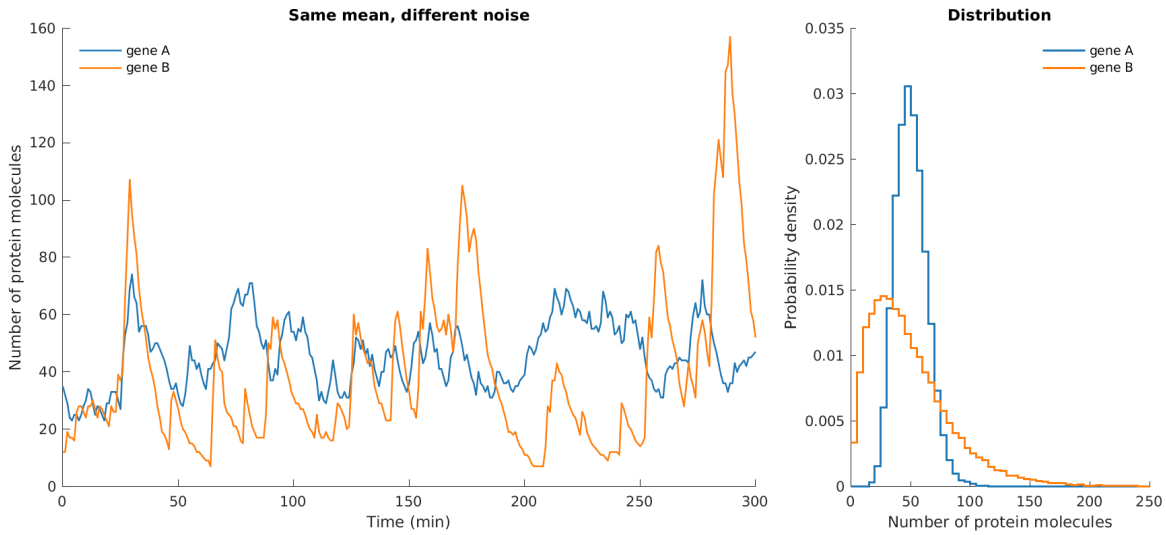
traces = zeros(numel(t_grid), 2);
for g = 1:2
    propensities = @(x) [km(g), gm * x(1), kp(g) * x(1), gp * x(2)];
    X = sbp.gillespie([0 0], changes, propensities, t_grid);
    traces(:, g) = X(:, 2);
    P = X(burn_in + 1:end, 2);
    results.(names{g}) = struct('mean', mean(P), ...
        'fano', var(P, 1) / mean(P));
    fprintf(['Gene %s: mean %.1f (theory %g), Fano factor %.2f ' ...
        '(theory %.2f)\n'], names{g}, mean(P), km(g) * kp(g) / (gm * gp),
         ↪ ...
        var(P, 1) / mean(P), 1 + kp(g) / (gm + gp));
end

%% Plot
colors = [0.122 0.467 0.706; 1.000 0.498 0.055];
fig = figure('Position', [100 100 1000 400]);
subplot(1, 3, [1 2])
window = 1001:1301;
```

```
for g = 1:2
    plot(t_grid(window) - t_grid(window(1)), traces(window, g), ...
        'Color', colors(g, :), 'LineWidth', 1.2);
    hold on
end
xlabel('Time (min)'); ylabel('Number of protein molecules');
title('Same mean, different noise');
legend({'gene A', 'gene B'}, 'Location', 'northwest', 'Box', 'off');
box off

subplot(1, 3, 3)
edges = 0:5:250;
for g = 1:2
    h = histcounts(traces(burn_in + 1:end, g), edges, ...
        'Normalization', 'pdf');
    stairs(edges, [h h(end)], 'Color', colors(g, :), 'LineWidth', 1.5);
    hold on
end
xlabel('Number of protein molecules'); ylabel('Probability density');
title('Distribution');
legend({'gene A', 'gene B'}, 'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-gene-expression');

sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs NumPy and Matplotlib.

```
"""Noise in gene expression: transcription and translation.

    0 -> M       at rate km           (transcription)
    M -> 0       at rate gm * M       (mRNA degradation)
    M -> M + P   at rate kp * M       (translation)
    P -> 0       at rate gp * P       (protein degradation)

Two genes make the same mean number of proteins, 50. Gene A is
transcribed often and each mRNA is translated a few times; gene B is
transcribed rarely and each mRNA is translated many times, so its
proteins are made in bursts. Theory (for example, Thattai and van
Oudenaarden, 2001) gives, at steady state,
    mean P = km*kp/(gm*gp),   Fano factor = var/mean = 1 + kp/(gm + gp).
"""
import numpy as np
import matplotlib.pyplot as plt
import sbp
from sbp.ssa import gillespie

gm, gp = 1.0, 0.1                          # per minute
genes = {"A": {"km": 2.0, "kp": 2.5},   # frequent transcription
         "B": {"km": 0.2, "kp": 25.0}}  # rare transcription, big bursts
t_grid = np.arange(0, 40001.0)             # one long run, sampled each minute
burn_in = 200                              # discard the approach to steady
 ↪ state
rng = np.random.default_rng(1)

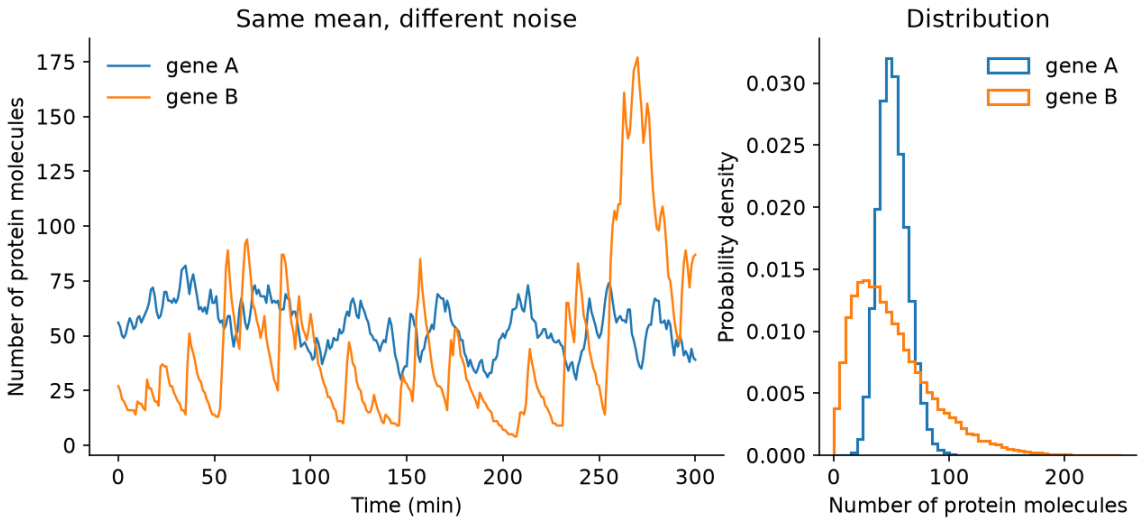
# Changes in (M, P) for each of the four reactions
changes = [[1, 0], [-1, 0], [0, 1], [0, -1]]

results = {}
traces = {}
for name, g in genes.items():
    def propensities(x, km=g["km"], kp=g["kp"]):
        M, P = x
        return np.array([km, gm * M, kp * M, gp * P])

    X = gillespie([0, 0], changes, propensities, t_grid, rng)
    P = X[burn_in:, 1]
    theory_mean = g["km"] * g["kp"] / (gm * gp)
    theory_fano = 1 + g["kp"] / (gm + gp)
    results[name] = {"mean": P.mean(), "fano": P.var() / P.mean()}
    traces[name] = X[:, 1]
    print(f"Gene {name}: mean {P.mean():.1f} (theory {theory_mean:g}), "
          f"Fano factor {P.var() / P.mean():.2f} (theory
           ↪ {theory_fano:.2f})")
```

```
# --- Plot
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4),
                               gridspec_kw={"width_ratios": [2, 1]})
window = slice(1000, 1301)
colors = {"A": "#1f77b4", "B": "#ff7f0e"}
for name in genes:
    ax1.plot(t_grid[window] - t_grid[window][0], traces[name][window],
             color=colors[name], lw=1.2, label=f"gene {name}")
    ax2.hist(traces[name][burn_in:], bins=np.arange(0, 251, 5),
             density=True, histtype="step", lw=1.5, color=colors[name],
             label=f"gene {name}")
ax1.set_xlabel("Time (min)")
ax1.set_ylabel("Number of protein molecules")
ax1.set_title("Same mean, different noise")
ax1.legend(frameon=False, loc="upper left")
ax2.set_xlabel("Number of protein molecules")
ax2.set_ylabel("Probability density")
ax2.set_title("Distribution")
ax2.legend(frameon=False)
sbp.save_figure(fig, __file__, "fig-gene-expression")

sbp.save_results(__file__, results)
sbp.show()
```



Both genes average about 50 proteins, but gene B jumps up in bursts, each time one of its rare mRNAs is translated many times over, and then decays; its Fano factor is about 24, against about 3 for gene A, and its distribution is wide and skewed. Ozbudak and colleagues found exactly this in *Bacillus subtilis*: increasing translation increased noise far more than

increasing transcription did (Ozbudak et al. 2002). Cells can therefore tune their noise, and not just their mean expression, through the balance of transcription and translation.

Since MATLAB and Python draw different random numbers, the two versions give slightly different estimates. The automated checks require both to be within 10% of theory for the mRNA example, and within 15% for the noisier protein estimates.

6.4 Exercises

1. In the mRNA example, start each run with 50 molecules in place of none. How does the mean approach the steady state? Compare with the solution of the ODE.
2. Add a dimerisation reaction, $2\mathrm{P} \to \mathrm{D}$, with propensity $k_d P(P-1)/2$, to the birth–death model for a protein P. Compare the mean of many SSA runs with the solution of the corresponding ODE for small and large numbers of molecules. When do they differ?
3. For the gene-expression model, vary k_p from 1 to 50 while adjusting k_m to keep the mean at 50 proteins. Plot the Fano factor against the burst size k_p/γ_m and compare with the theory.
4. Simulating every reaction becomes slow when molecules are abundant. Look up the **tau-leaping** method, which fires many reactions in each time step, and implement it for the mRNA example. How large can the time step be before the distribution is visibly wrong?

6.5 Going further

Noise is not always a nuisance. In circuits with two stable states, such as the toggle switch, noise can flip cells between states, letting a population hedge its bets. Exploring such circuits needs both the ODE tools of Chapter 3 and the stochastic tools of this chapter.

7 Constraint-based modelling

Kinetic models need rate laws and parameters that are rarely known for more than a handful of reactions. Constraint-based modelling sidesteps this. It asks only for the **stoichiometry** of a metabolic network, and a few **bounds** on the fluxes, and then asks: of all the flux distributions the network can support at steady state, which ones are possible, and which one is best for a given objective, such as growth?

This chapter works through flux balance analysis (FBA) and flux variability analysis (FVA), first on a small toy network that you can check by hand, and then on a real metabolic model of *E. coli*.

> **In the book**
>
> Chapter 8, *Introduction to Constraint-Based Modelling*: the stoichiometric matrix (§8.2), steady-state mass balance (§8.3), objective functions (§8.4), FBA as a linear program (§8.5), the worked toy example (§8.6), FVA (§8.7) and practical issues such as multiple optima and ATP maintenance (§8.8).

7.1 The toy network

We use the small network of the book's Fig. 8.1: eight metabolites, seven internal reactions, two exchange reactions that connect the cell to its surroundings, and a "biomass" reaction that drains the building blocks the cell needs to grow.

Table 7.1: The toy network. Fluxes are in mmol/gDW/h.

Reaction	Equation	Flux	Bounds
R1	$\mathrm{A} \rightarrow \mathrm{ATP} + \mathrm{B}$	v_1	$[0, 1000]$
R2	$\mathrm{A} \rightarrow \mathrm{C}$	v_2	$[0, 1000]$
R3	$\mathrm{C} \rightarrow \mathrm{ATP} + \mathrm{B}$	v_3	$[0, 1000]$
R4	$\mathrm{B} \rightleftharpoons 2\ \mathrm{D}$	v_4	$[-1000, 1000]$
R5	$\mathrm{B} \rightarrow \mathrm{F}$	v_5	$[0, 1000]$
R6	$\mathrm{D} \rightleftharpoons \mathrm{E}$	v_6	$[-1000, 1000]$
R7	$\mathrm{D} + \mathrm{ATP} \rightarrow \mathrm{G}$	v_7	$[0, 1000]$
E1	$\rightarrow \mathrm{A}$	b_1	$[0, 1000]$
E2	$\mathrm{E} \rightleftharpoons$	b_2	$[-1000, 1000]$
Biomass	$\mathrm{F} + 2\ \mathrm{G} + 10\ \mathrm{ATP} \rightarrow$	v_{bio}	$[0, 1000]$

Each reaction becomes a column of the **stoichiometric matrix S**, with one row per metabolite: negative entries for what a reaction consumes and positive entries for what it produces. At steady state, every metabolite is produced exactly as fast as it is consumed, so any feasible flux vector $\mathbf{v}$ satisfies

$$\mathbf{S}\mathbf{v} = \mathbf{0}.$$

With eight equations and ten unknowns, this system has infinitely many solutions. FBA picks one by optimising an objective, here the biomass flux.

7.2 Flux balance analysis

FBA is a **linear program** (LP):

$$\max_{\mathbf{v}} \ \mathbf{c}^{\top}\mathbf{v} \quad \text{subject to} \quad \mathbf{S}\mathbf{v} = \mathbf{0}, \qquad \mathbf{v}_{\text{lb}} \le \mathbf{v} \le \mathbf{v}_{\text{ub}},$$

where $\mathbf{c}$ picks out the objective; it has a 1 in the position of v_{bio} and zeros elsewhere. Any LP solver will do. Both `linprog` functions minimise, so we minimise $-\mathbf{c}^{\top}\mathbf{v}$ instead.

MATLAB

Needs the Optimization Toolbox (for `linprog`).

```
%% Flux balance analysis of a toy metabolic network
% The network is the one in Fig. 8.1 / Table 8.2 of the book: 8
 ↪  metabolites,
% 7 internal reactions, 2 exchange reactions and a 'biomass' reaction.
% FBA is solved directly as a linear program with linprog.
% Requires: Optimization Toolbox (for linprog).

%% The network
rxns = {'v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'b1', 'b2', 'vbio'};
mets = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'ATP'};

% Stoichiometric matrix: one row per metabolite, one column per reaction
%    v1  v2  v3  v4  v5  v6  v7  b1  b2 vbio
S = [-1  -1   0   0   0   0   0   1   0   0     % A
      1   0   1  -1  -1   0   0   0   0   0     % B
      0   1  -1   0   0   0   0   0   0   0     % C
      0   0   0   2   0  -1  -1   0   0   0     % D
      0   0   0   0   0   1   0   0  -1   0     % E
      0   0   0   0   1   0   0   0   0  -1     % F
      0   0   0   0   0   0   1   0   0  -2     % G
      1   0   1   0   0   0  -1   0   0 -10];   % ATP
```

```
% Bounds (mmol/gDW/h): reversible reactions v4, v6 and b2 may run backwards
lb = [0 0 0 -1000 0 -1000 0 0 -1000 0]';
ub = 1000 * ones(numel(rxns), 1);

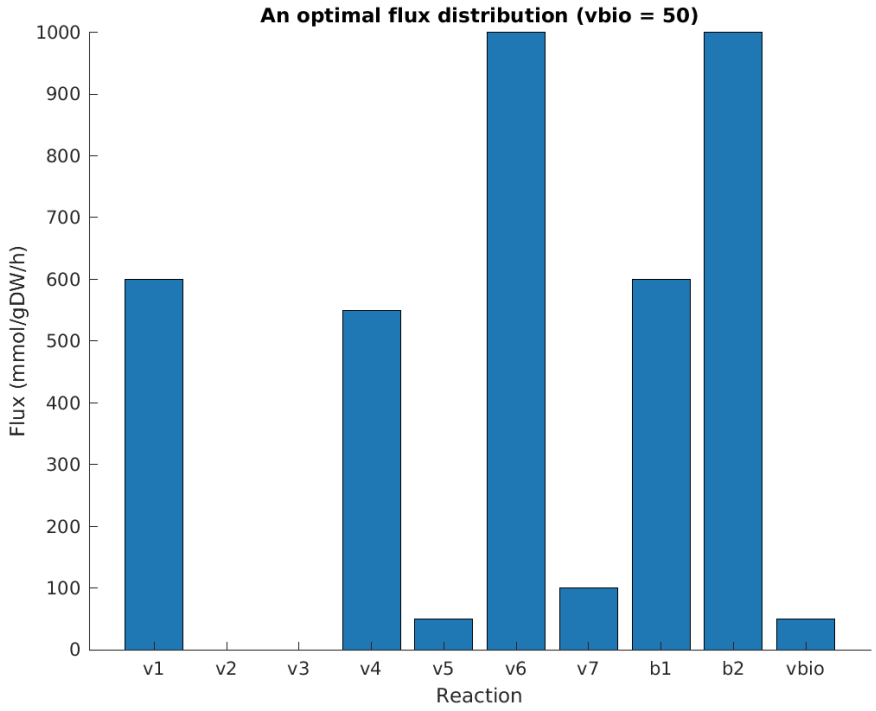
% Objective: maximise the biomass flux vbio
c = zeros(numel(rxns), 1);
c(strcmp(rxns, 'vbio')) = 1;

%% Solve: max c'v  s.t.  Sv = 0,  lb <= v <= ub
% linprog minimises, so we minimise -c'v
opts = optimoptions('linprog', 'Display', 'none');
beq = zeros(numel(mets), 1);
[v, fval, exitflag] = linprog(-c, [], [], S, beq, lb, ub, opts);
growth = -fval;

fprintf('Exit flag: %d (1 means optimal)\n', exitflag);
fprintf('Maximum biomass flux vbio = %g\n', growth);
for k = 1:numel(rxns)
    fprintf('  %4s = %8.1f\n', rxns{k}, v(k));
end
fprintf('Largest mass-balance violation |Sv| = %.1e\n', max(abs(S*v)));

%% Plot the flux distribution
fig = figure;
bar(categorical(rxns, rxns), v, 'FaceColor', [0.122 0.467 0.706]);
xlabel('Reaction');
ylabel('Flux (mmol/gDW/h)');
title(sprintf('An optimal flux distribution (vbio = %g)', growth));
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-toy-fba');

results.growth = growth;
results.fluxes = cell2struct(num2cell(v), rxns, 1);
results.max_mass_balance_error = max(abs(S*v));
sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs SciPy (for `linprog`).

```
"""Flux balance analysis of a toy metabolic network.

The network is the one in Fig. 8.1 / Table 8.2 of the book: 8 metabolites,
7 internal reactions, 2 exchange reactions and a 'biomass' reaction.
FBA is solved directly as a linear program with SciPy.
"""
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt
import sbp

# --- The network
rxns = ["v1", "v2", "v3", "v4", "v5", "v6", "v7", "b1", "b2", "vbio"]
mets = ["A", "B", "C", "D", "E", "F", "G", "ATP"]

# Stoichiometric matrix: one row per metabolite, one column per reaction
#             v1  v2  v3  v4  v5  v6  v7  b1  b2 vbio
S = np.array([[-1, -1,  0,  0,  0,  0,  0,  1,  0,   0],   # A
              [ 1,  0,  1, -1, -1,  0,  0,  0,  0,   0],   # B
              [ 0,  1, -1,  0,  0,  0,  0,  0,  0,   0],   # C
              [ 0,  0,  0,  2,  0, -1, -1,  0,  0,   0],   # D
              [ 0,  0,  0,  0,  0,  1,  0,  0, -1,   0],   # E
```

```
              [ 0,  0,  0,  0,  1,  0,  0,  0,  0, -1],   # F
              [ 0,  0,  0,  0,  0,  0,  1,  0,  0, -2],   # G
              [ 1,  0,  1,  0,  0,  0, -1,  0,  0, -10]])  # ATP

# Bounds (mmol/gDW/h): reversible reactions v4, v6 and b2 may run backwards
lb = np.array([0, 0, 0, -1000, 0, -1000, 0, 0, -1000, 0])
ub = np.full(len(rxns), 1000)

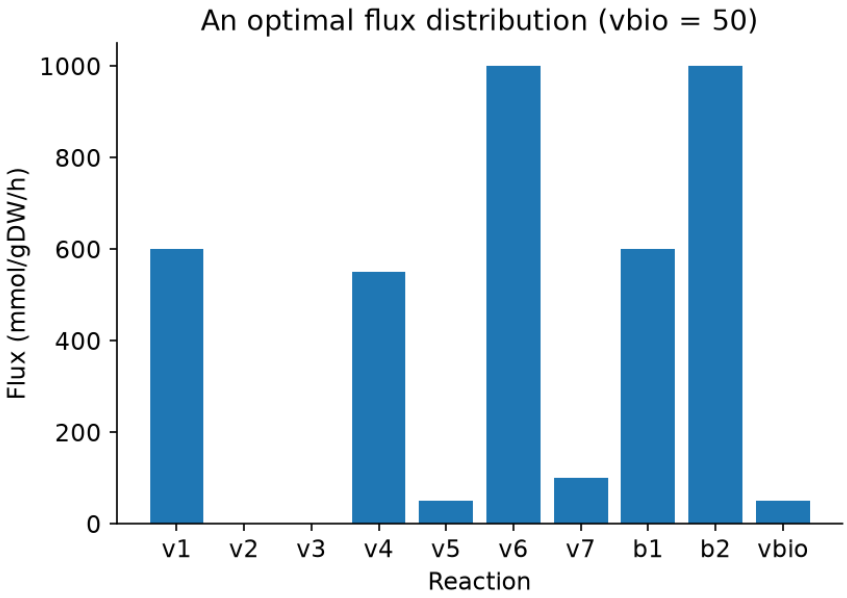
# Objective: maximise the biomass flux vbio
c = np.zeros(len(rxns))
c[rxns.index("vbio")] = 1

# --- Solve: max c'v  s.t.  Sv = 0,  lb <= v <= ub
# linprog minimises, so we minimise -c'v
res = linprog(-c, A_eq=S, b_eq=np.zeros(len(mets)),
              bounds=list(zip(lb, ub)), method="highs")
v = res.x + 0.0  # (+ 0.0 turns any -0.0 into 0.0 for printing)
growth = -res.fun

print(f"Status: {res.message}")
print(f"Maximum biomass flux vbio = {growth:g}")
for r, flux in zip(rxns, v):
    print(f"  {r:>4s} = {flux:8.1f}")
print(f"Largest mass-balance violation |Sv| = {np.abs(S @ v).max():.1e}")

# --- Plot the flux distribution
sbp.style()
fig, ax = plt.subplots()
ax.bar(rxns, v, color="#1f77b4")
ax.set_xlabel("Reaction")
ax.set_ylabel("Flux (mmol/gDW/h)")
ax.set_title(f"An optimal flux distribution (vbio = {growth:g})")
sbp.save_figure(fig, __file__, "fig-toy-fba")

sbp.save_results(__file__, {
    "growth": growth,
    "fluxes": dict(zip(rxns, v)),
    "max_mass_balance_error": np.abs(S @ v).max(),
})
sbp.show()
```



The maximum biomass flux is $v_{\text{bio}} = 50$, the same as in the book (Eq. 8.8). What limits it? Follow the stoichiometry: every unit of biomass needs 2 G, and so 2 units of flux through R7, and 10 ATP. Balancing ATP, B and D in turn shows that $v_4 = 11\, v_{\text{bio}}$ and $v_6 = 20\, v_{\text{bio}}$, so the upper bound of 1000 on v_6 (and on b_2, which must equal v_6) caps growth at 50.

> **i** The optimal growth rate is unique; the flux distribution is not
>
> R1 and the route R2 → R3 both turn A into ATP + B, so the cell can split the 600 units of A between them in any proportion and grow equally well. The book shows two such solutions (Eqs. 8.8 and 8.9). Which one you get depends on the solver, and the MATLAB and Python versions above may print different values of v_1, v_2 and v_3. Both are correct. This is why our automated checks compare only the growth rate for this example, and why the next section matters.

7.3 Flux variability analysis

FVA maps out *all* the optimal (or near-optimal) solutions at once. First solve the FBA problem to get the optimal objective f^*. Then, for each reaction j in turn, find the smallest and the largest flux it can carry while growth stays at least a fraction α of the optimum:

$$\min_{\mathbf{v}} / \max_{\mathbf{v}} \; v_j \quad \text{subject to} \quad \mathbf{S}\mathbf{v} = \mathbf{0}, \quad \mathbf{v}_{\text{lb}} \le \mathbf{v} \le \mathbf{v}_{\text{ub}}, \quad \mathbf{c}^{\top}\mathbf{v} \ge \alpha f^*.$$

That is two LPs per reaction. The code below runs FVA at $\alpha = 1$ (exactly optimal growth) and $\alpha = 0.9$.

MATLAB

Needs the Optimization Toolbox (for `linprog`).

```
%% Flux variability analysis (FVA) of the toy metabolic network
% FBA gives one optimal flux distribution, but often many exist. FVA finds,
% for every reaction, the smallest and largest flux compatible with
% (a chosen fraction of) the optimal growth rate.
% Requires: Optimization Toolbox (for linprog).

%% The network (same as toy_fba.m; book Fig. 8.1)
rxns = {'v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'b1', 'b2', 'vbio'};
%    v1  v2  v3  v4  v5  v6  v7  b1  b2 vbio
S = [-1  -1   0   0   0   0   0   1   0   0     % A
      1   0   1  -1  -1   0   0   0   0   0     % B
      0   1  -1   0   0   0   0   0   0   0     % C
      0   0   0   2   0  -1  -1   0   0   0     % D
      0   0   0   0   0   1   0   0  -1   0     % E
      0   0   0   0   1   0   0   0   0  -1     % F
      0   0   0   0   0   0   1   0   0  -2     % G
      1   0   1   0   0   0  -1   0   0 -10];   % ATP
lb = [0 0 0 -1000 0 -1000 0 0 -1000 0]';
ub = 1000 * ones(numel(rxns), 1);
n = numel(rxns);
c = zeros(n, 1);
c(strcmp(rxns, 'vbio')) = 1;
opts = optimoptions('linprog', 'Display', 'none');

%% Step 1: FBA, to find the optimal growth rate f*
[~, fval] = linprog(-c, [], [], S, zeros(size(S, 1), 1), lb, ub, opts);
f_star = -fval;

%% Step 2: FVA at 100% and at 90% of the optimum
[vmin100, vmax100] = fva(S, lb, ub, c, 1.0 * f_star, opts);
[vmin90,  vmax90]  = fva(S, lb, ub, c, 0.9 * f_star, opts);

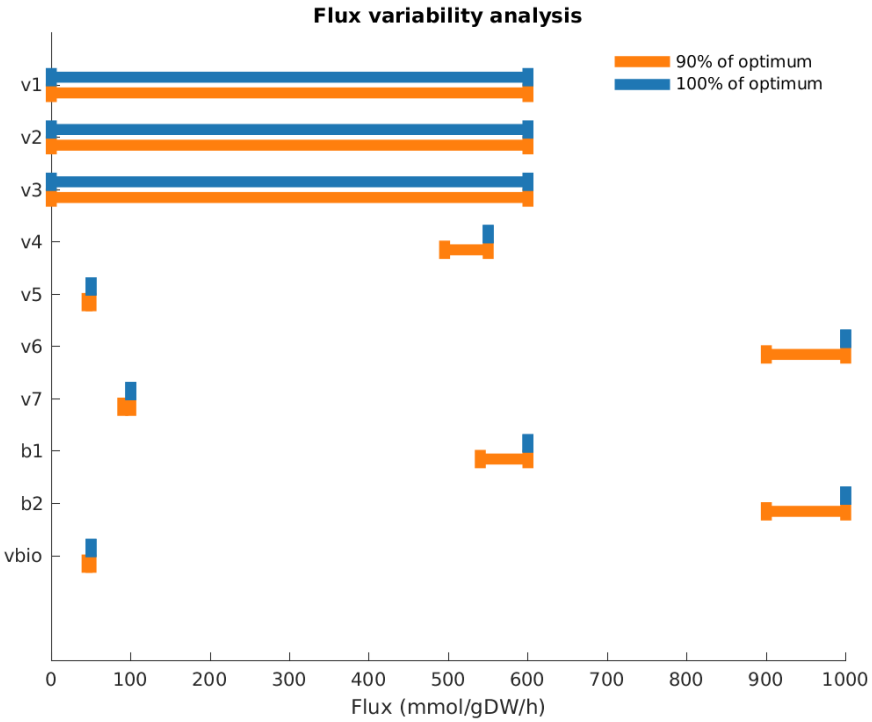
fprintf('Optimal growth f* = %g\n', f_star);
fprintf('%5s %8s %8s   (at 100%% of f*)\n', 'rxn', 'min', 'max');
for j = 1:n
    fprintf('%5s %8.1f %8.1f\n', rxns{j}, vmin100(j), vmax100(j));
end

%% Plot: flux ranges at 100% and 90% of optimal growth
blue = [0.122 0.467 0.706];
orange = [1.000 0.498 0.055];
fig = figure;
hold on
y = 1:n;
```

```
for j = 1:n
    plot([vmin90(j) vmax90(j)], y(j) + [0.15 0.15], '-|', 'Color', orange,
    ↪ ...
        'LineWidth', 6, 'MarkerSize', 10);
    plot([vmin100(j) vmax100(j)], y(j) - [0.15 0.15], '-|', 'Color', blue,
    ↪ ...
        'LineWidth', 6, 'MarkerSize', 10);
end
set(gca, 'YTick', y, 'YTickLabel', rxns, 'YDir', 'reverse');
xlabel('Flux (mmol/gDW/h)');
title('Flux variability analysis');
h = [plot(NaN, NaN, '-', 'Color', orange, 'LineWidth', 6), ...
     plot(NaN, NaN, '-', 'Color', blue, 'LineWidth', 6)];
legend(h, {'90% of optimum', '100% of optimum'}, ...
    'Location', 'northeast', 'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-toy-fva');

results.growth = f_star;
results.fva100.min = cell2struct(num2cell(vmin100), rxns, 1);
results.fva100.max = cell2struct(num2cell(vmax100), rxns, 1);
results.fva90.min = cell2struct(num2cell(vmin90), rxns, 1);
results.fva90.max = cell2struct(num2cell(vmax90), rxns, 1);
sbp.save_results(mfilename('fullpath'), results);

%% Local function
function [vmin, vmax] = fva(S, lb, ub, c, f_min, opts)
% Min and max of every flux, subject to growth c'v >= f_min.
% The extra constraint is written as -c'v <= -f_min.
n = numel(lb);
vmin = zeros(n, 1);
vmax = zeros(n, 1);
beq = zeros(size(S, 1), 1);
for j = 1:n
    e = zeros(n, 1);
    e(j) = 1;
    [~, vmin(j)] = linprog(e, -c', -f_min, S, beq, lb, ub, opts);
    [~, fmax] = linprog(-e, -c', -f_min, S, beq, lb, ub, opts);
    vmax(j) = -fmax;
end
end
```



Python

Needs SciPy (for `linprog`).

```
"""Flux variability analysis (FVA) of the toy metabolic network.

FBA gives one optimal flux distribution, but often many exist. FVA finds,
for every reaction, the smallest and largest flux compatible with
(a chosen fraction of) the optimal growth rate.
"""
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt
import sbp

# --- The network (same as toy_fba.py; book Fig. 8.1)
rxns = ["v1", "v2", "v3", "v4", "v5", "v6", "v7", "b1", "b2", "vbio"]
#             v1  v2  v3  v4  v5  v6  v7  b1  b2 vbio
S = np.array([[-1, -1,  0,  0,  0,  0,  0,  1,  0,   0],   # A
              [ 1,  0,  1, -1, -1,  0,  0,  0,  0,   0],   # B
              [ 0,  1, -1,  0,  0,  0,  0,  0,  0,   0],   # C
              [ 0,  0,  0,  2,  0, -1, -1,  0,  0,   0],   # D
              [ 0,  0,  0,  0,  0,  1,  0,  0, -1,   0],   # E
              [ 0,  0,  0,  0,  1,  0,  0,  0,  0,  -1],   # F
              [ 0,  0,  0,  0,  0,  0,  1,  0,  0,  -2],   # G
              [ 1,  0,  1,  0,  0,  0, -1,  0,  0, -10]])  # ATP
```

```
lb = np.array([0, 0, 0, -1000, 0, -1000, 0, 0, -1000, 0])
ub = np.full(len(rxns), 1000)
bounds = list(zip(lb, ub))
n = len(rxns)
c = np.zeros(n)
c[rxns.index("vbio")] = 1
zeros = np.zeros(S.shape[0])

# --- Step 1: FBA, to find the optimal growth rate f*
fba = linprog(-c, A_eq=S, b_eq=zeros, bounds=bounds, method="highs")
f_star = -fba.fun


def fva(fraction):
    """Min and max of every flux, subject to growth >= fraction * f*."""
    # The extra constraint c'v >= fraction*f*, written as -c'v <=
     ↪  -fraction*f*
    A_ub, b_ub = -c[np.newaxis, :], [-fraction * f_star]
    vmin, vmax = np.zeros(n), np.zeros(n)
    for j in range(n):
        e = np.zeros(n)
        e[j] = 1
        vmin[j] = linprog(e, A_ub=A_ub, b_ub=b_ub, A_eq=S, b_eq=zeros,
                          bounds=bounds, method="highs").fun
        vmax[j] = -linprog(-e, A_ub=A_ub, b_ub=b_ub, A_eq=S, b_eq=zeros,
                           bounds=bounds, method="highs").fun
    return vmin, vmax

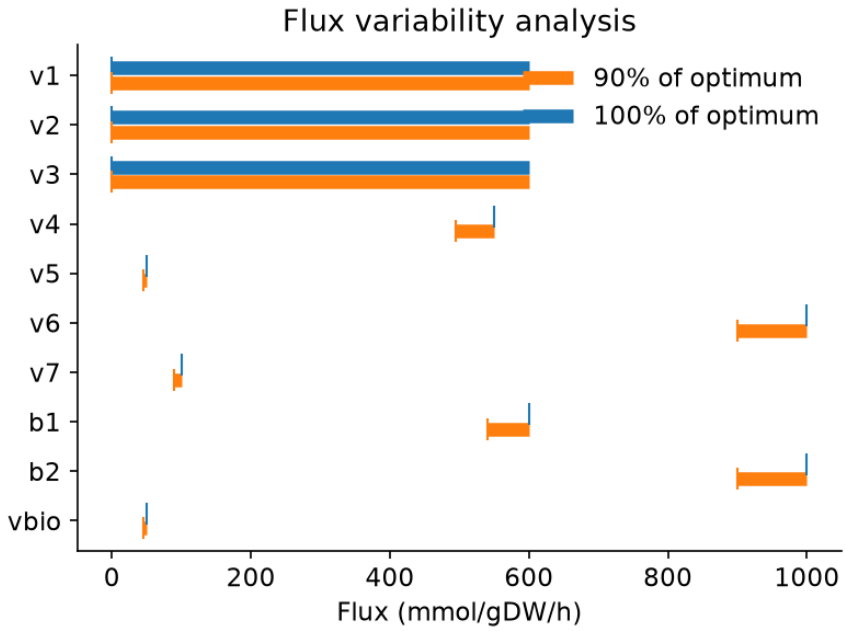

# --- Step 2: FVA at 100% and at 90% of the optimum
vmin100, vmax100 = fva(1.0)
vmin90, vmax90 = fva(0.9)

print(f"Optimal growth f* = {f_star:g}")
print(f"{'rxn':>5s} {'min':>8s} {'max':>8s}   (at 100% of f*)")
for r, lo, hi in zip(rxns, vmin100, vmax100):
    print(f"{r:>5s} {lo:8.1f} {hi:8.1f}")

# --- Plot: flux ranges at 100% and 90% of optimal growth
sbp.style()
fig, ax = plt.subplots()
y = np.arange(n)
ax.hlines(y + 0.15, vmin90, vmax90, color="#ff7f0e", lw=6,
          label="90% of optimum")
ax.hlines(y - 0.15, vmin100, vmax100, color="#1f77b4", lw=6,
          label="100% of optimum")
```

```
# Fixed fluxes have zero width, so mark both ends
ax.plot(vmin100, y - 0.15, "|", color="#1f77b4", ms=10)
ax.plot(vmin90, y + 0.15, "|", color="#ff7f0e", ms=10)
ax.set_yticks(y, rxns)
ax.invert_yaxis()
ax.set_xlabel("Flux (mmol/gDW/h)")
ax.set_title("Flux variability analysis")
ax.legend(loc="upper right", frameon=False)
sbp.save_figure(fig, __file__, "fig-toy-fva")

sbp.save_results(__file__, {
    "growth": f_star,
    "fva100": {"min": dict(zip(rxns, vmin100)),
               "max": dict(zip(rxns, vmax100))},
    "fva90": {"min": dict(zip(rxns, vmin90)),
              "max": dict(zip(rxns, vmax90))},
})
sbp.show()
```



At optimal growth, v_1, v_2 and v_3 can each take any value between 0 and 600, while every other flux is fixed. These are exactly the ranges in the book (Eq. 8.12). Relaxing growth to 90% of the optimum widens the other ranges too, because the reactions that limit growth no longer have to run at their maximum.

7.4 A genome-scale example: *E. coli* core

The same ideas carry over unchanged to real networks. The *E. coli* core model (Orth et al. 2010) has 95 reactions, 72 metabolites and 137 genes, covering glycolysis, the pentose phosphate pathway, the TCA cycle and oxidative phosphorylation. At this scale we no longer write $\mathbf{S}$ by hand. A standard model file (SBML) is read by a modelling package, COBRApy in Python and the COBRA Toolbox in MATLAB, which builds and solves the LP for us.

Two things to know when reading the code:

- **Exchange reactions** such as `EX_glc__D_e` (glucose) and `EX_o2_e` (oxygen) are written as *secretion*, so **uptake is a negative flux**. Allowing up to 10 mmol/gDW/h of glucose uptake therefore means setting the lower bound to -10.
- The model includes an **ATP maintenance** reaction, `ATPM`, with a lower bound of 8.39 mmol/gDW/h: ATP the cell must spend just to stay alive, whether it grows or not.

MATLAB

Needs the COBRA Toolbox; run `initCobraToolbox` once per session before this script.

```
%% FBA on the E. coli core metabolic model with the COBRA Toolbox
% 1. Predict growth with and without oxygen.
% 2. Vary the glucose uptake rate and watch growth respond.
% Requires: COBRA Toolbox (run initCobraToolbox once per session first).

% The .mat file is the same model as the SBML file the Python version reads
% (made by tools/make_mat_model.py); reading SBML in MATLAB needs libSBML.
model = readCbModel(fullfile(sbp.data_dir(), 'e_coli_core.mat'));
fprintf('%d reactions, %d metabolites, %d genes\n', ...
    numel(model.rxns), numel(model.mets), numel(model.genes));
fprintf('Objective: %s\n', strjoin(model.rxns(model.c ~= 0), ', '));

%% 1. Aerobic vs anaerobic growth
% Default medium: glucose uptake up to 10 mmol/gDW/h, oxygen unlimited.
% Uptake is a negative flux through an exchange reaction, hence lower
 ↪  bounds.
aerobic = optimizeCbModel(model, 'max');
fprintf('Aerobic growth rate:   %.4f /h\n', aerobic.f);

modelAnaerobic = changeRxnBounds(model, 'EX_o2_e', 0, 'l');  % no oxygen
 ↪  uptake
anaerobic = optimizeCbModel(modelAnaerobic, 'max');
fprintf('Anaerobic growth rate: %.4f /h\n', anaerobic.f);

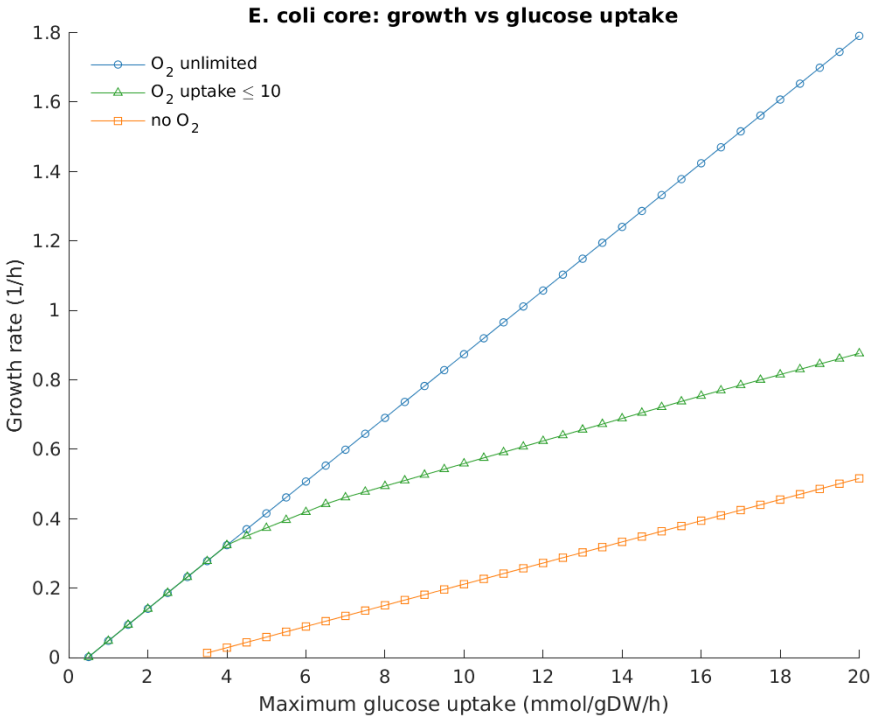
%% 2. Growth as a function of glucose uptake
```

```
uptake = 0:0.5:20;  % mmol/gDW/h
growthAer = growth_curve(model, 1000, uptake);  % O2 effectively unlimited
growthO2_10 = growth_curve(model, 10, uptake);  % O2 uptake at most 10
growthAnaer = growth_curve(model, 0, uptake);   % no O2

%% Plot
fig = figure;
plot(uptake, growthAer, 'o-', 'Color', [0.122 0.467 0.706], 'MarkerSize',
 ↪  4);
hold on
plot(uptake, growthO2_10, '^-', 'Color', [0.173 0.627 0.173], 'MarkerSize',
 ↪  4);
plot(uptake, growthAnaer, 's-', 'Color', [1.000 0.498 0.055], 'MarkerSize',
 ↪  4);
xlabel('Maximum glucose uptake (mmol/gDW/h)');
ylabel('Growth rate (1/h)');
title('E. coli core: growth vs glucose uptake');
legend({'O_2 unlimited', 'O_2 uptake \leq 10', 'no O_2'}, ...
    'Location', 'northwest', 'Box', 'off');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-ecoli-glucose');

results.n_reactions = numel(model.rxns);
results.growth_aerobic = aerobic.f;
results.growth_anaerobic = anaerobic.f;
results.glucose_uptake = uptake;
results.growth_vs_glucose_aerobic = growthAer;
results.growth_vs_glucose_o2_10 = growthO2_10;
results.growth_vs_glucose_anaerobic = growthAnaer;
sbp.save_results(mfilename('fullpath'), results);

%% Local function
function growth = growth_curve(model, o2UptakeLimit, uptake)
% Maximum growth for each glucose uptake limit; NaN where the LP is
 ↪  infeasible.
model = changeRxnBounds(model, 'EX_o2_e', -o2UptakeLimit, 'l');
growth = nan(size(uptake));
for k = 1:numel(uptake)
    m = changeRxnBounds(model, 'EX_glc__D_e', -uptake(k), 'l');
    sol = optimizeCbModel(m, 'max');
    if sol.stat == 1  % 1 means an optimal solution was found
        growth(k) = sol.f;
    end
end
end
```



Python

Needs COBRApy.

```
"""FBA on the E. coli core metabolic model with COBRApy.

1. Predict growth with and without oxygen.
2. Vary the glucose uptake rate and watch growth respond.
"""
import math

import numpy as np
import matplotlib.pyplot as plt
import cobra
import sbp

model = cobra.io.read_sbml_model(str(sbp.DATA / "e_coli_core.xml"))
print(f"{model.id}: {len(model.reactions)} reactions, "
      f"{len(model.metabolites)} metabolites, {len(model.genes)} genes")
objective = cobra.util.solver.linear_reaction_coefficients(model)
print("Objective:", ", ".join(r.id for r in objective))

# --- 1. Aerobic vs anaerobic growth
# Default medium: glucose uptake up to 10 mmol/gDW/h, oxygen unlimited.
# Uptake is a negative flux through an exchange reaction, hence lower
 ↪  bounds.
```

```
aerobic = model.optimize()
print(f"Aerobic growth rate:   {aerobic.objective_value:.4f} /h")

with model:  # changes inside 'with' are undone afterwards
    model.reactions.EX_o2_e.lower_bound = 0  # no oxygen uptake
    anaerobic = model.optimize()
print(f"Anaerobic growth rate: {anaerobic.objective_value:.4f} /h")

# --- 2. Growth as a function of glucose uptake
uptake = np.arange(0, 20.5, 0.5)  # mmol/gDW/h

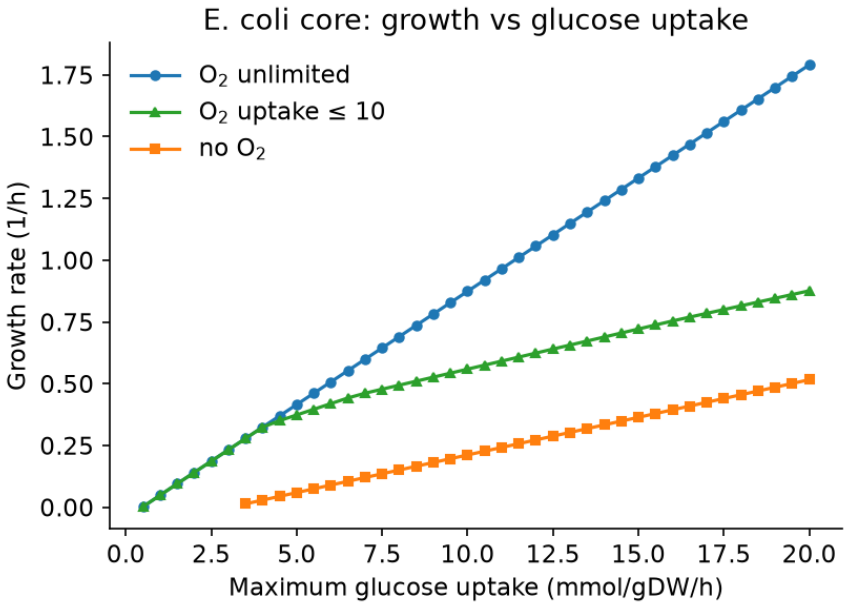

def growth_curve(o2_uptake_limit):
    """Maximum growth for each glucose uptake limit; NaN where
     ↪  infeasible."""
    growth = []
    with model:
        model.reactions.EX_o2_e.lower_bound = -o2_uptake_limit
        for u in uptake:
            model.reactions.EX_glc__D_e.lower_bound = -u
            # slim_optimize returns NaN when the LP is infeasible
            growth.append(model.slim_optimize(error_value=math.nan))
    return np.array(growth)


growth_aer = growth_curve(1000)    # oxygen effectively unlimited
growth_o2_10 = growth_curve(10)    # oxygen uptake at most 10 mmol/gDW/h
growth_anaer = growth_curve(0)     # no oxygen

# --- Plot
sbp.style()
fig, ax = plt.subplots()
ax.plot(uptake, growth_aer, "o-", color="#1f77b4", ms=4,
        label="O$_2$ unlimited")
ax.plot(uptake, growth_o2_10, "^-", color="#2ca02c", ms=4,
        label="O$_2$ uptake  10")
ax.plot(uptake, growth_anaer, "s-", color="#ff7f0e", ms=4,
        label="no O$_2$")
ax.set_xlabel("Maximum glucose uptake (mmol/gDW/h)")
ax.set_ylabel("Growth rate (1/h)")
ax.set_title("E. coli core: growth vs glucose uptake")
ax.legend(frameon=False)
sbp.save_figure(fig, __file__, "fig-ecoli-glucose")

sbp.save_results(__file__, {
    "n_reactions": len(model.reactions),
```

```
    "growth_aerobic": aerobic.objective_value,
    "growth_anaerobic": anaerobic.objective_value,
    "glucose_uptake": uptake,
    "growth_vs_glucose_aerobic": growth_aer,
    "growth_vs_glucose_o2_10": growth_o2_10,
    "growth_vs_glucose_anaerobic": growth_anaer,
})
sbp.show()
```



With glucose uptake capped at 10 mmol/gDW/h, the predicted growth rate is 0.874 h^{-1} with oxygen and 0.212 h^{-1} without it: aerobic respiration extracts far more ATP from each glucose than fermentation does. The growth curves show three more things.

- **At zero glucose the problem is infeasible**, not merely zero growth. With no carbon source, nothing can meet the ATP maintenance demand, so no flux vector satisfies all the constraints. The curves start only once glucose uptake covers maintenance, which takes more glucose without oxygen than with it.
- **When glucose is the only limit, growth rises in a straight line.** This is a general property of linear programs: scaling the one binding constraint scales the solution.
- **When oxygen is also limited** (the middle curve), the line bends at the point where oxygen runs out, and beyond it each extra unit of glucose buys less growth. A different constraint has become binding.

7.5 Exercises

1. In the toy network, raise the upper bound of v_6 to 2000. Does the optimal growth change? Why not? What happens if you also raise the upper bound of b_2?
2. Remove the reaction R2 from the toy network (set both its bounds to 0) and rerun FVA. What happens to the ranges of v_1 and v_3, and why?
3. Run FVA on the *E. coli* core model under aerobic conditions (`flux_variability_analysis` in COBRApy, `fluxVariability` in the COBRA Toolbox). Which reactions can carry no flux at all? (Book, §8.8.1.)
4. At glucose uptake 10 and oxygen uptake at most 10 mmol/gDW/h, which by-products does the model secrete? Are they unique? Use FVA on the exchange reactions to find out.

7.6 Going further

The next chapter builds on these tools to study how networks respond to perturbations: gene and reaction knock-outs, synthetic lethals and over-expression (book, Chapters 9 and 10).

8 Perturbations to metabolic networks

One of the most common uses of a metabolic model is to predict what happens when something is taken away: a gene knocked out, an enzyme inhibited by a drug. Which reactions can the cell not do without? Which pairs of reactions back each other up, so that either can be lost but not both? Constraint-based models answer such questions quickly, at the scale of whole genomes, and these predictions are used to find drug targets in pathogens and to design strains for metabolic engineering. This chapter builds on Chapter 7 and uses the same *E. coli* core model.

> **In the book**
>
> Chapter 10, *Perturbations to Metabolic Networks*: knock-outs (§10.1), gene versus reaction deletions and gene–protein–reaction rules (§10.1.1), synthetic lethals and algorithms to find them, including Fast-SL (§10.2), and applications such as drug target identification (§10.6). Chapter 9 covers methods that predict the flux distribution after a perturbation, such as MoMA (§9.2).

8.1 Single deletions

To delete a reaction from a constraint-based model, we fix its flux to zero by setting both its lower and upper bounds to zero (book, Eq. 10.2), and solve the FBA problem again. If the maximum growth rate falls below some small fraction of the wild type, 1% here, the deletion is called **lethal**, and the reaction **essential**.

Experiments delete genes, not reactions, and the two do not map one to one. Each reaction in the model carries a **gene–protein–reaction (GPR) rule**, a Boolean expression of the genes needed for it. An enzyme made of several subunits needs all of them (AND); isozymes can stand in for one another (OR). A gene deletion removes exactly those reactions whose rule becomes false. COBRApy and the COBRA Toolbox evaluate these rules for us.

We delete each of the 74 internal reactions of the model in turn (leaving out exchange reactions and the biomass reaction), and then each of its 137 genes, in the standard aerobic glucose medium.

MATLAB

Needs the COBRA Toolbox; run `initCobraToolbox` *once per session before this script.*

```
%% Single reaction and gene deletions in the E. coli core model
% A reaction is deleted by fixing its flux to zero (lower and upper bounds
% both 0; book Eq. 10.2); the maximum growth rate is then found by FBA. A
% deletion is called lethal if growth falls below 1% of the wild type.
% Genes are deleted through the gene-protein-reaction (GPR) rules, which
% the COBRA Toolbox evaluates for us: a reaction is lost only if the
% deletion makes its rule false, so isozymes can stand in for one another.
% Requires: COBRA Toolbox (run initCobraToolbox once per session first).

model = readCbModel(fullfile(sbp.data_dir(), 'e_coli_core.mat'));
sol = optimizeCbModel(model, 'max');
wt = sol.f;
% Delete internal reactions only: no exchange reactions, no biomass
rxns = sort(model.rxns(~startsWith(model.rxns, {'EX_', 'Biomass'})));

%% Reaction deletions
ratio = zeros(numel(rxns), 1);
for k = 1:numel(rxns)
    m = changeRxnBounds(model, rxns{k}, 0, 'b');
    sol = optimizeCbModel(m, 'max');
    if sol.stat == 1
        ratio(k) = sol.f / wt;
    end                          % infeasible: no growth, ratio stays 0
end
essential_rxns = rxns(ratio < 0.01);
n_reduced = sum(ratio >= 0.01 & ratio < 0.99);

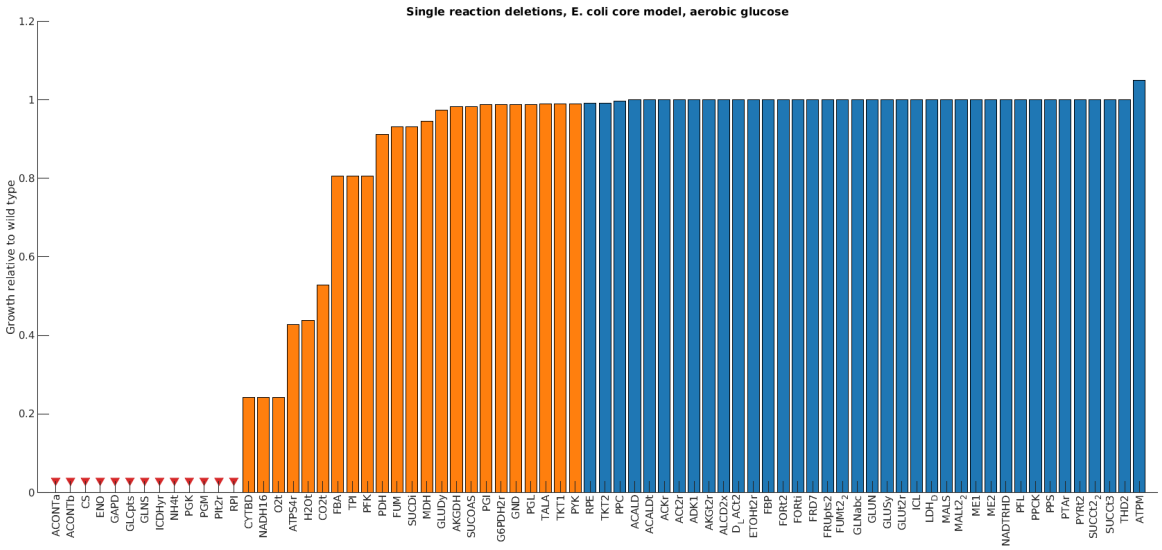
%% Gene deletions (the GPR rules decide which reactions are lost)
grRatio = singleGeneDeletion(model, 'FBA');
grRatio(isnan(grRatio)) = 0;   % infeasible: no growth
[essential_genes, idx] = sort(model.genes(grRatio < 0.01));
names = model.geneNames(grRatio < 0.01);
names = names(idx);

fprintf('Wild-type growth %.4f /h; %d reactions tested\n', wt,
 ↪  numel(rxns));
fprintf('%d essential reactions: %s\n', numel(essential_rxns), ...
    strjoin(essential_rxns, ', '));
fprintf('%d reactions whose loss reduces growth, but not below 1%%\n', ...
    n_reduced);
fprintf('%d essential genes, of %d: %s\n', numel(essential_genes), ...
    numel(model.genes), strjoin(strcat(essential_genes, ' (', names, ...
    ')'), ', '));

%% Plot: growth after each reaction deletion, sorted
[values, order] = sort(ratio);
```

```
colors = repmat([0.122 0.467 0.706], numel(values), 1);        % no effect
colors(values < 0.99, :) = repmat([1.000 0.498 0.055], ...
    sum(values < 0.99), 1);                                    % reduced
colors(values < 0.01, :) = repmat([0.839 0.153 0.157], ...
    sum(values < 0.01), 1);                                    % lethal
fig = figure('Position', [100 100 1000 400]);
b = bar(values, 'FaceColor', 'flat');
b.CData = colors;
hold on
lethal = find(values < 0.01);                % zero-height bars: mark them
plot(lethal, 0.03 * ones(size(lethal)), 'v', 'Color', [0.839 0.153 0.157],
 ↪  ...
    'MarkerFaceColor', [0.839 0.153 0.157], 'MarkerSize', 5);
set(gca, 'XTick', 1:numel(values), 'XTickLabel', rxns(order), ...
    'XTickLabelRotation', 90, 'FontSize', 7);
ylabel('Growth relative to wild type');
title('Single reaction deletions, E. coli core model, aerobic glucose');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-single-deletions');

results.wild_type_growth = wt;
results.growth_ratio = cell2struct(num2cell(ratio), rxns, 1);
results.essential_reactions = essential_rxns';
results.essential_genes = essential_genes';
sbp.save_results(mfilename('fullpath'), results);
```



Python

Needs COBRApy.

```python
"""Single reaction and gene deletions in the E. coli core model.

A reaction is deleted by fixing its flux to zero (lower and upper bounds
both 0; book Eq. 10.2); the maximum growth rate is then found by FBA. A
deletion is called lethal if growth falls below 1% of the wild type.
Genes are deleted through the gene-protein-reaction (GPR) rules, which
COBRApy evaluates for us: a reaction is lost only if the deletion makes
its rule false, so isozymes can stand in for one another.
"""
import numpy as np
import matplotlib.pyplot as plt
import cobra
from cobra.flux_analysis import single_gene_deletion
import sbp

model = cobra.io.read_sbml_model(str(sbp.DATA / "e_coli_core.xml"))
wt = model.slim_optimize()
cutoff = 0.01 * wt
# Delete internal reactions only: no exchange reactions, no biomass
rxns = sorted(r.id for r in model.reactions
              if not r.id.startswith(("EX_", "Biomass")))

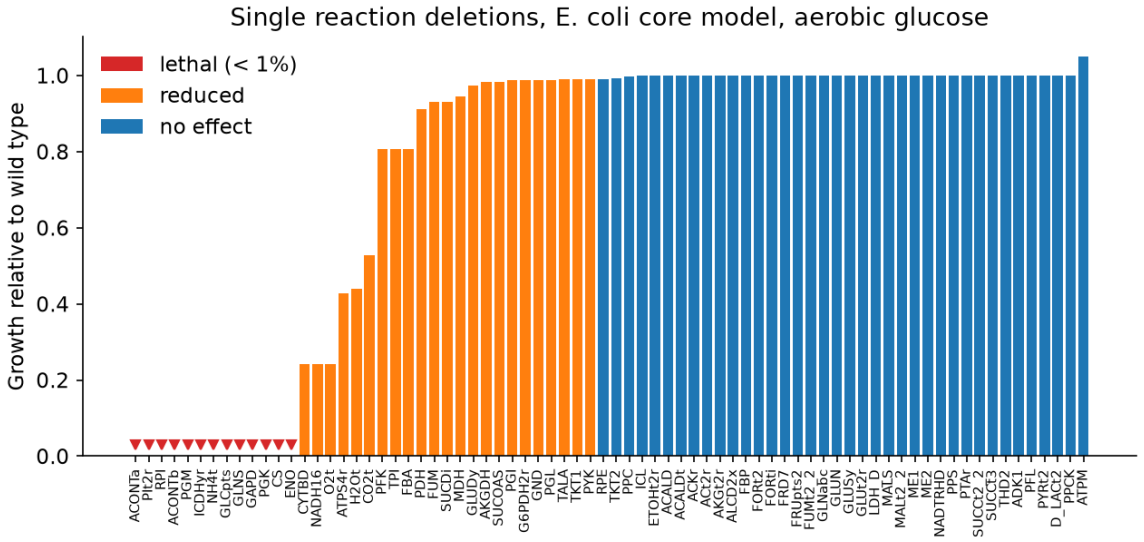
# --- Reaction deletions
ratio = {}
for rid in rxns:
    with model:                                   # undone after the block
        model.reactions.get_by_id(rid).bounds = (0, 0)
        ratio[rid] = model.slim_optimize(error_value=0.0) / wt
essential_rxns = [r for r in rxns if ratio[r] < 0.01]
reduced = [r for r in rxns if 0.01 <= ratio[r] < 0.99]

# --- Gene deletions (the GPR rules decide which reactions are lost)
gd = single_gene_deletion(model)
gene_ratio = {next(iter(ids)): (g if np.isfinite(g) else 0.0) / wt
              for ids, g in zip(gd["ids"], gd["growth"])}
essential_genes = sorted(g for g, r in gene_ratio.items() if r < 0.01)
names = [model.genes.get_by_id(g).name for g in essential_genes]

print(f"Wild-type growth {wt:.4f} /h; {len(rxns)} reactions tested")
print(f"{len(essential_rxns)} essential reactions: "
      f"{', '.join(essential_rxns)}")
print(f"{len(reduced)} reactions whose loss reduces growth, but not below
 ↪  1%")
print(f"{len(essential_genes)} essential genes, of {len(model.genes)}: "
      f"{', '.join(f'{g} ({n})' for g, n in zip(essential_genes, names))}")
```

```
# --- Plot: growth after each reaction deletion, sorted
order = sorted(rxns, key=lambda r: ratio[r])
values = np.array([ratio[r] for r in order])
colors = np.where(values < 0.01, "#d62728",
                  np.where(values < 0.99, "#ff7f0e", "#1f77b4"))
sbp.style()
fig, ax = plt.subplots(figsize=(10, 4))
ax.bar(range(len(order)), values, color=colors)
lethal = np.flatnonzero(values < 0.01)      # zero-height bars: mark them
ax.plot(lethal, np.full(lethal.size, 0.03), "v", color="#d62728", ms=5)
ax.set_xticks(range(len(order)), order, rotation=90, fontsize=7)
ax.set_ylabel("Growth relative to wild type")
ax.set_title("Single reaction deletions, E. coli core model, aerobic
 ↪  glucose")
for col, lab in [("#d62728", "lethal (< 1%)"), ("#ff7f0e", "reduced"),
                 ("#1f77b4", "no effect")]:
    ax.bar(0, 0, color=col, label=lab)
ax.legend(frameon=False, loc="upper left")
sbp.save_figure(fig, __file__, "fig-single-deletions")

sbp.save_results(__file__, {
    "wild_type_growth": wt,
    "growth_ratio": ratio,
    "essential_reactions": essential_rxns,
    "essential_genes": essential_genes,
})
sbp.show()
```



Thirteen reactions are essential (red), among them the glucose uptake system (GLCpts),

the lower steps of glycolysis (GAPD, PGK, PGM, ENO), the first steps of the TCA cycle (CS, ACONTa/b, ICDHyr), and the uptake of ammonium and phosphate. Losing another 23 reactions slows growth without stopping it (orange): the network reroutes flux, at a cost. Deleting the ATP maintenance reaction, ATPM, even *increases* growth slightly, since the ATP it consumes is freed for biosynthesis.

Only seven genes are essential, although thirteen reactions are. The difference comes from the GPR rules. Aconitase (ACONTa and ACONTb), for example, is encoded by two genes, *acnA* and *acnB*, either of which suffices, so neither gene is essential although the reaction is. In the other direction, *ptsH* and *ptsI* are each essential because the glucose transport system needs both. Predictions about genes, which are what experiments test, can therefore differ substantially from predictions about reactions.

8.2 Synthetic lethals

Two reactions form a **synthetic lethal** pair if deleting either one alone leaves the cell viable, but deleting both is lethal (book, §10.2). Such pairs reveal how a network backs itself up, and they are attractive targets for combination drugs.

The obvious way to find them is to test every pair of non-essential reactions. For the core model that is $\binom{61}{2} = 1830$ double deletions, which is easy; for a genome-scale model with 2000 reactions it is about two million, and for triplets more than a billion. The Fast-SL algorithm, developed in our group, cuts down the search with a simple argument (Pratapa et al. 2015). If some flux distribution that supports growth uses neither reaction of a pair, deleting both leaves that distribution intact, so the pair cannot be lethal. Hence:

- at least one reaction of a lethal pair, call it i, must carry flux in a wild-type solution; and
- the other reaction, j, must carry flux in a solution that remains once i is deleted.

The sparser these solutions, the fewer pairs need to be tested, so Fast-SL uses solutions that achieve the maximum growth with the minimum total flux, $\sum_j |v_j|$ (parsimonious FBA; book, §8.8.5). The code below finds all synthetic lethal pairs both ways, and checks that the answers agree.

MATLAB

Needs the COBRA Toolbox.

```
%% Synthetic lethal reaction pairs in the E. coli core model
% A pair of reactions is synthetic lethal if deleting both stops growth
% (below 1% of wild type) while deleting either one alone does not. We
% find all such pairs twice:
%   1. exhaustively, testing every pair of non-essential reactions;
%   2. as in Fast-SL (Pratapa et al., 2015), testing only pairs that can
%      possibly be lethal. If a growing flux distribution uses neither
```

```
%      reaction of a pair, deleting both leaves it intact, so the pair is
%      not lethal. Hence at least one reaction of a lethal pair carries
 ↪  flux
%      in the wild-type solution, and the other carries flux once the first
%      is deleted. Using sparse solutions (minimal total flux) keeps the
%      number of pairs to test small.
% Requires: COBRA Toolbox (run initCobraToolbox once per session first).

model = readCbModel(fullfile(sbp.data_dir(), 'e_coli_core.mat'));
sol = optimizeCbModel(model, 'max');
cutoff = 0.01 * sol.f;
rxns = sort(model.rxns(~startsWith(model.rxns, {'EX_', 'Biomass'})));

is_essential = false(size(rxns));
for k = 1:numel(rxns)
    is_essential(k) = growth_without(model, rxns(k)) < cutoff;
end
candidates = rxns(~is_essential);

%% 1. Exhaustive: every pair of non-essential reactions
pairs = nchoosek(1:numel(candidates), 2);
lethal_exhaustive = {};
for p = 1:size(pairs, 1)
    pair = candidates(pairs(p, :))';
    if growth_without(model, pair) < cutoff
        lethal_exhaustive(end + 1, :) = pair; %#ok<SAGROW>
    end
end

%% 2. Fast-SL: only pairs (i, j) with i active in the wild type and j
% active once i is deleted
tested = {};
lethal_fastsl = {};
for i = active_without(model, {}, candidates)'
    for j = active_without(model, i, candidates)'
        pair = sort([i, j]);
        key = strjoin(pair, ' ');
        if ~strcmp(i{1}, j{1}) && ~ismember(key, tested)
            tested{end + 1} = key; %#ok<SAGROW>
            if growth_without(model, pair) < cutoff
                lethal_fastsl(end + 1, :) = pair; %#ok<SAGROW>
            end
        end
    end
end
lethal_fastsl = sortrows(lethal_fastsl);
```

```
same = isequal(lethal_fastsl, sortrows(lethal_exhaustive));

fprintf('%d non-essential reactions\n', numel(candidates));
fprintf('Exhaustive: %d pairs tested, %d synthetic lethal\n', ...
    size(pairs, 1), size(lethal_exhaustive, 1));
fprintf('Fast-SL:    %d pairs tested, %d synthetic lethal\n', ...
    numel(tested), size(lethal_fastsl, 1));
fprintf('Same pairs found: %d\n', same);
partners = cellfun(@(r) sum(strcmp(lethal_fastsl(:), r)), candidates);
[~, order] = sortrows([-partners, (1:numel(candidates))']);
top = order(1:5);
fprintf('Reactions in the most pairs: %s\n', strjoin(strcat( ...
    candidates(top), ' (', arrayfun(@num2str, partners(top), ...
    'UniformOutput', false), ')'), ', '));

%% Plot: which reactions form lethal pairs
involved = unique(lethal_fastsl(:));
M = zeros(numel(involved));
for p = 1:size(lethal_fastsl, 1)
    a = strcmp(involved, lethal_fastsl{p, 1});
    b = strcmp(involved, lethal_fastsl{p, 2});
    M(a, b) = 1; M(b, a) = 1;
end
fig = figure('Position', [100 100 1100 500]);
subplot(1, 3, [1 2])
imagesc(M, [0 1]);
colormap([0.97 0.98 1; 0.12 0.47 0.71]);
axis square
set(gca, 'XTick', 1:numel(involved), 'XTickLabel', involved, ...
    'XTickLabelRotation', 90, 'YTick', 1:numel(involved), ...
    'YTickLabel', involved, 'FontSize', 7, 'Box', 'off');
title(sprintf('%d synthetic lethal pairs', size(lethal_fastsl, 1)));
subplot(1, 3, 3)
b = bar([size(pairs, 1), numel(tested)], 'FaceColor', 'flat');
b.CData = [0.6 0.6 0.6; 0.122 0.467 0.706];
set(gca, 'XTickLabel', {'exhaustive', 'Fast-SL'});
ylabel('Double deletions simulated');
title('Cost of the search');
box off
sbp.save_figure(fig, mfilename('fullpath'), 'fig-synthetic-lethals');

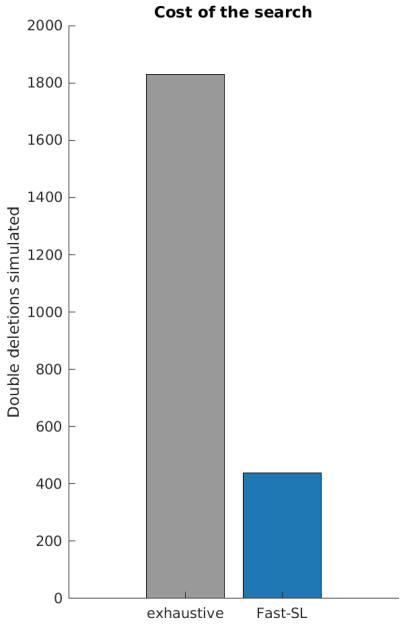
results.n_candidates = numel(candidates);
results.n_pairs_exhaustive = size(pairs, 1);
results.lethal_pairs = num2cell(lethal_fastsl, 2)';
results.same_as_exhaustive = same;
results.n_pairs_fastsl = numel(tested);
```

```
sbp.save_results(mfilename('fullpath'), results);

%% Local functions
function g = growth_without(model, deleted)
model = changeRxnBounds(model, deleted, 0, 'b');
sol = optimizeCbModel(model, 'max');
g = 0;
if sol.stat == 1
    g = sol.f;
end
end

function active = active_without(model, deleted, candidates)
% Candidate reactions carrying flux in a minimal-total-flux solution
if ~isempty(deleted)
    model = changeRxnBounds(model, deleted, 0, 'b');
end
sol = optimizeCbModel(model, 'max', 'one');
v = sol.x(findRxnIDs(model, candidates));
active = candidates(abs(v) > 1e-9);
end
```





Python

Needs COBRApy.

```
"""Synthetic lethal reaction pairs in the E. coli core model.

A pair of reactions is synthetic lethal if deleting both stops growth
(below 1% of wild type) while deleting either one alone does not. We
find all such pairs twice:
  1. exhaustively, testing every pair of non-essential reactions;
  2. as in Fast-SL (Pratapa et al., 2015), testing only pairs that can
     possibly be lethal. If a growing flux distribution uses neither
     reaction of a pair, deleting both leaves it intact, so the pair is
     not lethal. Hence at least one reaction of a lethal pair carries flux
     in the wild-type solution, and the other carries flux once the first
     is deleted. Using sparse solutions (minimal total flux, pFBA) keeps
     the number of pairs to test small.
"""
import itertools
import numpy as np
import matplotlib.pyplot as plt
import cobra
from cobra.flux_analysis import pfba
import sbp

model = cobra.io.read_sbml_model(str(sbp.DATA / "e_coli_core.xml"))
wt = model.slim_optimize()
cutoff = 0.01 * wt
rxns = sorted(r.id for r in model.reactions
              if not r.id.startswith(("EX_", "Biomass")))


def growth_without(deleted):
    with model:
        for rid in deleted:
            model.reactions.get_by_id(rid).bounds = (0, 0)
        return model.slim_optimize(error_value=0.0)


def active_without(deleted, candidates):
    """Candidate reactions carrying flux in a minimal-total-flux
     ↪ solution."""
    with model:
        for rid in deleted:
            model.reactions.get_by_id(rid).bounds = (0, 0)
        v = pfba(model).fluxes
    return [r for r in candidates if abs(v[r]) > 1e-9]


essential = [r for r in rxns if growth_without([r]) < cutoff]
```

```
candidates = [r for r in rxns if r not in essential]

# --- 1. Exhaustive: every pair of non-essential reactions
pairs = list(itertools.combinations(candidates, 2))
lethal_exhaustive = [p for p in pairs if growth_without(p) < cutoff]

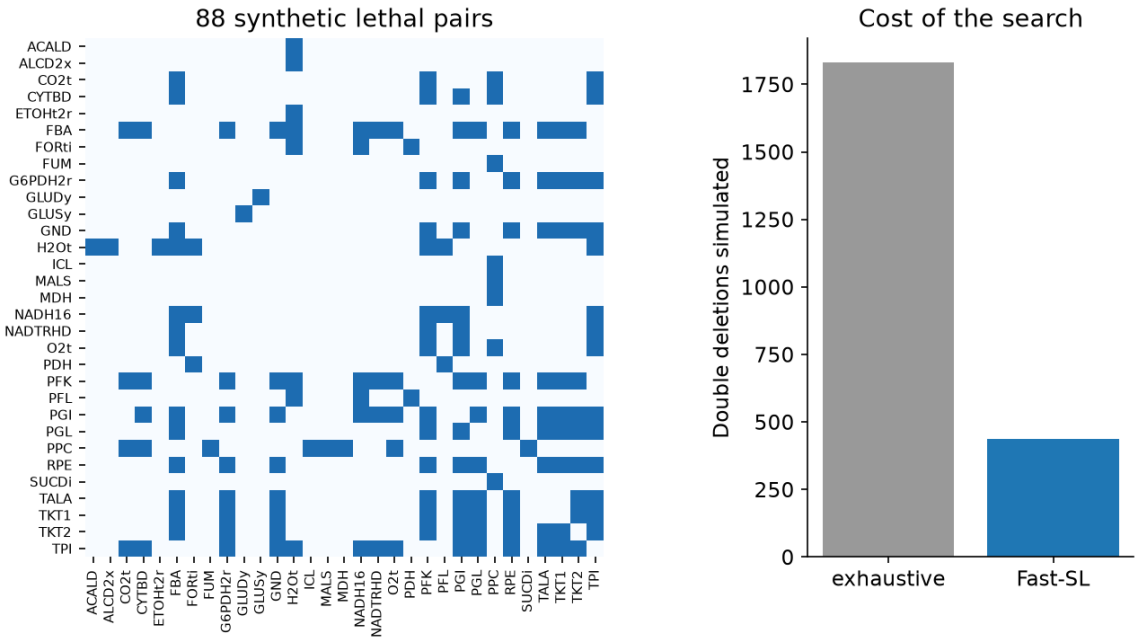
# --- 2. Fast-SL: only pairs (i, j) with i active in the wild type and j
# active once i is deleted
tested = set()
lethal_fastsl = set()
for i in active_without([], candidates):
    for j in active_without([i], candidates):
        pair = tuple(sorted((i, j)))
        if j != i and pair not in tested:
            tested.add(pair)
            if growth_without(pair) < cutoff:
                lethal_fastsl.add(pair)
lethal_fastsl = sorted(lethal_fastsl)

print(f"{len(candidates)} non-essential reactions")
print(f"Exhaustive: {len(pairs)} pairs tested, "
      f"{len(lethal_exhaustive)} synthetic lethal")
print(f"Fast-SL:    {len(tested)} pairs tested, "
      f"{len(lethal_fastsl)} synthetic lethal")
print(f"Same pairs found: {lethal_fastsl == sorted(lethal_exhaustive)}")
partners = {r: sum(r in p for p in lethal_fastsl) for r in candidates}
top = sorted(partners, key=lambda r: (-partners[r], r))[:5]
print("Reactions in the most pairs: "
      + ", ".join(f"{r} ({partners[r]})" for r in top))

# --- Plot: which reactions form lethal pairs
involved = sorted({r for p in lethal_fastsl for r in p})
idx = {r: k for k, r in enumerate(involved)}
M = np.zeros((len(involved), len(involved)))
for a, b in lethal_fastsl:
    M[idx[a], idx[b]] = M[idx[b], idx[a]] = 1
sbp.style()
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 5),
                               gridspec_kw={"width_ratios": [2.2, 1]})
ax1.imshow(M, cmap="Blues", vmin=0, vmax=1.3)
ax1.set_xticks(range(len(involved)), involved, rotation=90, fontsize=7)
ax1.set_yticks(range(len(involved)), involved, fontsize=7)
ax1.set_title(f"{len(lethal_fastsl)} synthetic lethal pairs")
for spine in ax1.spines.values():
    spine.set_visible(False)
ax2.bar(["exhaustive", "Fast-SL"], [len(pairs), len(tested)],
```

```
        color=["#999999", "#1f77b4"])
ax2.set_ylabel("Double deletions simulated")
ax2.set_title("Cost of the search")
sbp.save_figure(fig, __file__, "fig-synthetic-lethals")

sbp.save_results(__file__, {
    "n_candidates": len(candidates),
    "n_pairs_exhaustive": len(pairs),
    "lethal_pairs": [list(p) for p in lethal_fastsl],
    "same_as_exhaustive": lethal_fastsl == sorted(lethal_exhaustive),
    "n_pairs_fastsl": len(tested),
})
sbp.show()
```



Both searches find the same 88 synthetic lethal pairs, but Fast-SL simulates only about a quarter as many double deletions. The saving grows rapidly with the size of the network and with the order of the deletions, which is what makes searches for lethal triplets and quadruplets in genome-scale models feasible.

Many of the pairs make biological sense. The reactions of upper glycolysis (PGI, PFK, FBA, TPI) are each paired with reactions of the pentose phosphate pathway (G6PDH2r, PGL, GND, RPE, TALA, TKT1, TKT2): the two pathways are alternative routes from glucose 6-phosphate into lower glycolysis, and the cell can use either, but not neither. Glutamate can be made by glutamate dehydrogenase (GLUDy) or by glutamate synthase (GLUSy), and anaplerotic PEP carboxylase (PPC) is backed up by the glyoxylate shunt (ICL, MALS). Other pairs, such as those involving water transport (H2Ot), arise from the way the model

is written, and are a reminder that every prediction must be checked against biology (book, §10.7).

The exact number of double deletions that Fast-SL needs depends on which of several equally sparse solutions the solver returns, so it can differ slightly between MATLAB and Python; the set of lethal pairs cannot, and the automated checks require it to be identical.

8.3 Exercises

1. Repeat the single deletions for anaerobic growth (oxygen uptake set to zero). Which reactions become essential, and why?
2. Use the gene-deletion functions (`single_gene_deletion` in COBRApy, `singleGeneDeletion` in the COBRA Toolbox) to find essential genes, and the GPR rules to explain every case where a reaction is essential but none of its genes is.
3. Change the lethality cut-off from 1% to 5% and to 50% of wild-type growth. How do the lists of essential reactions and synthetic lethal pairs change?
4. Extend the Fast-SL code to find synthetic lethal *triplets*: three reactions whose joint deletion is lethal, when no pair among them is. How many triple deletions does it test, compared with the $\binom{n}{3}$ of an exhaustive search? (For genome-scale models, use the full implementations at https://github.com/RamanLab/FastSL and https://github.com/RamanLab/FastSL-py.)
5. A **minimal reactome** is a set of reactions that supports growth, from which no reaction can be removed without killing the cell. Starting from all reactions, remove reactions one at a time, in random order, keeping each removal only if the cell still grows. How large is the result, and does it change with the order? (This is the second problem of the BT5240 constraint-based assignment.)

8.4 Going further

Deletions are the simplest perturbations. Over-expression, enzyme inhibition and changes of medium can be modelled in the same framework, and methods such as MoMA and ROOM predict how fluxes redistribute after a perturbation, instead of assuming that the mutant immediately grows optimally (book, Chapters 9 and 10).

References

Barabási, Albert-László, and Réka Albert. 1999. "Emergence of Scaling in Random Networks." *Science* 286 (5439): 509–12. https://doi.org/10.1126/science.286.5439.509.

Fauré, Adrien, Aurélien Naldi, Claudine Chaouiya, and Denis Thieffry. 2006. "Dynamical Analysis of a Generic Boolean Model for the Control of the Mammalian Cell Cycle." *Bioinformatics* 22 (14): e124–31. https://doi.org/10.1093/bioinformatics/btl210.

Gillespie, Daniel T. 1977. "Exact Stochastic Simulation of Coupled Chemical Reactions." *The Journal of Physical Chemistry* 81 (25): 2340–61. https://doi.org/10.1021/j100540a008.

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.

Orth, Jeffrey D., Ronan M. T. Fleming, and Bernhard Ø. Palsson. 2010. "Reconstruction and Use of Microbial Metabolic Networks: The Core *Escherichia Coli* Metabolic Model as an Educational Guide." *EcoSal Plus* 4 (1). https://doi.org/10.1128/ecosalplus.10.2.1.

Ozbudak, Ertugrul M., Mukund Thattai, Iren Kurtser, Alan D. Grossman, and Alexander van Oudenaarden. 2002. "Regulation of Noise in the Expression of a Single Gene." *Nature Genetics* 31 (1): 69–73. https://doi.org/10.1038/ng869.

Pratapa, Aditya, Shankar Balachandran, and Karthik Raman. 2015. "Fast-SL: An Efficient Algorithm to Identify Synthetic Lethal Sets in Metabolic Networks." *Bioinformatics* 31 (20): 3299–305. https://doi.org/10.1093/bioinformatics/btv352.

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.

Segel, Lee A., and Marshall Slemrod. 1989. "The Quasi-Steady-State Assumption: A Case Study in Perturbation." *SIAM Review* 31 (3): 446–77. https://doi.org/10.1137/1031091.

Thattai, Mukund, and Alexander van Oudenaarden. 2001. "Intrinsic Noise in Gene Regulatory Networks." *Proceedings of the National Academy of Sciences* 98 (15): 8614–19. https://doi.org/10.1073/pnas.151588598.

Watts, Duncan J., and Steven H. Strogatz. 1998. "Collective Dynamics of 'Small-World' Networks." *Nature* 393: 440–42. https://doi.org/10.1038/30918.