

Chemical Kinetics 2302333
A Computational Approach with Python and SymPy

Viwat Vchirawongkwin

Contents

Preface	i
1 Introduction to Python and SymPy for Chemical Kinetics	1
1.1 Goal	1
1.2 Topics Covered	2
1.3 Overview of Python for Scientific Computing	2
1.3.1 Why Use Python for Scientific Computing?	2
1.4 Introduction to SymPy	3
1.4.1 Features of SymPy	3
1.5 Basic Chemical Kinetics Review	3
1.5.1 Importance of Chemical Kinetics	4
1.6 Basic Usage of Jupyter Notebook	4
1.6.1 Benefits of Jupyter Notebook for Chemical Kinetics	4
1.7 Hands-On Activities	5
1.7.1 Installing Python, Jupyter Notebook, and SymPy	5
1.7.2 Writing and Running Basic Python Code	5
1.7.3 Symbolic Computation of Simple Kinetic Equations	5
1.8 SymPy Function Tutorial	6
1.9 Assignment	12
2 Mathematical Formulation of Reaction Kinetics	13
2.1 Introduction and Goals	13
2.2 Fundamental Concepts in Reaction Kinetics	14
2.2.1 Reaction Rates and Rate Laws	14
2.2.2 Simple Reaction Orders	14
2.3 Multi-Step Reaction Mechanisms	15
2.3.1 Reversible Reactions	15
2.3.2 Consecutive and Parallel Reactions	15
2.3.3 Self-Catalyzed Reactions	16
2.4 Solving and Visualizing Kinetic Equations with Python	17
2.4.1 Symbolic Derivation with SymPy	17
2.4.2 Plotting Kinetic Curves	18
2.5 Modeling Reactions with Two Elementary Steps	19

2.5.1	Reversible Reactions Revisited	19
2.5.2	Reversible Reactions Revisited	20
2.5.3	Consecutive Reaction Mechanism	27
2.5.4	Parallel Reactions	34
2.5.5	Simplest Self-Catalyzed Reaction	39
2.6	Hands-On Activities and Assignment	42
2.6.1	Activity 1: Symbolic Derivation of Rate Laws	42
2.6.2	Activity 2: Numerical Integration of a Consecutive Reaction	42
2.6.3	Activity 3: Second-Order Reaction with Two Reactants	43
3	Multi-Step Reactions and Analytical Solutions	47
3.1	Introduction	47
3.2	Developing a Mathematical Model of a Reaction	47
3.3	Jupyter Notebook Example using SymPy	49
3.4	Matrix Method for Solving the Direct Problem	50
3.4.1	Example: A Consecutive-Competitive Reaction with a Reversible Step	51
3.4.2	Eigenvalue Decomposition Method	51
3.4.3	Eigenvector Ordering Considerations	52
3.4.4	Application and Numerical Evaluation	52
3.4.5	Example Calculation using Python	52
3.5	Approximate Methods of Chemical Kinetics	53
3.5.1	The Steady-State Concentration Method	53
3.6	Hands-On Activities	74
3.7	Assignment	76
3.8	Conclusion	76

Listings

1.1	Basic SymPy Example	3
1.2	Installing Jupyter Notebook	4
1.3	Running Jupyter Notebook	4
1.4	Installing Required Libraries	5
1.5	Defining a Function	5
1.6	Defining Symbols and Equations	5
1.7	Substituting Values into Equations	5
1.8	Solving a First-Order Reaction ODE	5
1.9	Enable pretty printing using <code>init_printing</code>	6
1.10	Defining symbolic variables using <code>symbols</code>	6
1.11	Defining a symbolic function $A(t)$ using <code>Function</code>	7
1.12	Creating a symbolic equation using <code>Eq</code>	7
1.13	Computing the derivative of $A(t)$ with respect to t	7
1.14	Solving the first-order ODE $A'(t) = -kA(t)$ using <code>dsolve</code> . . .	7
1.15	Solving the algebraic equation $2A = B$ for A using <code>solve</code> . . .	8
1.16	Computing the indefinite integral of $kA(t)$ with respect to t using <code>integrate</code>	8
1.17	Creating a 2x2 matrix using <code>Matrix</code>	8
1.18	Converting the symbolic function $A(t)$ into a numerical func- tion using <code>lambdify</code>	8
1.19	Substituting a numerical value into a symbolic expression us- ing <code>subs</code>	9
1.20	Plotting the function $A(t)$ over the range $t = 0$ to 10 using <code>plot</code> . .	9
1.21	Converting the derivative $\frac{d}{dt}A(t)$ to its LaTeX representation using <code>latex</code>	10
1.22	Simplifying, expanding, factorizing, and collecting terms in an expression.	10
1.23	Defining a piecewise function using <code>Piecewise</code>	11
1.24	Calculating the limit of $\frac{A(t)}{t}$ as t approaches 0 using <code>limit</code> . . .	11
1.25	Computing the Taylor series expansion of $A(t)$ around $t = 0$ using <code>series</code>	11
2.1	Python code for solving integrated rate equations for various reaction orders using <code>Sympy</code>	17

2.2	Plotting kinetic curves for different reaction orders using NumPy and Matplotlib	18
2.3	Python Code for Solving the Reversible Reaction ODE and Plotting Concentrations	24
2.4	Python Code for Second-Order Reversible Reaction	26
2.5	Symbolic Solution with SymPy	31
2.6	Numerical Integration and Visualization with SciPy	32
2.7	Numerical Simulation of a Complex Successive Reaction	33
2.8	Numerical Simulation of Parallel Reaction Kinetics	36
2.9	Numerical Simulation of a Second-Order Parallel Reaction	37
2.10	Python script for simulating the kinetics of a self-catalyzed reaction	40
3.1	Sympy code for computing concentration derivatives from a stoichiometric matrix and reaction rates	49
3.2	Sympy code for computing concentration evolution using matrix exponentials via eigen-decomposition of the rate constant matrix	52
3.3	Python simulation demonstrating the steady-state approximation.	54
3.4	Python code for applying the steady-state approximation to the Lindemann mechanism.	57
3.5	Symbolic Verification of the Steady-State Approximation in Ethane Thermal Cracking	63
3.6	Symbolic Evaluation of Ethane Concentration	65
3.7	Numerical Simulation of Kinetic Curves for Ethane Cracking	67
3.8	Quasi-equilibrium principle simulation using SciPy and Matplotlib	71
3.9	Derivation and visualization of Michaelis–Menten kinetics	73

Preface

This book introduces the application of Python and SymPy for modeling and solving problems in chemical kinetics. Designed for undergraduate students, it emphasizes symbolic computation and interactive coding to enhance understanding.

Chapter 1

Introduction to Python and SymPy for Chemical Kinetics

This chapter introduces Python programming as a tool for solving problems in chemical kinetics, with a focus on symbolic computation using the SymPy library. Additionally, we will explore Jupyter Notebook as an interactive computing environment that facilitates writing, executing, and visualizing computations in real time.

Understanding chemical kinetics is crucial for chemists, as it provides insights into reaction rates, mechanisms, and how various conditions affect chemical transformations. However, solving the mathematical equations involved in chemical kinetics can often be complex and tedious, especially for multi-step reactions or non-linear systems. By using computational tools like Python, we can:

- Automate repetitive calculations and equation solving.
- Symbolically manipulate and analyze kinetic models efficiently.
- Visualize reaction dynamics and interpret chemical behavior using computational methods.

In modern chemistry, computational approaches are not just helpful but essential. Industries and research laboratories worldwide use these techniques to design experiments, predict reaction outcomes, and develop new chemical processes. By learning Python and SymPy, you are equipping yourself with skills that bridge theoretical chemistry and real-world applications, preparing you for future challenges in both academia and industry.

1.1 Goal

The primary goal of this chapter is to introduce Python and SymPy as computational tools for solving chemical kinetics problems, with an emphasis

on symbolic computation and interactive modeling.

1.2 Topics Covered

- Overview of Python for Scientific Computing
- Introduction to SymPy
- Basic Chemical Kinetics Review
- Basic Usage of Jupyter Notebook

1.3 Overview of Python for Scientific Computing

Python is a versatile programming language widely used in scientific computing due to its ease of use, extensive libraries, and powerful computational capabilities. The most relevant libraries for chemical kinetics modeling include:

- **NumPy**: Provides support for numerical computations.
- **SymPy**: Enables symbolic computation, essential for solving equations analytically.
- **Matplotlib**: Used for visualization and plotting results.
- **SciPy**: Offers advanced scientific computing tools.

1.3.1 Why Use Python for Scientific Computing?

Python has become the preferred language for scientific computing due to its:

- Open-source nature, allowing free access and contributions from the community.
- Extensive ecosystem of scientific libraries and frameworks.
- Easy syntax, making it accessible to beginners and experts alike.
- Cross-platform compatibility, ensuring flexibility in different computing environments.
- Strong support for data visualization and analysis, crucial for interpreting chemical kinetics results.

1.4 Introduction to SymPy

SymPy is a Python library for symbolic mathematics. It allows users to define mathematical expressions, solve equations, and perform algebraic manipulations.

1.4.1 Features of SymPy

SymPy offers numerous features useful for chemical kinetics, including:

- Exact symbolic calculations instead of approximate numerical solutions.
- Simplification and factorization of complex algebraic expressions.
- Computation of derivatives and integrals relevant to reaction rate equations.
- Equation solving for analytical solutions of kinetic models.
- L^AT_EX output support for publishing high-quality equations.

```
1 from sympy import symbols, Eq, solve, pprint
2
3 # Define symbols
4 k, A = symbols('k_A')
5
6 # Define a simple rate equation
7 rate_eq = Eq(A - k*A, 1)
8 pprint(rate_eq)
9
10 # Solve for A
11 solution = solve(rate_eq, A)
12 pprint(solution)
```

Listing 1.1: Basic SymPy Example

1.5 Basic Chemical Kinetics Review

Chemical kinetics studies reaction rates and the factors affecting them. The general form of a rate law is:

$$\text{Rate} = k[A]^m[B]^n \quad (1.1)$$

where k is the rate constant, and m and n are reaction orders.

1.5.1 Importance of Chemical Kinetics

Understanding chemical kinetics is essential for:

- Predicting the speed of chemical reactions and their mechanisms.
- Designing chemical reactors and optimizing reaction conditions.
- Developing pharmaceuticals by studying drug decomposition rates.
- Controlling industrial processes, such as polymerization and catalysis.
- Environmental applications, such as modeling pollutant degradation.

1.6 Basic Usage of Jupyter Notebook

Jupyter Notebook is an interactive Python environment that allows users to write and execute code, visualize data, and document workflows in a single document. It is ideal for chemical kinetics computations.

1.6.1 Benefits of Jupyter Notebook for Chemical Kinetics

Jupyter Notebook is particularly useful because it:

- Provides an interactive coding environment for experimenting with kinetics models.
- Supports visualization tools to analyze reaction rate trends.
- Allows step-by-step documentation with Markdown and LaTeX integration.
- Enables easy sharing of computational work through Jupyter Notebook files (.ipynb).

To install Jupyter Notebook:

```
1 pip install jupyter
```

Listing 1.2: Installing Jupyter Notebook

To launch Jupyter Notebook:

```
1 jupyter notebook
```

Listing 1.3: Running Jupyter Notebook

1.7 Hands-On Activities

1.7.1 Installing Python, Jupyter Notebook, and SymPy

To get started, install Python and necessary libraries:

```
1 pip install sympy numpy matplotlib scipy
```

Listing 1.4: Installing Required Libraries

1.7.2 Writing and Running Basic Python Code

Example of defining and evaluating a simple function:

```
1 def reaction_rate(k, A):  
2     return k * A  
3  
4 print(reaction_rate(0.1, 2.0))
```

Listing 1.5: Defining a Function

Defining Symbols and Equations:

```
1 from sympy import symbols, Eq, pprint  
2  
3 # Define symbols  
4 A, A0, k, t = symbols('A_0 k t')  
5  
6 # Define a first-order rate law  
7 rate_law = Eq(A, A0 * symbols('e')**(-k * t))  
8 pprint(rate_law)
```

Listing 1.6: Defining Symbols and Equations

Substituting Values:

```
1 # Substitute values into the rate law  
2 A_initial = 1.0 # Initial concentration  
3 k_value = 0.2   # Rate constant  
4 t_value = 5.0   # Time  
5  
6 # Evaluate the equation  
7 result = rate_law.subs({A0: A_initial, k: k_value, t: t_value})  
8 pprint(result)
```

Listing 1.7: Substituting Values into Equations

1.7.3 Symbolic Computation of Simple Kinetic Equations

Using SymPy to solve a first-order reaction equation:

```
1 from sympy import symbols, Function, Eq, dsolve, Derivative,  
   pprint  
2
```

```

3 # Define the symbols and function
4 t, k, A0 = symbols('t k A0')
5 A = Function('A')
6
7 # Define the differential equation dA/dt = -k*A
8 rate_eq = Eq(Derivative(A(t), t), -k * A(t))
9 pprint(rate_eq)
10
11 # Solve the ODE with the initial condition A(0) = A0
12 solution = dsolve(rate_eq, A(t), ics={A(0): A0})
13 pprint(solution)

```

Listing 1.8: Solving a First-Order Reaction ODE

This solution represents the integrated rate law for a first-order reaction.

1.8 SymPy Function Tutorial

This section introduces essential SymPy functions used in solving Chemical Kinetics problems. Each function is explained with a brief description, example, and a caption for clarity.

0. init_printing

Purpose: Enable pretty printing for symbolic expressions in Jupyter notebooks.

```

1 from sympy import symbols, sin, init_printing
2
3 # Call init_printing to enable pretty printing
4 init_printing()
5
6 x = symbols('x')
7 expression = sin(x) / x
8
9 expression # In a Jupyter Notebook, this will be rendered as a
             nicely formatted math expression.

```

Listing 1.9: Enable pretty printing using init_printing.

1. symbols

Purpose: Define symbolic variables for equations.

```

1 from sympy import symbols
2
3 A, B, t, k = symbols('A B t k')
4 print(A, B, t, k) # Output: A B t k

```

Listing 1.10: Defining symbolic variables using symbols.

2. Function

Purpose: Define functions dependent on variables, e.g., $A(t)$.

```
1 from sympy import Function
2
3 A = Function('A')(t)
4 print(A) # Output: A(t)
```

Listing 1.11: Defining a symbolic function $A(t)$ using Function.

3. Eq

Purpose: Define symbolic equations.

```
1 from sympy import Eq, pprint
2
3 rate_law = Eq(A.diff(t), -k * A)
4 print(rate_law) # Output: Derivative(A, t) = -k*A
5 pprint(rate_law)
```

Listing 1.12: Creating a symbolic equation using Eq.

4. Derivative

Purpose: Compute the derivative of an expression.

```
1 derivative = A.diff(t)
2 print(derivative) # Output: Derivative(A, t)
3 pprint(derivative)
```

Listing 1.13: Computing the derivative of $A(t)$ with respect to t .

5. dsolve

Purpose: Solve differential equations symbolically.

```
1 from sympy import dsolve
2
3 solution = dsolve(Eq(A.diff(t), -k * A), A)
4 print(solution) # Output: A(t) = C1*exp(-k*t)
5 pprint(solution)
```

Listing 1.14: Solving the first-order ODE $A'(t) = -kA(t)$ using dsolve.

6. solve

Purpose: Solve algebraic equations.

```

1 from sympy import solve
2
3 result = solve(Eq(2 * A, B), A)
4 print(result) # Output: [B/2]

```

Listing 1.15: Solving the algebraic equation $2A = B$ for A using solve.

7. integrate

Purpose: Compute integrals (definite or indefinite).

```

1 from sympy import symbols, integrate, pprint, exp
2
3 # Define symbols: t is the upper limit; tau is the integration
  # variable.
4 tau, t, k, A0 = symbols('tau t k A0')
5
6 # Define the time-dependent concentration expression A(tau)
7 A_expr = A0 * exp(-k * tau)
8
9 # Compute the definite integral of k*A(tau) from tau=0 to tau=t
10 integral_definite = integrate(k * A_expr, (tau, 0, t))
11
12 pprint(integral_definite)

```

Listing 1.16: Computing the indefinite integral of $kA(t)$ with respect to t using integrate.

8. Matrix

Purpose: Define and manipulate matrices.

```

1 from sympy import Matrix
2
3 M = Matrix([[1, 2], [3, 4]])
4 print(M) # Output: Matrix([[1, 2], [3, 4]])
5 pprint(M)

```

Listing 1.17: Creating a 2x2 matrix using Matrix.

9. lambdify

Purpose: Convert symbolic expressions to numerical functions.

```

1 from sympy import symbols, Function, Eq, dsolve, Derivative, exp,
  lambdify, pprint
2
3 # Define the symbols and the function
4 t, k, A0 = symbols('t k A0')
5 A = Function('A')
6

```



```

7 # Define the differential equation: dA/dt = -k*A(t)
8 rate_eq = Eq(Derivative(A(t), t), -k * A(t))
9 pprint(rate_eq)
10
11 # Solve the ODE with the initial condition A(0) = A0
12 solution = dsolve(rate_eq, A(t), ics={A(0): A0})
13 pprint(solution)
14
15 # Extract the expression for A(t), which is A0*exp(-k*t)
16 A_expr = solution.rhs
17
18 # If you want to evaluate numerically, you need to substitute
19 # numerical values.
20 # For example, let A0 = 10 and k = 0.5.
21 A_expr_numeric = A_expr.subs({A0: 10, k: 0.5})
22
23 # Now lambdify the numerical expression with respect to t.
24 numerical_func = lambdify(t, A_expr_numeric, 'numpy')
25
26 # Evaluate A(t) at t = 5
27 result = numerical_func(5)
28 print("A(5) = ", result)

```

Listing 1.18: Converting the symbolic function $A(t)$ into a numerical function using `lambdify`.

10. subs

Purpose: Substitute values into expressions.

```

1 from sympy import symbols, Function
2
3 t, B = symbols('t B')
4 A = Function('A')
5
6 # Use A(t) instead of A
7 expression = A(t) + B
8
9 # To substitute, you must substitute the entire function call A(t)
10 substituted = expression.subs(A(t), 1)
11 print(substituted) # Output: 1 + B

```

Listing 1.19: Substituting a numerical value into a symbolic expression using `subs`.

11. plot

Purpose: Visualize symbolic expressions.

```

1 from sympy import symbols, Function, Eq, dsolve, Derivative, exp
2 from sympy.plotting import plot
3 from sympy import pprint

```

```

4
5 # Define the symbols and the function
6 t, k, A0 = symbols('t k A0')
7 A = Function('A')
8
9 # Define the ODE: dA/dt = -k * A(t)
10 rate_eq = Eq(Derivative(A(t), t), -k * A(t))
11 pprint(rate_eq)
12
13 # Solve the ODE with an initial condition A(0) = A0
14 solution = dsolve(rate_eq, A(t), ics={A(0): A0})
15 pprint(solution)
16
17 # Extract the right-hand side expression: A(t) = A0*exp(-k*t)
18 A_expr = solution.rhs
19
20 # Substitute numerical values (for example: A0 = 1, k = 0.2)
21 A_expr_numeric = A_expr.subs({A0: 1, k: 0.2})
22
23 # Plot the numerical expression over t from 0 to 10
24 plot(A_expr_numeric, (t, 0, 10))

```

Listing 1.20: Plotting the function $A(t)$ over the range $t = 0$ to 10 using `plot`.

12. L^AT_EX

Purpose: Convert expressions into L^AT_EX format.

```

1 from sympy import symbols, Function, latex
2
3 # Define the symbol for time
4 t = symbols('t')
5
6 # Define A as a function of t
7 A = Function('A')(t)
8
9 pprint(A.diff(t))
10
11 # Get the LaTeX representation of the derivative dA/dt
12 latex_expr = latex(A.diff(t))
13 print(latex_expr)

```

Listing 1.21: Converting the derivative $\frac{d}{dt}A(t)$ to its LaTeX representation using `latex`.

13. simplify, expand, factor, collect

Purpose: Simplify and manipulate expressions.

```

1 from sympy import simplify, expand, factor, collect, pprint
2

```

```

3 expr = (A + B)**2
4 pprint(simplify(expr)) # Simplify expression
5 pprint(expand(expr))   # Expand expression
6 pprint(factor(expr))    # Factorize expression

```

Listing 1.22: Simplifying, expanding, factorizing, and collecting terms in an expression.

14. Piecewise

Purpose: Define piecewise functions.

```

1 from sympy import Piecewise
2
3 piecewise_expr = Piecewise((A, A > 0), (0, True))
4 print(piecewise_expr)
5 pprint(piecewise_expr) # Output: Piecewise((A, A > 0), (0, True))

```

Listing 1.23: Defining a piecewise function using Piecewise.

15. limit

Purpose: Calculate limits of expressions.

```

1 from sympy import limit
2
3 limit_value = limit(A / t, t, 0)
4 print(limit_value) # Output: limit(A/t, t, 0)

```

Listing 1.24: Calculating the limit of $\frac{A(t)}{t}$ as t approaches 0 using limit.

16. series

Purpose: Compute Taylor or Laurent series expansions.

```

1 from sympy import symbols, Function, Eq, dsolve, Derivative,
  series, pprint
2
3 # Define the symbols and the function
4 t, k, A0 = symbols('t k A0')
5 A = Function('A')
6
7 # Define the differential equation dA/dt = -k*A(t)
8 rate_eq = Eq(Derivative(A(t), t), -k * A(t))
9
10 # Solve the ODE with the initial condition A(0) = A0
11 solution = dsolve(rate_eq, A(t), ics={A(0): A0})
12 print("Solution of the ODE:")
13 pprint(solution)
14
15 # Extract the expression for A(t)

```

```

16 A_expr = solution.rhs
17
18 # Compute the Taylor series expansion of A(t) about t = 0 up to
19   order 3 (i.e., including terms up to t^2)
20 taylor_series = series(A_expr, t, 0, 3)
21 print("\nTaylor series expansion of A(t) up to order 2:")
22 pprint(taylor_series)

```

Listing 1.25: Computing the Taylor series expansion of $A(t)$ around $t = 0$ using series.

1.9 Assignment

Use SymPy to solve the following problems and submit your Python code along with the solutions:

1. Derive the integrated rate law for a zero-order reaction symbolically.
2. Solve the second-order rate equation:

$$\frac{d[A]}{dt} = -k[A]^2 \quad (1.2)$$

Derive and express $[A]$ as a function of time t .

Chapter 2

Mathematical Formulation of Reaction Kinetics

2.1 Introduction and Goals

Chemical kinetics is the study of reaction rates and the mechanisms underlying chemical transformations. A solid mathematical foundation is crucial for analyzing experimental data, predicting reaction behavior, and deriving kinetic models that describe real chemical systems.

In this chapter, we aim to:

- Develop a quantitative understanding of reaction rates and how they are measured.
- Derive rate laws based on experimental observations and theoretical principles.
- Classify reactions by their kinetic order and determine the implications for reaction dynamics.
- Explore reaction mechanisms, including single-step, multi-step, reversible, consecutive, and parallel reactions.
- Investigate self-catalyzed reactions and their unique kinetic characteristics.
- Utilize computational tools such as `SymPy` to symbolically derive and manipulate rate equations.
- Apply Python's `sympy.diff()` and `sympy.integrate()` functions to solve differential equations related to reaction kinetics.
- Implement numerical integration methods to simulate kinetic behavior and validate models against experimental data.

By mastering these concepts, you will be equipped with the analytical and computational skills needed to tackle complex kinetic problems in chemistry.

2.2 Fundamental Concepts in Reaction Kinetics

2.2.1 Reaction Rates and Rate Laws

The reaction rate quantifies how fast reactants are transformed into products. A typical rate law has the form:

$$\text{Rate} = k[A]^m[B]^n, \quad (2.1)$$

where:

- k is the rate constant.
- $[A]$ and $[B]$ are the concentrations of the reactants.
- m and n denote the order of the reaction with respect to each reactant.

Rate laws are determined experimentally and provide insights into the underlying reaction mechanism.

2.2.2 Simple Reaction Orders

First-Order Reaction

A first-order reaction follows the differential equation:

$$\frac{d[A]}{dt} = -k[A], \quad (2.2)$$

with the solution (given the initial concentration $[A]_0$):

$$[A](t) = [A]_0 e^{-kt}. \quad (2.3)$$

Second-Order Reaction

For a second-order reaction involving two reactants A and B (assuming equal initial concentrations), the rate law is:

$$\frac{d[A]}{dt} = -k[A][B], \quad (2.4)$$

which, upon integration, gives:

$$\frac{1}{[A]} - \frac{1}{[A]_0} = kt. \quad (2.5)$$

2.3 Multi-Step Reaction Mechanisms

Many chemical reactions proceed via more than one elementary step. In this section, we consolidate the discussion on reversible, consecutive, and parallel reactions into a single framework.

2.3.1 Reversible Reactions

Reversible reactions occur in two opposing steps:



For first-order reversible steps, the net rate is:

$$r(t) = k_1 C_A(t) - k_2 C_B(t). \quad (2.7)$$

To analyze the reaction progress, we introduce a helper function $x(t)$ representing the amount of A that has reacted:

$$\frac{dx(t)}{dt} = k_1 [C_{A0} - x(t)] - k_2 [C_{B0} + x(t)], \quad (2.8)$$

with the initial condition $x(0) = 0$. Integration yields:

$$x(t) = \frac{k_1 C_{A0} + k_2 C_{B0}}{k_1 + k_2} [1 - e^{-(k_1 + k_2)t}]. \quad (2.9)$$

Substituting $x(t)$ into the concentration expressions for A and B gives a full description of how the system approaches equilibrium.

2.3.2 Consecutive and Parallel Reactions

Consecutive Reactions

In a consecutive reaction, a reactant A converts to an intermediate B , which then forms the final product C :



The corresponding rate equations are:

$$\frac{d[A]}{dt} = -k_1[A], \quad (2.11)$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B], \quad (2.12)$$

$$\frac{d[C]}{dt} = k_2[B]. \quad (2.13)$$

These equations can be solved symbolically (using `SymPy`) or numerically (using `SciPy`) to study the evolution of all species.

Parallel Reactions

Parallel reactions involve one reactant that forms multiple products via competing pathways:



The kinetic equations for parallel reactions are:

$$\frac{d[A]}{dt} = -(k_1 + k_2)[A], \quad (2.15)$$

$$\frac{d[B]}{dt} = k_1[A], \quad (2.16)$$

$$\frac{d[C]}{dt} = k_2[A]. \quad (2.17)$$

This framework is especially useful in industrial and biochemical processes where reaction conditions control product distribution.

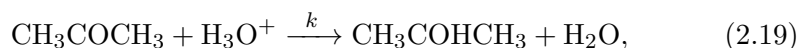
2.3.3 Self-Catalyzed Reactions

Self-catalyzed reactions are accelerated by one of their products. A classic example is the acid-catalyzed iodination of acetone:



This reaction proceeds in two stages:

1. Enolization (Rate-Determining Step):



2. Iodination: The enol form reacts with iodide to form the final product.

Due to the increasing concentration of H_3O^+ , the rate law for the enolization step becomes:

$$\frac{dC_A}{dt} = k C_A C_{\text{H}_3\text{O}^+}. \quad (2.20)$$

Expressing the reaction in terms of the extent $x(t)$ (with $x(0) = 0$) leads to:

$$\frac{dx(t)}{dt} = k \left(C_{A0} - x(t) \right) \left(C_{B0} + x(t) \right), \quad (2.21)$$

which gives rise to kinetic curves with characteristic bends due to the auto-catalytic effect.

2.4 Solving and Visualizing Kinetic Equations with Python

This section demonstrates two approaches:

1. **Symbolic Solutions** using SymPy to derive analytical expressions.
2. **Numerical Simulations and Plotting** using SciPy and Matplotlib to visualize concentration profiles.

2.4.1 Symbolic Derivation with SymPy

Below is an example Python script that uses SymPy to solve the rate equations for different reaction orders by separating variables and integrating.

```

1 from sympy import symbols, Eq, solve, integrate, pprint, latex,
  sqrt
2
3 # Define symbols for time (t), rate constant (k), and reaction
  order (n)
4 t, k, n = symbols('t_k_n')
5
6 # Define concentration variables: C_A (at time t) and initial
  concentration C_A0
7 C_A = symbols('C_A')
8 C_A0 = symbols('C_A0')
9
10 # Dictionary to store solutions for different reaction orders
11 solutions = {}
12
13 # Loop over different reaction orders: 0, 1, 2, and 3
14 for order in [0, 1, 2, 3]:
15     # Separate variables:  $C_A^{-order} dC_A = -k dt$ 
16     lhs_integrated = integrate(C_A**(-order), C_A)
17     rhs_integrated = integrate(-k, t)
18
19     # Form the general integrated solution with an integration
      constant C1
20     general_solution = Eq(lhs_integrated, rhs_integrated + symbols
      ('C1'))
21
22     # Use the initial condition  $C_A(0) = C_A0$  to solve for C1
23     C1_value = solve(general_solution.subs(t, 0).subs(C_A, C_A0),
      symbols('C1'))
24     if C1_value:
25         C1_value = C1_value[0] # Extract the solution
26         solution = solve(general_solution.subs(symbols('C1'),
          C1_value), C_A)
27         if solution:
28             solutions[order] = Eq(C_A, solution[0]) # Store the
              solution
29

```

```

30 # Display the solutions for each reaction order
31 for order, solution in solutions.items():
32     print(f"Solution for reaction order n={order}:")
33     pprint(solution)
34     print("LaTeX format:")
35     print(latex(solution))
36     print("\n" + "-"*50 + "\n")

```

Listing 2.1: Python code for solving integrated rate equations for various reaction orders using Sympy

2.4.2 Plotting Kinetic Curves

The following script demonstrates how to numerically evaluate and plot the concentration profiles for various reaction orders using Matplotlib.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Define numerical parameters
5 k_value = 1          # Rate constant
6 C_A0_value = 1       # Initial concentration
7 time = np.linspace(0, 1, 100) # Time points
8
9 # Define analytical functions for each reaction order
10 def C_A_0order(t):
11     return np.maximum(C_A0_value - k_value * t, 0)
12
13 def C_A_1order(t):
14     return C_A0_value * np.exp(-k_value * t)
15
16 def C_A_2order(t):
17     return C_A0_value / (1 + k_value * C_A0_value * t)
18
19 def C_A_3order(t):
20     return C_A0_value / np.sqrt(1 + 2 * k_value * C_A0_value**2 *
21                                t)
22
23 # Plot the curves
24 plt.figure(figsize=(8, 6))
25 plt.plot(time, C_A_0order(time), label='n=0', linestyle='solid')
26 plt.plot(time, C_A_1order(time), label='n=1', linestyle='dotted')
27 plt.plot(time, C_A_2order(time), label='n=2', linestyle='dashed')
28 plt.plot(time, C_A_3order(time), label='n=3', linestyle='dashdot')
29
30 plt.xlabel('Time(t)')
31 plt.ylabel('Concentration C_A')
32 plt.title('Kinetic Curves for Different Reaction Orders')
33 plt.legend()
34 plt.grid(True)

```

```
34 plt.show()
```

Listing 2.2: Plotting kinetic curves for different reaction orders using NumPy and Matplotlib

2.5 Modeling Reactions with Two Elementary Steps

In this section, we focus on reactions that involve two elementary steps. We consider both reversible and consecutive mechanisms in a unified approach.

2.5.1 Reversible Reactions Revisited

A reversible reaction consists of two opposing processes, where reactant A converts into product B and vice versa:



The reaction rate is given by:

$$r(t) = k_1 C_A(t) - k_2 C_B(t). \quad (2.23)$$

To simplify the mathematical analysis, we introduce the **extent of reaction** function $x(t)$, which represents the amount of A that has reacted by time t . Given the initial condition $x(0) = 0$, the governing differential equation is:

$$\frac{dx(t)}{dt} = k_1 [C_{A0} - x(t)] - k_2 [C_{B0} + x(t)]. \quad (2.24)$$

Solving this equation gives the explicit solution:

$$x(t) = \frac{k_1 C_{A0} + k_2 C_{B0}}{k_1 + k_2} \left[1 - e^{-(k_1 + k_2)t} \right]. \quad (2.25)$$

From this, the concentrations of A and B are determined as:

$$C_A(t) = C_{A0} - x(t), \quad (2.26)$$

$$C_B(t) = C_{B0} + x(t). \quad (2.27)$$

These equations describe how the reactant and product approach equilibrium over time.

2.5.2 Reversible Reactions Revisited

In many chemical systems, reactions are reversible; that is, the conversion of reactants to products occurs simultaneously with the reverse process. Consider the simple reversible reaction



where A transforms into B at a rate governed by k_1 while B converts back to A with rate constant k_2 . The net reaction rate at time t is given by

$$r(t) = k_1 C_A(t) - k_2 C_B(t), \quad (2.29)$$

with $C_A(t)$ and $C_B(t)$ being the concentrations of A and B , respectively.

To simplify the analysis, we introduce the **extent of reaction** $x(t)$, defined as the amount of A that has been converted to B by time t . With the initial condition $x(0) = 0$, the concentrations can be expressed as

$$C_A(t) = C_{A0} - x(t), \quad (2.30)$$

$$C_B(t) = C_{B0} + x(t), \quad (2.31)$$

where C_{A0} and C_{B0} are the initial concentrations of A and B , respectively. Substituting these relations into the rate expression leads to the governing differential equation:

$$\frac{dx(t)}{dt} = k_1 [C_{A0} - x(t)] - k_2 [C_{B0} + x(t)]. \quad (2.32)$$

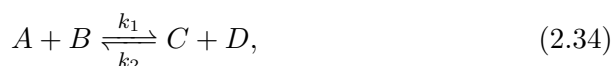
Its solution is

$$x(t) = \frac{k_1 C_{A0} + k_2 C_{B0}}{k_1 + k_2} \left[1 - e^{-(k_1 + k_2)t} \right], \quad (2.33)$$

from which the time-dependent concentrations of A and B are readily recovered. This result illustrates how the system gradually approaches equilibrium.

Second-Order Reversible Reaction

More complex systems may involve reactions that follow second-order kinetics. A common example is the bimolecular reversible reaction



where k_1 and k_2 are the rate constants for the forward ($A + B \rightarrow C + D$) and reverse ($C + D \rightarrow A + B$) reactions, respectively.

Defining the Reaction Progress. We again introduce the **extent of reaction** $x(t)$ so that the concentrations evolve as

$$[A](t) = a_0 - x(t), \quad (2.35)$$

$$[B](t) = b_0 - x(t), \quad (2.36)$$

$$[C](t) = c_0 + x(t), \quad (2.37)$$

$$[D](t) = d_0 + x(t), \quad (2.38)$$

with a_0 , b_0 , c_0 , and d_0 denoting the initial concentrations of A , B , C , and D .

The Rate Law. The forward and reverse reaction rates can be written as

$$r_{\text{forward}} = k_1 (a_0 - x(t)) (b_0 - x(t)), \quad (2.39)$$

$$r_{\text{reverse}} = k_2 (c_0 + x(t)) (d_0 + x(t)). \quad (2.40)$$

Thus, the net rate of reaction is

$$\frac{dx}{dt} = k_1 (a_0 - x) (b_0 - x) - k_2 (c_0 + x) (d_0 + x). \quad (2.41)$$

After expanding and grouping like terms, this rate law can be rearranged into the quadratic form

$$\frac{dx}{dt} = -\gamma x^2 + \lambda x + \delta, \quad (2.42)$$

with the parameters defined by

$$\gamma = k_1 + k_2,$$

$$\lambda = k_1(a_0 + b_0) - k_2(c_0 + d_0),$$

$$\delta = k_1 a_0 b_0 - k_2 c_0 d_0.$$

Step-by-Step Derivation of the Solution for $x(t)$

The following box outlines the derivation of the analytical solution for $x(t)$ using standard techniques.

Derivation of $x(t)$

Step 1: Separation of Variables

Rewrite Eq. (2.42) by separating the variables:

$$\frac{dx}{-\gamma x^2 + \lambda x + \delta} = dt.$$

Step 2: Completing the Square

Factor out $-\gamma$ from the quadratic term:

$$-\gamma x^2 + \lambda x + \delta = -\gamma \left[x^2 - \frac{\lambda}{\gamma} x - \frac{\delta}{\gamma} \right].$$

Complete the square in the bracket:

$$x^2 - \frac{\lambda}{\gamma} x = \left(x - \frac{\lambda}{2\gamma} \right)^2 - \frac{\lambda^2}{4\gamma^2}.$$

Thus,

$$x^2 - \frac{\lambda}{\gamma} x - \frac{\delta}{\gamma} = \left(x - \frac{\lambda}{2\gamma} \right)^2 - \left(\frac{\lambda^2}{4\gamma^2} + \frac{\delta}{\gamma} \right).$$

Define

$$\alpha^2 = \frac{\lambda^2}{4\gamma^2} + \frac{\delta}{\gamma},$$

so that

$$-\gamma x^2 + \lambda x + \delta = -\gamma \left[\left(x - \frac{\lambda}{2\gamma} \right)^2 - \alpha^2 \right].$$

For integration, rewrite the denominator as

$$\gamma \left[\alpha^2 + \left(x - \frac{\lambda}{2\gamma} \right)^2 \right],$$

leading to

$$\frac{dx}{\gamma \left[\alpha^2 + \left(x - \frac{\lambda}{2\gamma} \right)^2 \right]} = dt.$$

Step 3: Trigonometric Substitution and Integration

Let

$$u = x - \frac{\lambda}{2\gamma} \quad \text{so that} \quad du = dx.$$

Then,

$$\frac{du}{\gamma (\alpha^2 + u^2)} = dt.$$

Integrate both sides:

$$\int \frac{du}{\alpha^2 + u^2} = \gamma \int dt.$$

Recalling the standard integral

$$\int \frac{du}{\alpha^2 + u^2} = \frac{1}{\alpha} \tan^{-1} \left(\frac{u}{\alpha} \right) + \text{constant},$$

we have

$$\frac{1}{\alpha} \tan^{-1} \left(\frac{u}{\alpha} \right) = \gamma t + C.$$

Returning to x via $u = x - \frac{\lambda}{2\gamma}$ gives

$$\frac{1}{\alpha} \tan^{-1} \left(\frac{x - \frac{\lambda}{2\gamma}}{\alpha} \right) = \gamma t + C.$$

Multiplying by α yields

$$\tan^{-1} \left(\frac{x - \frac{\lambda}{2\gamma}}{\alpha} \right) = \gamma \alpha t + C',$$

where $C' = \alpha C$.

Step 4: Applying the Initial Condition

Since $x(0) = 0$,

$$\tan^{-1} \left(\frac{-\frac{\lambda}{2\gamma}}{\alpha} \right) = \tan^{-1} \left(-\frac{\lambda}{2\gamma\alpha} \right) = -\tan^{-1} \left(\frac{\lambda}{2\gamma\alpha} \right).$$

Thus, $C' = -\tan^{-1} \left(\frac{\lambda}{2\gamma\alpha} \right)$ and we obtain

$$\tan^{-1} \left(\frac{x - \frac{\lambda}{2\gamma}}{\alpha} \right) = \gamma \alpha t - \tan^{-1} \left(\frac{\lambda}{2\gamma\alpha} \right).$$

Step 5: Solving for $x(t)$

Taking the tangent of both sides leads to

$$\frac{x - \frac{\lambda}{2\gamma}}{\alpha} = \tan \left(\gamma \alpha t - \tan^{-1} \left(\frac{\lambda}{2\gamma\alpha} \right) \right).$$

Thus, solving for x gives

$$x(t) = \alpha \tan \left(\gamma \alpha t - \tan^{-1} \left(\frac{\lambda}{2\gamma\alpha} \right) \right) + \frac{\lambda}{2\gamma}.$$

With further algebraic manipulation (including appropriate re-scaling of time), the solution can be recast in the compact form

$$x(t) = \frac{2\delta}{\lambda + \gamma \tan\left(\frac{\lambda}{2}t\right)} \tan^{-1}\left(\frac{\lambda}{2}t\right).$$

Here, the parameter

$$\lambda = \sqrt{4\beta\delta - \gamma^2}$$

(with a suitable definition of β) combines the kinetic and concentration information.

Numerical and Analytical Solutions Using Python

To reinforce these ideas, we can solve the differential equation both **symbolically** (using SymPy) and **numerically** (using SciPy). The following Python code demonstrates the procedure for the simple reversible reaction.

```

1 from sympy import symbols, Function, Eq, dsolve, pprint, exp,
  solve
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.integrate import odeint
5
6 # Define symbols
7 t, k1, k2, A0, B0 = symbols('t k1 k2 A0 B0')
8 x = Function('x')(t)
9
10 # Define the differential equation
11 dx_dt = Eq(x.diff(t), k1 * (A0 - x) - k2 * (B0 + x))
12
13 # Solve the differential equation
14 sol_x = dsolve(dx_dt, x)
15 C1 = symbols('C1')
16
17 # Determine the integration constant using x(0)=0
18 C1_value = solve(sol_x.rhs.subs(t, 0) - 0, C1)[0]
19
20 # Substitute back to obtain x(t)
21 x_solution = sol_x.rhs.subs(C1, C1_value)
22
23 # Express concentrations in terms of x(t)
24 C_A = A0 - x_solution
25 C_B = B0 + x_solution
26
27 print("\nExplicit Solution for x(t):")
28 pprint(x_solution)
29 print("\nSolution for C_A(t):")
30 pprint(C_A)
31 print("\nSolution for C_B(t):")
32 pprint(C_B)

```



```

33
34 # Convert symbolic expressions to numerical functions
35 from sympy import lambdify
36 x_func = lambdify((t, k1, k2, A0, B0), x_solution, 'numpy')
37
38 # Numerical ODE model
39 def reversible_reaction(y, t, k1, k2, A0, B0):
40     x = y[0]
41     dx_dt = k1 * (A0 - x) - k2 * (B0 + x)
42     return [dx_dt]
43
44 # Define parameters and initial conditions
45 A0_val, B0_val = 1.0, 0.2
46 k1_val, k2_val = 0.45, 0.12
47 t_vals = np.linspace(0, 10, 100)
48 y0 = [0] # x(0)=0
49
50 # Solve numerically using odeint
51 sol_x_numeric = odeint(reversible_reaction, y0, t_vals, args=(
    k1_val, k2_val, A0_val, B0_val))
52
53 # Compute concentrations from numerical x(t)
54 C_A_numeric = A0_val - sol_x_numeric[:, 0]
55 C_B_numeric = B0_val + sol_x_numeric[:, 0]
56
57 # Plot the results
58 plt.figure(figsize=(8, 6))
59 plt.plot(t_vals, C_A_numeric, label=r'$C_A(t)$', linestyle='solid'
    )
60 plt.plot(t_vals, C_B_numeric, label=r'$C_B(t)$', linestyle='dotted'
    )
61 plt.xlabel('Time')
62 plt.ylabel('Concentration')
63 plt.title('Reversible Reaction Kinetics')
64 plt.legend()
65 plt.grid(True)
66 plt.show()

```

Listing 2.3: Python Code for Solving the Reversible Reaction ODE and Plotting Concentrations

Key Takeaways:

- **Analytical Approach:** Defining the extent of reaction $x(t)$ allows us to derive explicit expressions for the concentrations $C_A(t)$ and $C_B(t)$ and to study the approach to equilibrium.
- **Numerical Simulation:** Numerical integration with SciPy's `odeint` complements the analytical solution and aids in visualizing the kinetic behavior.

- **Graphical Insights:** Plots generated from the numerical solution clearly illustrate the decrease in A and the corresponding increase in B until equilibrium is established.

Below is an additional Python snippet for a second-order reversible reaction, following the same strategy.

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.integrate import odeint
5
6 # Define symbols
7 t, gamma, lambda_, delta = sp.symbols('t_gamma_lambda_delta', real
    =True, positive=True)
8 a0, b0, c0, d0, k1, k2 = sp.symbols('a0_b0_c0_d0_k1_k2', real=True
    , positive=True)
9
10 # Define x(t) as the extent of reaction
11 x = sp.Function('x')(t)
12
13 # Define the differential equation
14 ode = sp.Eq(sp.diff(x, t), -gamma * x**2 + lambda_ * x + delta)
15
16 # Solve the ODE with x(0)=0
17 sol = sp.dsolve(ode, x, ics={x.subs(t, 0): 0})
18 simplified_sol = sp.simplify(sol.rhs)
19
20 # Convert the solution to a numerical function
21 x_func = sp.lambdify((t, gamma, lambda_, delta), simplified_sol, '
    numpy')
22
23 # Define parameter values
24 A0_val, B0_val, C0_val, D0_val = 0.06, 0.05, 0.04, 0.03
25 k1_val, k2_val = 0.5, 0.25
26
27 # Calculate gamma, lambda, and delta
28 gamma_val = k1_val + k2_val
29 lambda_val = k1_val * (A0_val + B0_val) - k2_val * (C0_val +
    D0_val)
30 delta_val = k1_val * A0_val * B0_val - k2_val * C0_val * D0_val
31
32 # Define time array
33 t_vals = np.linspace(0, 60, 300)
34 x_vals = x_func(t_vals, gamma_val, lambda_val, delta_val)
35
36 # Define numerical ODE model for the second-order reaction
37 def second_order_reversible(y, t, k1, k2, A0, B0, C0, D0):
38     x = y[0]
39     dx_dt = k1 * (A0 - x) * (B0 - x) - k2 * (C0 + x) * (D0 + x)
40     return [dx_dt]
41
42 # Solve numerically with odeint
43 y0 = [0] # x(0)=0

```

```

44 sol_x_numeric = odeint(second_order_reversible, y0, t_vals, args=(
    k1_val, k2_val, A0_val, B0_val, C0_val, D0_val))
45
46 # Compute concentrations
47 C_A_numeric = A0_val - sol_x_numeric[:, 0]
48 C_B_numeric = B0_val - sol_x_numeric[:, 0]
49 C_C_numeric = C0_val + sol_x_numeric[:, 0]
50 C_D_numeric = D0_val + sol_x_numeric[:, 0]
51
52 # Plot the concentration profiles
53 plt.figure(figsize=(8, 6))
54 plt.plot(t_vals, C_A_numeric, label=r'$A_0 - x(t)$', linestyle='
    solid', color='black')
55 plt.plot(t_vals, C_B_numeric, label=r'$B_0 - x(t)$', linestyle='
    dotted', color='black')
56 plt.plot(t_vals, C_C_numeric, label=r'$C_0 + x(t)$', linestyle='
    dashed', color='black')
57 plt.plot(t_vals, C_D_numeric, label=r'$D_0 + x(t)$', linestyle='
    dashdot', color='black')
58 plt.xlabel('Time')
59 plt.ylabel('Concentration')
60 plt.title('Second-Order Reversible Reaction Kinetics')
61 plt.legend()
62 plt.grid(True)
63 plt.show()

```

Listing 2.4: Python Code for Second-Order Reversible Reaction

Key Takeaways:

- The analytical approach starts from the chemical equation by introducing the extent of reaction $x(t)$.
- Recasting the rate law into a quadratic form enables us to solve the differential equation using standard integration techniques, including trigonometric substitution.
- Symbolic computation with SymPy produces an elegant solution, while numerical integration with SciPy provides a practical tool for simulation.

2.5.3 Consecutive Reaction Mechanism

A consecutive reaction involves an intermediate species that forms from the initial reactant and subsequently converts into the final product. In its simplest form, the reaction is given by



At $t = 0$, the initial concentrations are

$$C_A(0) = C_{A0}, \quad C_B(0) = 0, \quad C_C(0) = 0. \quad (2.44)$$

The rate equations governing this system are

$$\frac{d[A]}{dt} = -k_1[A], \quad (2.45)$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B], \quad (2.46)$$

$$\frac{d[C]}{dt} = k_2[B]. \quad (2.47)$$

Since the equation for $[A](t)$ is a simple first-order decay, we can integrate it directly:

$$C_A(t) = C_{A0} e^{-k_1 t}. \quad (2.48)$$

Solving for the Intermediate $B(t)$

Substituting $C_A(t)$ into Eq. (2.46) gives

$$\frac{dC_B}{dt} + k_2 C_B = k_1 C_{A0} e^{-k_1 t}. \quad (2.49)$$

This is a first-order linear differential equation and can be solved using the integrating factor method. The integrating factor is

$$\mu(t) = e^{\int k_2 dt} = e^{k_2 t}. \quad (2.50)$$

Multiplying through by $\mu(t)$, we obtain

$$e^{k_2 t} \frac{dC_B}{dt} + k_2 e^{k_2 t} C_B = k_1 C_{A0} e^{(k_2 - k_1)t}. \quad (2.51)$$

Recognizing the left-hand side as the derivative of $C_B e^{k_2 t}$, we write

$$\frac{d}{dt} (C_B e^{k_2 t}) = k_1 C_{A0} e^{(k_2 - k_1)t}. \quad (2.52)$$

Integrating both sides from 0 to t (and using $C_B(0) = 0$) yields

$$C_B(t) e^{k_2 t} = \int_0^t k_1 C_{A0} e^{(k_2 - k_1)\tau} d\tau. \quad (2.53)$$

Evaluating the integral gives the solution

$$C_B(t) = C_{A0} \frac{k_1}{k_2 - k_1} (e^{-k_1 t} - e^{-k_2 t}), \quad (2.54)$$

provided that $k_2 \neq k_1$.

Material Balance and the Final Expression for $C_C(t)$

Applying a total mass balance (i.e. conservation of material)

$$C_{A0} = C_A(t) + C_B(t) + C_C(t), \quad (2.55)$$

we solve for the concentration of the final product C :

$$C_C(t) = C_{A0} - C_A(t) - C_B(t). \quad (2.56)$$

Substituting the expressions for $C_A(t)$ and $C_B(t)$ results in

$$C_C(t) = C_{A0} \left[1 - e^{-k_1 t} - \frac{k_1}{k_2 - k_1} \left(e^{-k_1 t} - e^{-k_2 t} \right) \right], \quad (2.57)$$

which describes the time-dependent concentration profiles of A , B , and C . In this mechanism, the intermediate B accumulates transiently before decaying into C .

Maximum Intermediate Concentration $C_{B,\max}$

Unlike the monotonically decaying A , the intermediate B first increases, reaches a maximum, and then decreases as it is converted to C . To find the time t_{\max} when $C_B(t)$ is maximum, one sets

$$\frac{dC_B}{dt} = 0. \quad (2.58)$$

Using the rate equation

$$\frac{dC_B}{dt} = k_1 C_{A0} e^{-k_1 t} - k_2 C_B(t), \quad (2.59)$$

and solving for t leads to

$$t_{\max} = \frac{\ln(k_2/k_1)}{k_2 - k_1}. \quad (2.60)$$

At $t = t_{\max}$, the peak concentration of the intermediate is given by

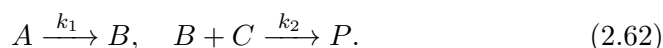
$$C_{B,\max} = C_{A0} \frac{k_1}{k_2 - k_1} \left[\left(\frac{k_2}{k_1} \right)^{-\frac{k_1}{k_2 - k_1}} - \left(\frac{k_2}{k_1} \right)^{-\frac{k_2}{k_2 - k_1}} \right]. \quad (2.61)$$

Interpretation:

- The timing and magnitude of $C_{B,\max}$ depend on the ratio k_1/k_2 .
- If $k_1 \gg k_2$, the intermediate accumulates substantially before converting to C .
- If $k_1 \ll k_2$, B is quickly consumed, and its concentration remains low.

A More Complex Successive Reaction Mechanism

In many practical systems, not all steps follow simple first-order kinetics. For example, consider a mechanism where the first step remains first order, but the second step is a **second-order** reaction:



In this case:

- The first step is a first-order irreversible reaction, converting A to intermediate B with rate constant k_1 .
- The second step is a second-order reaction in which B reacts with an additional reagent C to form the final product P , with rate constant k_2 .

Mathematical Model. Assuming the concentration of C remains constant (or is in large excess), we denote the concentration functions as $C_A(t)$, $C_B(t)$, and $C_C(t)$ (with $C_C(t)$ taken as constant). The differential equations become

$$\frac{dC_A}{dt} = -k_1 C_A, \quad (2.63)$$

$$\frac{dC_B}{dt} = k_1 C_A - k_2 C_B C_C. \quad (2.64)$$

The solution for $C_A(t)$ remains

$$C_A(t) = C_{A0} e^{-k_1 t}. \quad (2.65)$$

Substituting into the equation for $C_B(t)$ yields

$$\frac{dC_B}{dt} + k_2 C_C C_B = k_1 C_{A0} e^{-k_1 t}. \quad (2.66)$$

Following the integrating factor method with

$$\mu(t) = e^{\int k_2 C_C dt} = e^{k_2 C_C t}, \quad (2.67)$$

we multiply through by $\mu(t)$ and integrate:

$$\frac{d}{dt} \left(C_B e^{k_2 C_C t} \right) = k_1 C_{A0} e^{(k_2 C_C - k_1)t}. \quad (2.68)$$

Integration from 0 to t (with $C_B(0) = 0$) gives

$$C_B(t) e^{k_2 C_C t} = \int_0^t k_1 C_{A0} e^{(k_2 C_C - k_1)\tau} d\tau. \quad (2.69)$$

Evaluating the integral leads to

$$C_B(t) = C_{A0} \frac{k_1}{k_2 C_C - k_1} \left(e^{-k_1 t} - e^{-k_2 C_C t} \right), \quad (2.70)$$

provided that $k_2 C_C \neq k_1$. This expression captures the transient accumulation of the intermediate B in a system where the second step exhibits second-order kinetics.

Interpretation:

- If $k_1 \gg k_2 C_C$, the conversion of A to B is fast and B builds up before being consumed.
- If $k_1 \ll k_2 C_C$, the intermediate B reacts almost as quickly as it is formed, resulting in a low concentration.
- The two exponential terms represent the competing effects of formation and consumption of B .

Python Implementation Using SymPy and SciPy

The following Python code snippets (using the `listings` package) demonstrate how to verify these solutions symbolically with SymPy and numerically integrate the differential equations with SciPy.

```

1 from sympy import symbols, Function, Eq, dsolve, pprint, exp,
   solve
2
3 # Define symbols and functions
4 t, k1, k2, C_A0 = symbols('t k1 k2 C_A0')
5 C_A = Function('C_A')(t)
6 C_B = Function('C_B')(t)
7 C_C = Function('C_C')(t)
8
9 # Differential equations for the basic mechanism
10 eq1 = Eq(C_A.diff(t), -k1 * C_A)
11 eq2 = Eq(C_B.diff(t), k1 * C_A - k2 * C_B)
12
13 # Solve for C_A(t)
14 sol_A = dsolve(eq1, C_A)
15 C_A_solution = sol_A.rhs.subs('C1', C_A0)
16
17 # Substitute C_A(t) into eq2 and solve for C_B(t)
18 eq2_subs = eq2.subs(C_A, C_A_solution)
19 sol_B = dsolve(eq2_subs, C_B)
20 C1_B = symbols('C1_B')
21 general_solution_B = sol_B.rhs.subs('C1', C1_B)
22 C1_B_value = solve(general_solution_B.subs(t, 0), C1_B)[0]
23 C_B_solution = general_solution_B.subs(C1_B, C1_B_value)
24
25 print("Solution for C_A(t):")

```

```

26 pprint(C_A_solution)
27 print("\nSolution for C_B(t):")
28 pprint(C_B_solution)

```

Listing 2.5: Symbolic Solution with SymPy

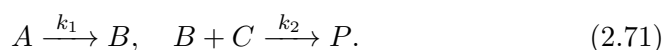
```

1 from scipy.integrate import odeint
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Define the model for the basic consecutive reaction A -> B -> C
6 def model(y, t, k1, k2):
7     A, B, C = y
8     dA_dt = -k1 * A
9     dB_dt = k1 * A - k2 * B
10    dC_dt = k2 * B
11    return [dA_dt, dB_dt, dC_dt]
12
13 # Initial conditions and parameters
14 A0, B0, C0 = 1.0, 0.0, 0.0
15 k1, k2 = 1.0/2.0, 1.0/3.0
16 time = np.linspace(0, 10, 100)
17
18 # Compute t_max for the intermediate B
19 t_max = np.log(k2 / k1) / (k2 - k1)
20
21 # Solve the ODE system
22 y0 = [A0, B0, C0]
23 sol = odeint(model, y0, time, args=(k1, k2))
24
25 # Plot the results
26 plt.figure(figsize=(8, 6))
27 plt.plot(time, sol[:, 0], label='[A]', linestyle='solid')
28 plt.plot(time, sol[:, 1], label='[B]', linestyle='dotted')
29 plt.plot(time, sol[:, 2], label='[C]', linestyle='dashed')
30 plt.axvline(x=t_max, linestyle="--", color="red", label=r"$t_{\max}$")
31 plt.xlabel('Time')
32 plt.ylabel('Concentration')
33 plt.title('Consecutive Reaction Kinetics')
34 plt.legend()
35 plt.grid(True)
36 plt.show()

```

Listing 2.6: Numerical Integration and Visualization with SciPy

The following Python code demonstrates how to numerically solve the reaction system for the complex successive mechanism, where the reaction scheme is:



In this simulation, we assume the following initial conditions:

2.5. MODELING REACTIONS WITH TWO ELEMENTARY STEPS 33

- $A_0 = 1.0$ (initial concentration of A)
- $B_0 = 0.0$ (no B is present initially)
- $C_0 = 0.9$ (initial concentration of C)
- $P_0 = 0.0$ (no product P is present initially)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import odeint
4
5 # Define reaction rate constants
6 k1 = 0.5    # Rate constant for A -> B
7 k2 = 0.1    # Rate constant for B + C -> P
8
9 # Initial concentrations (Reordered for better logical flow)
10 A0 = 1.0    # Initial concentration of A
11 B0 = 0.0    # Initially, no B is present
12 C0 = 0.9    # Initial concentration of C
13 P0 = 0.0    # Initially, no P is present
14
15 # Time range for the simulation
16 t = np.linspace(0, 20, 200) # Time from 0 to 20 units
17
18 # Define the system of differential equations
19 def reaction_rates(y, t, k1, k2):
20     A, B, C, P = y # Now the variables follow the same order as
21                     # the initial conditions
22     dA_dt = -k1 * A # A -> B
23     dB_dt = k1 * A - k2 * B * C # B is formed and consumed
24                               # in B + C -> P
25     dC_dt = -k2 * B * C # C is consumed in the
26                       # reaction
27     dP_dt = k2 * B * C # P is formed from B + C
28     return [dA_dt, dB_dt, dC_dt, dP_dt]
29
30 # Solve the system numerically
31 initial_conditions = [A0, B0, C0, P0] # Reordered for consistency
32 solution = odeint(reaction_rates, initial_conditions, t, args=(k1,
33                       k2))
34
35 # Extract the concentrations from the solution array
36 A_conc = solution[:, 0]
37 B_conc = solution[:, 1]
38 C_conc = solution[:, 2]
39 P_conc = solution[:, 3] # Now C(t) is properly changing over time
40
41 # Plot the results
42 plt.figure(figsize=(8, 6))
43 plt.plot(t, A_conc, label=r'$A(t)$', linestyle='solid')
44 plt.plot(t, B_conc, label=r'$B(t)$', linestyle='dashdot')
45 plt.plot(t, C_conc, label=r'$C(t)$', linestyle='dotted', color="
green")

```

```

42 plt.plot(t, P_conc, label=r'$P(t)$', linestyle='dashed', color="
    red")
43
44 # Labels and title
45 plt.xlabel('Time')
46 plt.ylabel('Concentration')
47 plt.title('Kinetics of a Complex Successive Reaction')
48 plt.legend()
49 plt.grid(True)
50
51 # Show the plot
52 plt.show()

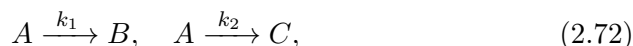
```

Listing 2.7: Numerical Simulation of a Complex Successive Reaction

2.5.4 Parallel Reactions

Parallel reactions occur when a single reactant can undergo multiple reaction pathways, leading to different products. Such reactions are common in organic and catalytic processes, where a reactant has several possible transformation routes.

A general scheme for a parallel reaction is given by:



where:

- k_1 and k_2 are the rate constants for the competing pathways.
- B and C are the distinct products formed through separate routes.

Rate Equations and Analytical Solution

The kinetic model describing the concentration changes over time is governed by the following system of differential equations:

$$\frac{dC_A}{dt} = -(k_1 + k_2) C_A, \quad (2.73)$$

$$\frac{dC_B}{dt} = k_1 C_A, \quad (2.74)$$

$$\frac{dC_C}{dt} = k_2 C_A. \quad (2.75)$$

Step 1: Solving for $C_A(t)$ Since the differential equation for $C_A(t)$ is a simple first-order decay, its solution is

$$C_A(t) = C_{A0} e^{-(k_1+k_2)t}, \quad (2.76)$$

where C_{A0} is the initial concentration of A .

Step 2: Solving for $C_B(t)$ and $C_C(t)$ Substitute the expression for $C_A(t)$ into the rate equations for $C_B(t)$ and $C_C(t)$. For $C_B(t)$ we have

$$C_B(t) = \int_0^t k_1 C_A(\tau) d\tau = k_1 C_{A0} \int_0^t e^{-(k_1+k_2)\tau} d\tau. \quad (2.77)$$

Evaluating the integral gives

$$C_B(t) = \frac{k_1}{k_1 + k_2} C_{A0} \left(1 - e^{-(k_1+k_2)t}\right). \quad (2.78)$$

Similarly, for $C_C(t)$:

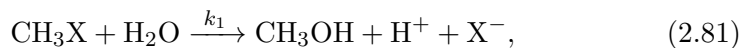
$$C_C(t) = \frac{k_2}{k_1 + k_2} C_{A0} \left(1 - e^{-(k_1+k_2)t}\right). \quad (2.79)$$

It is evident from these expressions that the ratio of the product concentrations remains constant:

$$\frac{C_B(t)}{C_C(t)} = \frac{k_1}{k_2}. \quad (2.80)$$

Parallel Second-Order Reactions: Hydrolysis of Methyl Halogenide

Many important reactions involve second-order parallel kinetics. A well-known example is the hydrolysis of methyl halogenide (CH_3X) in an alkaline solution, which proceeds through two competing reaction pathways:



Here, both water and hydroxide ions compete to react with the methyl halogenide. The first stage follows pseudo-first-order kinetics due to the excess solvent, while the second stage is a bimolecular elementary reaction.

If the initial concentrations of methyl halogenide (CH_3X) and hydroxide ions (OH^-) are a and b , respectively, then the kinetics can be described by the differential equation:

$$\frac{dx(t)}{dt} = k_1(a - x(t)) + k_2(a - x(t))(b - x(t)). \quad (2.83)$$

where $x(t)$ represents the reacted fraction of CH_3X .

By introducing the substitutions:

$$\alpha = k_1 + k_2(b - a), \quad (2.84)$$

$$\beta = \frac{k_2 a}{k_1 + k_2 b}, \quad (2.85)$$

the equation simplifies, leading to an analytical solution. The concentration profiles of CH_3X , CH_3OH , and the intermediate species can be plotted to visualize the reaction kinetics.

Python Implementation

The following Python code numerically integrates the system of ordinary differential equations and plots the concentration profiles of A , B , and C :

```

1 from scipy.integrate import odeint
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Define rate equations for parallel reactions
6 def parallel_reactions(y, t, k1, k2):
7     A, B, C = y
8     dA_dt = -(k1 + k2) * A
9     dB_dt = k1 * A
10    dC_dt = k2 * A
11    return [dA_dt, dB_dt, dC_dt]
12
13 # Initial conditions
14 A0, B0, C0 = 1.0, 0.0, 0.0 # Initially, only A is present
15 k1, k2 = 0.81, 0.27        # Rate constants for the competing
16                             # pathways
17 time = np.linspace(0, 10, 100) # Time range for the simulation
18
19 # Solve the system numerically
20 y0 = [A0, B0, C0]
21 solution = odeint(parallel_reactions, y0, time, args=(k1, k2))
22
23 # Extract the concentration profiles
24 A_conc = solution[:, 0]
25 B_conc = solution[:, 1]
26 C_conc = solution[:, 2]
27
28 # Plot the results
29 plt.figure(figsize=(8, 6))
30 plt.plot(time, A_conc, label=r'$C_A(t)$', linestyle='solid')
31 plt.plot(time, B_conc, label=r'$C_B(t)$', linestyle='dotted')
32 plt.plot(time, C_conc, label=r'$C_C(t)$', linestyle='dashed')
33 plt.xlabel('Time (t)')
34 plt.ylabel('Concentration')
35 plt.title('Parallel Reaction Kinetics')
36 plt.legend()
37 plt.grid(True)
38 plt.show()

```

Listing 2.8: Numerical Simulation of Parallel Reaction Kinetics

Interpretation

- The concentration of $A(t)$ decreases exponentially because it is consumed simultaneously in both reaction pathways.
- The products $B(t)$ and $C(t)$ are formed with growth patterns that complement the decay of $A(t)$.
- The constant ratio $\frac{C_B(t)}{C_C(t)} = \frac{k_1}{k_2}$ shows that the relative amounts of products B and C depend solely on the rate constants.

The Python script below models the kinetics of a second-order parallel reaction, in which the methyl halogenide (CH_3X) reacts via two competing pathways:



In this model, the variable $x(t)$ represents the extent of reaction (i.e., the fraction of CH_3X that has reacted at time t). The rate equation accounts for both pathways. The script uses the `odeint` function to solve the differential equation numerically and then extracts the concentration profiles for CH_3X (remaining reactant) and the two products formed via each pathway. Finally, it plots the kinetic curves.

```

1 from scipy.integrate import odeint
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Define the system of differential equations for the parallel
  reaction.
6 # This function models a second-order parallel reaction:
7 #   CH3X + H2O ->[k1] CH3OH   (Pathway 1)
8 #   CH3X + OH^- ->[k2] CH3OH   (Pathway 2)
9 def parallel_reaction_kinetics(y, t, k1, k2, C_A0, C_B0):
10     """
11     Parameters:
12     y: list -> [x], where x is the extent of reaction at
       time t.
13     t: float -> Time.
14     k1: float -> Rate constant for the hydrolysis pathway (
       with H2O).
15     k2: float -> Rate constant for the reaction with OH^-.
16     C_A0: float -> Initial concentration of CH3X.
17     C_B0: float -> Initial concentration of OH^-.
18
19     Returns:
20     A list containing the derivative dx/dt.
21     """
22     x = y[0] # x(t) is the reacted fraction of CH3X.
23 
```

```

24     # Rate equation for the extent of reaction, considering both
      pathways.
25     dx_dt = k1 * (C_A0 - x) + k2 * (C_A0 - x) * (C_B0 - x)
26
27     return [dx_dt]
28
29 # Initial Conditions
30 C_A0 = 1.0 # Initial concentration of CH3X
31 C_B0 = 0.5 # Initial concentration of OH-
32 x0   = [0] # Initially, no reaction has occurred
33
34 # Rate Constants
35 k1 = 0.3 # Rate constant for hydrolysis with water
36 k2 = 0.1 # Rate constant for reaction with OH-
37
38 # Time Grid for the Simulation
39 time = np.linspace(0, 10, 100) # Simulate from t = 0 to t = 10
      seconds
40
41 # Solve the ODE System Numerically
42 solution = odeint(parallel_reaction_kinetics, x0, time, args=(k1,
      k2, C_A0, C_B0))
43
44 # Extract the Concentration Profiles
45 x_t      = solution[:, 0] # Extent of reaction
46 C_A_t    = C_A0 - x_t     # Remaining CH3X concentration
47 # The following approximations assign portions of the reacted CH3X
48 # to the two pathways based on the relative rate constants.
49 C_B_t    = (k1 / (k1 + k2 * C_B0)) * x_t # CH3OH from the water
      pathway
50 C_C_t    = (k2 * C_A0 / (k1 + k2 * C_B0)) * x_t # CH3OH from the OH-
      pathway
51
52 # Plot the Results
53 plt.figure(figsize=(8, 6))
54 plt.plot(time, C_A_t, label=r'$C_A(t)$-CH3X', linestyle='solid',
      color='blue')
55 plt.plot(time, C_B_t, label=r'$C_B(t)$-CH3OH(Water Pathway)',
      linestyle='dotted', color='green')
56 plt.plot(time, C_C_t, label=r'$C_C(t)$-CH3OH(OH- Pathway)',
      linestyle='dashed', color='red')
57
58 # Graph Formatting
59 plt.xlabel('Time(t)[s]')
60 plt.ylabel('Concentration[M]')
61 plt.title('Second-Order Parallel Reaction Kinetics')
62 plt.legend()
63 plt.grid(True)
64 plt.show()

```

Listing 2.9: Numerical Simulation of a Second-Order Parallel Reaction

Interpretation of the Results

- The concentration of CH_3X ($C_A(t)$) decreases as it reacts via both pathways.
- The products, represented by $C_B(t)$ and $C_C(t)$, increase over time, with their relative amounts governed by the rate constants k_1 and k_2 as well as the initial concentration of OH^- .
- The fraction of product formation is influenced by the competition between the water and hydroxide ion pathways, demonstrating how variations in reaction conditions affect the product distribution.

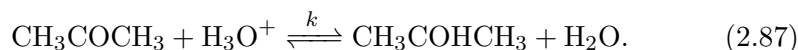
2.5.5 Simplest Self-Catalyzed Reaction

A **self-catalyzed** reaction is one in which a product of the reaction accelerates its own formation. A classic example is the iodination of acetone in an acidic medium, represented by the overall reaction:



The reaction mechanism proceeds in two main stages:

1. **Enolization (Rate-Determining Step):** In the first step, acetone is converted into its enol form. This step is relatively slow and is given by:



2. **Iodination of the Enol:** Once the enol is formed, it reacts rapidly with iodide to produce the final products.

As the reaction advances, the concentration of hydronium ions increases, which in turn accelerates the enolization step. Consequently, the overall reaction rate is controlled by the acetone enolization process and can be expressed as:

$$\frac{dC_A}{dt} = kC_A C_{\text{H}_3\text{O}^+}, \quad (2.88)$$

where C_A denotes the concentration of acetone, and k is the rate constant for the enolization step.

If we denote the initial concentrations of acetone and hydronium ions by C_{A0} and C_{B0} , respectively, and introduce $x(t)$ as the reacted fraction of acetone at time t , then the kinetic equation for the self-catalyzed reaction is:

$$\frac{dx(t)}{dt} = k(C_{A0} - x(t))(C_{B0} + x(t)), \quad (2.89)$$

with the initial condition $x(0) = 0$.

The solution of this differential equation provides the concentration profile of the autocatalytic species (hydronium ions) as:

$$C_B(t) = C_{B0} + x(t) = \frac{C_{A0} + C_{B0}}{1 + \frac{C_{A0}}{C_{B0}}e^{-k(C_{A0}+C_{B0})t}}. \quad (2.90)$$

This expression is analogous to the logistic growth equation, which is commonly observed in various natural phenomena. The corresponding kinetic curve shows an initial plateau, followed by rapid growth, and eventually levels off as the reaction approaches completion.

Maximum Reaction Rate

The reaction rate reaches a maximum at a specific time t_{\max} . This point is determined by setting the second derivative of $C_B(t)$ with respect to time equal to zero. The resulting expression for t_{\max} is:

$$t_{\max} = \frac{\ln(C_{A0}/C_{B0})}{k(C_{A0} + C_{B0})}. \quad (2.91)$$

It is important to emphasize that although species B (hydronium ions) enhances the reaction rate, it accumulates in significant quantities during the reaction. Therefore, it does not meet the classical definition of a catalyst.

Python Implementation

The following Python script numerically integrates the self-catalyzed reaction equation and plots the concentration profiles of $C_A(t)$ and $C_B(t)$:

```

1 from scipy.integrate import odeint
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import sympy as sp # Import sympy for symbolic computations
5
6 # Define the ODE for the self-catalyzed reaction
7 def self_catalyzed(y, t, k, CA0, CB0):
8     x = y[0] # Reacted fraction of acetone
9     dx_dt = k * (CA0 - x) * (CB0 + x)
10    return [dx_dt]
11
12 # Initial conditions
13 CA0, CB0 = 0.8, 0.001 # Initial concentrations of acetone and
    hydronium ions
14 x0 = [0] # Initial reacted fraction
15 k = 0.2 # Rate constant
16

```



```

17 # Define the time points for integration
18 time = np.linspace(0, 100, 200)
19
20 # Solve the differential equation
21 solution = odeint(self_catalyzed, x0, time, args=(k, CA0, CB0))
22
23 # Extract the data
24 x_t = solution[:, 0]
25 CB_t = CB0 + x_t # Concentration of hydronium ions (product B)
26 CA_t = CA0 - x_t # Concentration of acetone
27 reaction_rate = k * CA_t * CB_t # Reaction rate
28
29 # Symbolic computation of t_max using sympy
30 t_sym, CA0_sym, CB0_sym, k_sym = sp.symbols('t_CA0_CB0_k')
31 t_max_expr = sp.log(CA0_sym / CB0_sym) / (k_sym * (CA0_sym +
32     CB0_sym)) # Expression for t_max
33
34 # Pretty print the symbolic expression for t_max
35 print("\nSymbolic expression for t_max:")
36 sp.pprint(t_max_expr)
37
38 # Evaluate t_max numerically
39 t_max_value = t_max_expr.subs({CA0_sym: CA0, CB0_sym: CB0, k_sym:
40     k}).evalf()
41 print(f"\nNumerical value of t_max = {t_max_value:.4f}")
42
43 # Create the figure and subplots
44 fig, axes = plt.subplots(2, 1, figsize=(8, 10))
45
46 # First plot: Concentration profiles
47 axes[0].plot(time, CA_t, label=r'$C_A(t)$', linestyle='solid')
48 axes[0].plot(time, CB_t, label=r'$C_B(t)$', linestyle='dashed')
49 axes[0].set_xlabel('Time(t)')
50 axes[0].set_ylabel('Concentration')
51 axes[0].set_title('Self-Catalyzed Reaction Kinetics')
52 axes[0].legend()
53 axes[0].grid(True)
54
55 # Second plot: Reaction rate vs. time
56 axes[1].plot(time, reaction_rate, label=r'$k C_A(t) C_B(t)$',
57     linestyle='solid')
58 axes[1].axvline(x=float(t_max_value), color='red', linestyle='
59     dashed', label=r'$t_{\max}$')
60 axes[1].set_xlabel('Time(t)')
61 axes[1].set_ylabel('Reaction Rate')
62 axes[1].set_title('Reaction Rate vs Time')
63 axes[1].legend()
64 axes[1].grid(True)
65
66 plt.tight_layout()
67 plt.show()

```

Listing 2.10: Python script for simulating the kinetics of a self-catalyzed reaction

Key Observations

- The concentration of acetone, $C_A(t)$, decreases as it is consumed during the reaction.
- The autocatalytic species $C_B(t)$ follows a logistic growth pattern: it starts with a slow accumulation, then undergoes rapid growth, and finally levels off.
- The reaction rate reaches its maximum at t_{\max} , as derived above.
- Although $C_B(t)$ accelerates the reaction, its significant accumulation distinguishes it from a conventional catalyst.

2.6 Hands-On Activities and Assignment**2.6.1 Activity 1: Symbolic Derivation of Rate Laws**

- Use SymPy to derive the rate law for a first-order reaction.
- Verify your solution by comparing with the standard form $[A] = [A]_0 e^{-kt}$.

```

1 from sympy import symbols, Function, Eq, dsolve, pprint
2
3 t, k = symbols('t k')
4 A = Function('A')(t)
5 deq = Eq(A.diff(t), -k * A)
6 sol = dsolve(deq, A)
7 pprint(sol)

```

2.6.2 Activity 2: Numerical Integration of a Consecutive Reaction

- Model the consecutive reaction $A \rightarrow B \rightarrow C$ by solving the differential equations numerically.
- Plot the concentration profiles of A , B , and C over time.

```

1 from scipy.integrate import odeint
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 def model(y, t, k1, k2):
6     A, B, C = y
7     dA_dt = -k1 * A
8     dB_dt = k1 * A - k2 * B
9     dC_dt = k2 * B

```

```

10     return [dA_dt, dB_dt, dC_dt]
11
12 time = np.linspace(0, 10, 100)
13 k1, k2 = 1.0, 0.5
14 y0 = [1.0, 0.0, 0.0]
15 sol = odeint(model, y0, time, args=(k1, k2))
16
17 plt.plot(time, sol[:, 0], label='[A]')
18 plt.plot(time, sol[:, 1], label='[B]')
19 plt.plot(time, sol[:, 2], label='[C]')
20 plt.xlabel('Time')
21 plt.ylabel('Concentration')
22 plt.legend()
23 plt.title('Consecutive Reaction Kinetics')
24 plt.grid(True)
25 plt.show()

```

2.6.3 Activity 3: Second-Order Reaction with Two Reactants

Overview: Consider a second-order reaction:



with the rate equation:

$$\frac{dx}{dt} = k(A_0 - x)(B_0 - x), \quad (2.93)$$

where x is the extent of reaction. Use SymPy to derive the integrated rate law and then plot the concentration profiles for A and B .

Step 1: Derivation

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sympy import symbols, Eq, integrate, solve, pprint, lambdify
4
5 # Define symbols and parameters
6 t, k = symbols('t k', real=True, positive=True)
7 A0, B0 = symbols('A0 B0', real=True, positive=True)
8 x = symbols('x', real=True, positive=True)
9 C1 = symbols('C1')
10
11 # Integrate the separated variables: 1/[(A0-x)(B0-x)] dx = k dt
12 lhs_integrated = integrate(1/((A0 - x)*(B0 - x)), x)
13 rhs_integrated = integrate(k, t)
14 integrated_eq = Eq(lhs_integrated, rhs_integrated + C1)
15
16 # Determine the integration constant using x(0) = 0
17 C1_value = solve(integrated_eq.subs(t, 0).subs(x, 0), C1)[0]
18 final_equation = integrated_eq.subs(C1, C1_value)
19

```

```

20 # Solve explicitly for x(t)
21 x_solution = solve(final_equation, x)[0]
22 print("Final_Integrated_Rate_Law:")
23 pprint(Eq(x, x_solution))

```

Step 2: Plotting the Kinetic Curves

```

1 # Convert the symbolic solution to a numerical function
2 x_numeric = lambdify((t, A0, B0, k), x_solution, "numpy")
3
4 # Set numerical parameters
5 k_value = 0.02
6 A0_value = 0.08
7 B0_value = 0.06
8 time = np.linspace(0, 10000, 200)
9
10 # Compute the extent of reaction and reactant concentrations
11 x_values = x_numeric(time, A0_value, B0_value, k_value)
12 A_values = A0_value - x_values
13 B_values = B0_value - x_values
14
15 # Ensure concentrations remain non-negative
16 A_values[A_values < 0] = 0
17 B_values[B_values < 0] = 0
18
19 # Plot the results
20 plt.figure(figsize=(8, 6))
21 plt.plot(time, A_values, label='[A]_(Reactant_A)')
22 plt.plot(time, B_values, label='[B]_(Reactant_B)')
23 plt.xlabel('Time_(s)')
24 plt.ylabel('Concentration_(mol/L)')
25 plt.title('Second-Order_Kinetics_with_Two_Reactants')
26 plt.legend()
27 plt.grid(True)
28 plt.show()

```

Assignments

Task: Implement a Python script that models the consecutive reaction $A \rightarrow B \rightarrow C$. Your submission should include:

- A mathematical derivation of the rate laws.
- A complete Python code implementation using both SymPy (for analytical solutions) and SciPy (for numerical integration).
- Graphical plots of the reactant and product concentrations over time.
- A brief discussion interpreting your results and comparing them with experimental data (if available).

Requirements:

- Define the differential equations for the reaction sequence.
- Use `sympy.dsolve()` to obtain the symbolic solution.
- Validate your model by comparing your computed results with experimental datasets.

Additional Assignments:

- Extend the model to include a reversible reaction $B \rightleftharpoons C$ and analyze its effect on product formation.
- Compare the behavior of first-order and second-order kinetics for the reaction $A \rightarrow B \rightarrow C$.
- Implement a stochastic simulation of the reaction using the Gillespie algorithm and compare it with deterministic solutions.
- Develop an interactive Jupyter Notebook with sliders to modify reaction rate constants dynamically and visualize real-time changes in concentration profiles.
- Investigate the impact of different initial concentrations on the reaction outcome and discuss possible experimental implications.

Chapter 3

Multi-Step Reactions and Analytical Solutions

3.1 Introduction

Multi-step reactions involve a series of elementary reactions that collectively define the overall chemical transformation. Understanding their kinetics requires developing mathematical models, solving reaction rate equations, and using analytical techniques such as the matrix method. This chapter introduces these concepts and provides computational tools for solving multi-step kinetic models.

3.2 Developing a Mathematical Model of a Reaction

To analyze multi-step reactions, we first establish a mathematical framework:

1. Identify all elementary steps and corresponding rate constants.
2. Write rate equations for each species based on the law of mass action.
3. Represent the system as a set of coupled differential equations.
4. Express the system in matrix form for computational efficiency.

Consider a reaction with consecutive steps:



The rate equations are:

$$\frac{d[A]}{dt} = -k_1[A] \quad (3.2)$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B] \quad (3.3)$$

$$\frac{d[C]}{dt} = k_2[B] \quad (3.4)$$

Representing this system in matrix form:

$$\frac{d}{dt} \begin{bmatrix} [A] \\ [B] \\ [C] \end{bmatrix} = \begin{bmatrix} -k_1 & 0 & 0 \\ k_1 & -k_2 & 0 \\ 0 & k_2 & 0 \end{bmatrix} \begin{bmatrix} [A] \\ [B] \\ [C] \end{bmatrix} \quad (3.5)$$

The stoichiometric matrix, which represents the changes in reactant and product concentrations for each step, is:

$$\alpha = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \quad (3.6)$$

The rate vector, which contains the reaction rates for each elementary step, is given by:

$$r = \begin{bmatrix} k_1[A] \\ k_2[B] \end{bmatrix} \quad (3.7)$$

To find the time evolution of species concentrations, we compute the product of the transposed stoichiometric matrix and the rate vector:

$$\frac{d}{dt} C(t) = \alpha^T r \quad (3.8)$$

Explicitly calculating:

$$\alpha^T r = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} k_1[A] \\ k_2[B] \end{bmatrix} = \begin{bmatrix} -k_1[A] \\ k_1[A] - k_2[B] \\ k_2[B] \end{bmatrix} \quad (3.9)$$

Expanding the matrix multiplication:

$$\frac{d[A]}{dt} = -k_1[A] \quad (3.10)$$

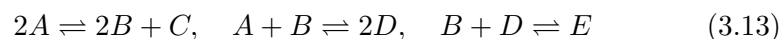
$$\frac{d[B]}{dt} = k_1[A] - k_2[B] \quad (3.11)$$

$$\frac{d[C]}{dt} = k_2[B] \quad (3.12)$$

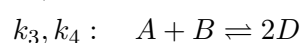
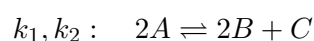
which recovers the original set of differential equations. This demonstrates how the stoichiometric matrix method simplifies constructing and solving reaction kinetics models.

3.3 Jupyter Notebook Example using SymPy

To demonstrate solving this system symbolically, we use SymPy in a Jupyter Notebook: Consider a reaction with consecutive steps:



The forward and backward rate constants for each step are denoted as follows:



The stoichiometric matrix, which represents the changes in reactant and product concentrations for each step, is:

$$\alpha = \begin{bmatrix} -2 & 2 & 1 & 0 & 0 \\ 2 & -2 & -1 & 0 & 0 \\ -1 & -1 & 0 & 2 & 0 \\ 1 & 1 & 0 & -2 & 0 \\ 0 & -1 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 & -1 \end{bmatrix} \quad (3.14)$$

The rate vector, which contains the reaction rates for each elementary step, is given by:

$$r = \begin{bmatrix} k_1[A]^2 \\ k_2[B]^2[C] \\ k_3[A][B] \\ k_4[D]^2 \\ k_5[B][D] \\ k_6[E] \end{bmatrix} \quad (3.15)$$

To find the time evolution of species concentrations, we compute the product of the transposed stoichiometric matrix and the rate vector:

$$\frac{d}{dt}C(t) = \alpha^T r \quad (3.16)$$

```

1 import sympy as sp
2
3 # Define symbols
4 A, B, C, D, E = sp.symbols('A B C D E')
5 k1, k2, k3, k4, k5, k6 = sp.symbols('k1 k2 k3 k4 k5 k6')
6
7 # Define rate equations
8 r = sp.Matrix([
9     k1*A**2,

```

```

10     k2*B**2*C,
11     k3*A*B,
12     k4*D**2,
13     k5*B*D,
14     k6*E
15 ])
16
17 # Stoichiometric matrix
18 alpha = sp.Matrix([[-2, 2, 1, 0, 0],
19                    [ 2, -2, -1, 0, 0],
20                    [-1, -1, 0, 2, 0],
21                    [ 1, 1, 0, -2, 0],
22                    [ 0, -1, 0, -1, 1],
23                    [ 0, 1, 0, 1, -1]])
24
25 # Compute concentration derivatives
26 dC_dt = alpha.T * r
27 sp.pprint(dC_dt)

```

Listing 3.1: SymPy code for computing concentration derivatives from a stoichiometric matrix and reaction rates

This example demonstrates how to construct and solve the reaction system symbolically using SymPy.

3.4 Matrix Method for Solving the Direct Problem

In chemical kinetics, when the reaction model can be expressed as a linear system of first-order differential equations, an analytical solution always exists. This framework is especially useful for sequences of elementary reactions, including cases with reversible or competitive steps. By leveraging matrix algebra, we can efficiently solve these coupled equations and obtain the concentration profiles of all species.

The general kinetic system for multi-step reactions can be written as:

$$\frac{d}{dt}\mathbf{C}(t) = \mathbf{K}\mathbf{C}(t), \quad (3.17)$$

where:

- $\mathbf{C}(t)$ is the vector of species concentrations,
- \mathbf{K} is the square rate constant matrix.

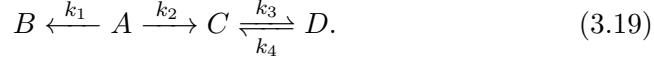
The solution to this system is given by the matrix exponential:

$$\mathbf{C}(t) = e^{\mathbf{K}t} \mathbf{C}_0, \quad (3.18)$$

with \mathbf{C}_0 representing the initial concentration vector.

3.4.1 Example: A Consecutive-Competitive Reaction with a Reversible Step

Consider the following kinetic scheme:



This reaction leads to the system of coupled differential equations:

$$\frac{d}{dt} \begin{bmatrix} C_A(t) \\ C_B(t) \\ C_C(t) \\ C_D(t) \end{bmatrix} = \begin{bmatrix} -(k_1 + k_2) & 0 & 0 & 0 \\ k_1 & 0 & 0 & 0 \\ k_2 & 0 & -k_3 & k_4 \\ 0 & 0 & k_3 & -k_4 \end{bmatrix} \begin{bmatrix} C_A(t) \\ C_B(t) \\ C_C(t) \\ C_D(t) \end{bmatrix}. \quad (3.20)$$

Here, the rate constant matrix \mathbb{K} collects all the kinetic rate coefficients. The solution for the concentration vector is then:

$$\mathbb{C}(t) = e^{\mathbb{K}t} \mathbb{C}_0. \quad (3.21)$$

3.4.2 Eigenvalue Decomposition Method

A common strategy to compute the matrix exponential $e^{\mathbb{K}t}$ is to perform an eigenvalue decomposition of \mathbb{K} . We write:

$$\mathbb{K} = \mathbb{X} \Lambda \mathbb{X}^{-1}, \quad (3.22)$$

where:

- \mathbb{X} is the matrix of eigenvectors,
- Λ is a diagonal matrix with the eigenvalues λ_i of \mathbb{K} along its diagonal.

Using this decomposition, the matrix exponential becomes:

$$e^{\mathbb{K}t} = \mathbb{X} e^{\Lambda t} \mathbb{X}^{-1}, \quad (3.23)$$

where

$$e^{\Lambda t} = \begin{bmatrix} e^{\lambda_0 t} & 0 & \dots & 0 \\ 0 & e^{\lambda_1 t} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & e^{\lambda_n t} \end{bmatrix}. \quad (3.24)$$

Thus, the concentration profile at time t is:

$$\mathbb{C}(t) = \mathbb{X} e^{\Lambda t} \mathbb{X}^{-1} \mathbb{C}_0. \quad (3.25)$$

This method provides an analytical solution as long as the eigenvalues (and corresponding eigenvectors) of \mathbb{K} can be computed.

3.4.3 Eigenvector Ordering Considerations

In some computational environments, the order of eigenvectors returned from an eigenvalue decomposition may not match the order of the corresponding eigenvalues. Since the eigenvectors satisfy:

$$\mathbb{K}^{(i)} = \lambda_i^{(i)}, \quad (3.26)$$

it is essential to ensure that the columns of \mathbb{X} (the eigenvectors) are arranged in the same order as the eigenvalues in Λ . This is usually verified by checking that:

$$\mathbb{K} \mathbb{X} = \mathbb{X} \Lambda. \quad (3.27)$$

Note: Although libraries such as SymPy and NumPy typically return eigenvectors in a consistent order, including a discussion on eigenvector ordering is still useful. It helps students understand potential pitfalls when using other software or custom routines where ordering might be arbitrary.

3.4.4 Application and Numerical Evaluation

Once an analytical expression for $\mathbb{C}(t)$ is established, it can be applied to:

- Plot the time evolution of species concentrations,
- Analyze the reaction kinetics under various sets of rate constants,
- Compare model predictions with experimental data.

Reminder: The order of matrix multiplication is crucial in these computations since matrix products are generally non-commutative.

This matrix-based method thus provides a powerful and systematic approach for solving complex kinetic problems, especially when direct integration of the differential equations is cumbersome.

3.4.5 Example Calculation using Python

Using SymPy, the computation of the solution follows:

```

1 import sympy as sp
2
3 # Define symbols
4 t = sp.Symbol('t')
5 C0 = sp.Matrix([sp.Symbol('C0_1'), 0, 0, 0])
6 k1, k2, k3, k4 = sp.symbols('k1_k2_k3_k4')
7
8 # Define the rate constant matrix
9 K = sp.Matrix([[-(k1 + k2), 0, 0, 0],
10                [k1, 0, 0, 0],

```

```

11         [k2, 0, -k3, k4],
12         [0, 0, k3, -k4]])
13
14 # Compute eigenvalues and eigenvectors
15 eigen_data = K.eigenvecs()
16 eigenvalues = [ev[0] for ev in eigen_data]
17 eigenvectors = []
18 for ev in eigen_data:
19     for vec in ev[2]: # Include all eigenvectors for degenerate
20         cases
21         eigenvectors.append(vec)
22
23 # Ensure the eigenvector matrix is square
24 if len(eigenvectors) == K.shape[0]:
25     X = sp.Matrix.hstack(*eigenvectors) # Construct eigenvector
26     matrix
27     e_Lambda_t = sp.diag(*[sp.exp(ev[0] * t) for ev in eigen_data
28         for _ in ev[2]])
29
30 # Compute matrix exponential using similarity transformation
31 exp_Kt = X * e_Lambda_t * X.inv()
32
33 # Compute concentration evolution
34 C_t = exp_Kt * C0
35
36 # Display final concentration evolution
37 print("\nConcentration Evolution:")
38 sp.pprint(C_t)
39 else:
40     print("Error: Eigenvector matrix is not square. Direct
41         diagonalization is not possible.")

```

Listing 3.2: SymPy code for computing concentration evolution using matrix exponentials via eigen-decomposition of the rate constant matrix

This method allows solving systems where analytical expressions are required, but also facilitates numerical computation using **SciPy** or **NumPy**.

3.5 Approximate Methods of Chemical Kinetics

In many complex kinetic mechanisms, direct analytical solutions may be difficult to obtain. Approximate methods—such as the steady-state concentration method—offer a practical approach by reducing differential equations to algebraic ones, thereby simplifying the analysis of reaction kinetics.

3.5.1 The Steady-State Concentration Method

In chemical kinetics, multi-step reaction mechanisms often involve short-lived reactive intermediates that complicate the analysis. One powerful simplification technique is the **steady-state approximation**. This approach assumes that the concentration of an intermediate remains nearly

constant over most of the reaction because its rate of formation is almost exactly balanced by its rate of consumption.

To illustrate this concept, consider the following consecutive reaction:



where

- k_1 and k_2 are the rate constants for the reversible conversion between A and the intermediate B ,
- k_3 is the rate constant for the conversion of B into the product C .

Applying the Steady-State Approximation If the intermediate B is very reactive and is consumed much faster than it is formed (i.e., when $k_3 \gg k_1$), its concentration remains low and changes only negligibly over time. In this situation, one sets the time derivative of B 's concentration to zero:

$$\frac{dC_B}{dt} = k_1 C_A - k_2 C_B - k_3 C_B = 0. \quad (3.29)$$

Solving for C_B yields:

$$C_B = \frac{k_1}{k_2 + k_3} C_A. \quad (3.30)$$

Since the product C is formed from B at a rate proportional to k_3 , the steady-state rate of product formation becomes

$$r_P = k_3 C_B = \frac{k_1 k_3}{k_2 + k_3} C_A. \quad (3.31)$$

Validation via Simulation To confirm the validity of the steady-state approximation, its prediction is compared with numerical solutions of the full kinetic equations. For example, consider the following parameters:

$$k_1 = 0.15 \text{ s}^{-1}, \quad k_2 = 0.07 \text{ s}^{-1}, \quad k_3 = 10 \text{ s}^{-1}, \quad A_0 = 0.1 \text{ mol/L}. \quad (3.32)$$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import solve_ivp
4
5 # Define rate constants and initial concentration
6 k1, k2, k3 = 0.15, 0.07, 10
7 A0 = 0.1
8
9 # Define the system of ODEs
10 def reaction_system(t, y):
11     A, B = y

```

```
12     dA_dt = -k1 * A
13     dB_dt = k1 * A - (k2 + k3) * B
14     return [dA_dt, dB_dt]
15
16 # Solve the ODEs over the interval from 0 to 2 seconds
17 t_span = [0, 2]
18 t_eval = np.linspace(0, 2, 100)
19 sol = solve_ivp(reaction_system, t_span, [A0, 0], t_eval=t_eval)
20
21 # Extract the concentration profiles for A and B
22 A, B = sol.y
23 t = sol.t
24
25 # Compute the steady-state prediction for the rate of product
    formation
26 rP_steady = (k1 * k3 / (k2 + k3)) * A
27
28 # Plot the numerical and steady-state predicted rates
29 plt.figure(figsize=(8, 5))
30 plt.plot(t, k3 * B, label=r'$k_3B(t)$', color='black')
31 plt.plot(t, rP_steady, '--', label=r'$\frac{k_1k_3}{k_2+k_3}A(
    t)$', color='black')
32 plt.xlabel('Time(s)')
33 plt.ylabel('Rate(mol/L/s)')
34 plt.legend()
35 plt.grid()
36 plt.show()
```

Listing 3.3: Python simulation demonstrating the steady-state approximation.

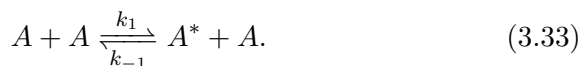
The simulation shows that the numerical solution for $k_3B(t)$ (solid black line) aligns very well with the steady-state prediction (dashed line). The curves converge within approximately **0.5 seconds**, confirming that the steady-state approximation is valid under these conditions.

Application of the Steady-State Principle

The steady-state approximation is a powerful tool in chemical kinetics that greatly simplifies the analysis of complex reaction mechanisms. By assuming that the concentration of a reactive intermediate quickly reaches and then remains nearly constant (after an initial transient period), we can replace a set of differential equations with simpler algebraic expressions. This approach is not only conceptually insightful but also often provides results that agree well with full numerical simulations.

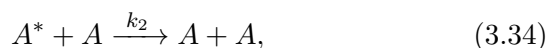
To illustrate the method, consider the classic Lindemann mechanism for unimolecular decomposition reactions. In this mechanism, the overall conversion of reactant A to product P occurs via an excited intermediate A^* through the following steps:

1. **Excitation:** A bimolecular collision between two A molecules generates the excited species:



2. **Follow-up Reactions:** Once formed, A^* can either be deexcited or proceed to form product:

- *Deexcitation via collision:*



- *Conversion to Product:*



Applying the steady-state assumption to the intermediate A^* (i.e., $\frac{d[A^*]}{dt} \approx 0$) gives us the condition

$$k_1[A]^2 = (k_2[A] + k_3)[A^*], \quad (3.36)$$

which can be rearranged to express the steady-state concentration of A^* in terms of $[A]$:

$$[A^*]_{ss} = \frac{k_1[A]^2}{k_2[A] + k_3}. \quad (3.37)$$

This result reveals two important limiting cases:

- **Low Pressure (or Low $[A]$):** When $k_2[A] \ll k_3$, the denominator is dominated by k_3 , and the overall reaction rate exhibits second-order behavior in $[A]$.
- **High Pressure (or High $[A]$):** When $k_2[A] \gg k_3$, the term $k_2[A]$ controls the rate, leading to first-order kinetics with respect to $[A]$ because the frequent deexcitation collisions effectively reduce the concentration of the excited intermediate.

Note: The steady-state approximation transforms a complex system of differential equations into more manageable algebraic expressions, providing clear insight into the kinetics of the reaction mechanism.

The following Python code demonstrates the application of the steady-state approximation to the Lindemann mechanism. The code is organized into three main parts:

1. **Symbolic Derivation:** We use `sympy` to derive the steady-state expression for $[A^*]$.

2. **Numerical Integration:** The full set of ordinary differential equations (ODEs) for the system is solved numerically using `scipy`'s `solve_ivp`.
3. **Comparison:** The steady-state value of $[A^*]$ is evaluated at each time point and compared with the numerical solution.

```

1 import sympy as sp
2 import numpy as np
3 from scipy.integrate import solve_ivp
4 import matplotlib.pyplot as plt
5
6 # -----
7 # 1) SYMBOLIC DERIVATION: Solve for [A*] under the steady-state
8 #    condition.
9 #    The steady-state condition is: d[A*]/dt = 0, i.e.,
10 #    k1*[A]^2 = (k2*[A] + k3)*[A*]
11 # -----
12 # Define symbolic variables
13 A_sym, Astar_sym = sp.symbols('A Astar', positive=True)
14 k1_sym, k2_sym, k3_sym = sp.symbols('k1 k2 k3', positive=True)
15
16 # Write the steady-state equation for A*
17 eq_ss = sp.Eq(k1_sym*A_sym**2, Astar_sym * (k2_sym*A_sym + k3_sym)
18              )
19 # Solve for [A*]
20 Astar_ss_expr = sp.solve(eq_ss, Astar_sym)[0]
21
22 print("Symbolic steady-state solution for [A*]:")
23 sp.pprint(Astar_ss_expr)
24 # -----
25 # 2) NUMERICAL INTEGRATION: Solve the full ODE system.
26 #
27 #    d[A]/dt      = -k1*A^2 + k2*A*A*
28 #    d[A*]/dt     = k1*A^2 - k2*A*A* - k3*A*
29 #    d[P]/dt      = k3*A*
30 # -----
31
32 # Define numerical values for the rate constants
33 k1_val = 1.0
34 k2_val = 10.0
35 k3_val = 1.0
36
37 # Initial conditions: [A]=1.0, [A*]=0.0, [P]=0.0
38 A0 = 1.0
39 Astar0 = 0.0
40 P0 = 0.0
41
42 def lindemann_odes(t, y):
43     A, Astar, P = y
44     dA_dt      = -k1_val*A**2 + k2_val*A*Astar
45     dAstar_dt  = k1_val*A**2 - k2_val*A*Astar - k3_val*Astar

```

```

46     dP_dt      = k3_val*Astar
47     return [dA_dt, dAstar_dt, dP_dt]
48
49 # Time span and points for evaluation
50 t_span = (0, 5)
51 t_eval = np.linspace(0, 5, 200)
52 sol = solve_ivp(lindemann_odes, t_span, [A0, Astar0, P0], t_eval=
    t_eval)
53 t_vals = sol.t
54 A_vals = sol.y[0]
55 Astar_vals = sol.y[1]
56 P_vals = sol.y[2]
57
58 # -----
59 # 3) EVALUATE THE STEADY-STATE EXPRESSION:
60 #     Convert the symbolic expression for [A*] into a function and
61 #     evaluate it using the numerical [A] from the ODE solution.
62 # -----
63
64 Astar_ss_func = sp.lambdify(
65     (A_sym, k1_sym, k2_sym, k3_sym),
66     Astar_ss_expr,
67     'numpy'
68 )
69 Astar_ss_vals = Astar_ss_func(A_vals, k1_val, k2_val, k3_val)
70
71 # -----
72 # 4) PLOTTING THE RESULTS:
73 #     Compare the numerical solution for [A*] with the steady-state
74 #     approximation.
75 # -----
76 plt.figure(figsize=(8,6))
77 plt.plot(t_vals, A_vals,      'b-', label='[A](t)␣(numeric)')
78 plt.plot(t_vals, Astar_vals,  'r-', label='[A*](t)␣(numeric)')
79 plt.plot(t_vals, P_vals,      'g-', label='[P](t)␣(numeric)')
80 plt.plot(t_vals, Astar_ss_vals, 'r--', label='[A*](t)␣steady-state'
    )
81 plt.xlabel('Time')
82 plt.ylabel('Concentration')
83 plt.title('Lindemann␣Mechanism:␣Comparison␣with␣Steady-State␣
    Approximation')
84 plt.legend()
85 plt.grid(True)
86 plt.show()

```

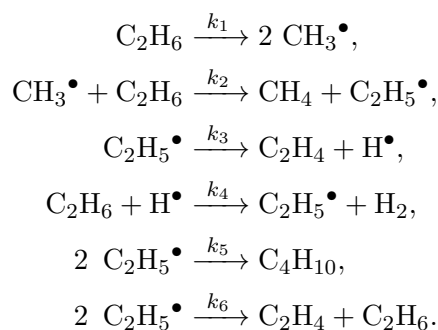
Listing 3.4: Python code for applying the steady-state approximation to the Lindemann mechanism.

This example demonstrates how the steady-state approximation not only simplifies the derivation of rate expressions but also accurately describes the system behavior across different regimes. At low pressures (or low $[A]$), the reaction follows second-order kinetics, while at high pressures (or high $[A]$) the mechanism effectively exhibits first-order kinetics due to the dominance

of deexcitation collisions.

Application to Ethane Thermal Cracking

In the thermal cracking of ethane, the reaction mechanism is often simplified to the following scheme:



In this mechanism, the radicals CH_3^\bullet , $\text{C}_2\text{H}_5^\bullet$, and H^\bullet are extremely reactive. We therefore assume that these intermediates rapidly achieve steady state. That is, their net production rates quickly become negligibly small compared to those of the stable species, so that we can set their time derivatives to zero. This steady-state approximation (SSA) considerably simplifies the analysis.

Steady-State Concentrations of the Radical Intermediates 1. The CH_3^\bullet Radical:

The CH_3^\bullet radical is produced in Reaction (1) and consumed in Reaction (2). The corresponding rate contributions are:

$$\text{Production: } 2k_1[\text{C}_2\text{H}_6], \quad \text{Consumption: } k_2[\text{CH}_3^\bullet][\text{C}_2\text{H}_6].$$

At steady state, the net rate is zero:

$$2k_1[\text{C}_2\text{H}_6] - k_2[\text{CH}_3^\bullet][\text{C}_2\text{H}_6] = 0.$$

Dividing both sides by $[\text{C}_2\text{H}_6]$ (assuming it is nonzero) immediately yields

$$[\text{CH}_3^\bullet] = \frac{2k_1}{k_2}. \quad (3.38)$$

2. The $\text{C}_2\text{H}_5^\bullet$ Radical:

It is convenient to introduce the following shorthand notation:

$$C_1 = [\text{C}_2\text{H}_6], \quad C_4 = [\text{C}_2\text{H}_5^\bullet], \quad C_5 = [\text{H}^\bullet].$$

The $\text{C}_2\text{H}_5^\bullet$ radical is generated by:

- Reaction (2): at a rate

$$r_2 = k_2 [\text{CH}_3^\bullet] C_1.$$

Substituting the expression from (3.38) gives

$$r_2 = k_2 \left(\frac{2k_1}{k_2} \right) C_1 = 2k_1 C_1.$$

- Reaction (4): at a rate

$$r_4 = k_4 C_1 C_5.$$

The radical is consumed by:

- Reaction (3): at a rate

$$r_3 = k_3 C_4.$$

- The dimerization reactions, (5) and (6), which together remove $\text{C}_2\text{H}_5^\bullet$ at a rate

$$2(k_5 + k_6)C_4^2.$$

Thus, the steady-state balance for $\text{C}_2\text{H}_5^\bullet$ is

$$2k_1 C_1 + k_4 C_1 C_5 - k_3 C_4 - 2(k_5 + k_6)C_4^2 = 0.$$

3. The H^\bullet Radical:

The hydrogen radical is produced in Reaction (3) and consumed in Reaction (4):

$$k_3 C_4 - k_4 C_1 C_5 = 0.$$

Solving for C_5 gives

$$C_5 = \frac{k_3 C_4}{k_4 C_1}. \quad (3.39)$$

Eliminating C_5 in the $\text{C}_2\text{H}_5^\bullet$ balance:

Substitute the expression for C_5 from (3.39) into the steady-state equation for $\text{C}_2\text{H}_5^\bullet$:

$$2k_1 C_1 + k_4 C_1 \left(\frac{k_3 C_4}{k_4 C_1} \right) - k_3 C_4 - 2(k_5 + k_6)C_4^2 = 0.$$

Notice that the term $k_4 C_1 (k_3 C_4 / (k_4 C_1))$ simplifies to $k_3 C_4$ and cancels with the $-k_3 C_4$. This leaves

$$2k_1 C_1 = 2(k_5 + k_6)C_4^2,$$

so that

$$[\text{C}_2\text{H}_5^\bullet] = C_4 = \sqrt{\frac{k_1 C_1}{k_5 + k_6}} = \sqrt{\frac{k_1 [\text{C}_2\text{H}_6]}{k_5 + k_6}}. \quad (3.40)$$

Using Equation (3.39) together with the result for C_4 , the steady-state concentration of the H^\bullet radical becomes

$$[\text{H}^\bullet] = C_5 = \frac{k_3}{k_4} \sqrt{\frac{k_1}{(k_5 + k_6)C_1}} = \frac{k_3}{k_4} \sqrt{\frac{k_1}{(k_5 + k_6)[\text{C}_2\text{H}_6]}}. \quad (3.41)$$

In summary, the steady-state concentrations of the radical intermediates are:

$$[\text{CH}_3^\bullet] = \frac{2k_1}{k_2}, \quad [\text{C}_2\text{H}_5^\bullet] = \sqrt{\frac{k_1[\text{C}_2\text{H}_6]}{k_5 + k_6}}, \quad [\text{H}^\bullet] = \frac{k_3}{k_4} \sqrt{\frac{k_1}{(k_5 + k_6)[\text{C}_2\text{H}_6]}}.$$

Differential Equations for the Stable Species With the radical concentrations now expressed in terms of the stable species, we can write the differential equations for the species whose concentrations change significantly over time.

Ethane (C_2H_6):

Ethane is consumed in several reactions:

- Reaction (1): $r_1 = k_1[\text{C}_2\text{H}_6]$,
- Reaction (2): $r_2 = 2k_1[\text{C}_2\text{H}_6]$ (using the result for $[\text{CH}_3^\bullet]$ from (3.38)),
- Reaction (4): $r_4 = k_4[\text{C}_2\text{H}_6][\text{H}^\bullet] = k_3\sqrt{\frac{k_1[\text{C}_2\text{H}_6]}{k_5 + k_6}}$ (after substituting (3.41)).

It is regenerated in Reaction (6) at a rate

$$r_6 = k_6[\text{C}_2\text{H}_5^\bullet]^2 = \frac{k_1k_6}{k_5 + k_6}[\text{C}_2\text{H}_6].$$

Thus, the net rate of consumption of ethane is

$$\begin{aligned} \frac{d[\text{C}_2\text{H}_6]}{dt} &= -r_1 - r_2 - r_4 + r_6 \\ &= -k_1[\text{C}_2\text{H}_6] - 2k_1[\text{C}_2\text{H}_6] - k_3\sqrt{\frac{k_1[\text{C}_2\text{H}_6]}{k_5 + k_6}} + \frac{k_1k_6}{k_5 + k_6}[\text{C}_2\text{H}_6] \\ &= -\left(3k_1 - \frac{k_1k_6}{k_5 + k_6}\right)[\text{C}_2\text{H}_6] - k_3\sqrt{\frac{k_1[\text{C}_2\text{H}_6]}{k_5 + k_6}}. \end{aligned} \quad (3.42)$$

Methane (CH_4):

Methane is produced solely in Reaction (2), so its rate of formation is

$$\frac{d[\text{CH}_4]}{dt} = 2k_1[\text{C}_2\text{H}_6]. \quad (3.43)$$

Ethylene (C_2H_4):

Ethylene is formed via two pathways:

- Reaction (3): $r_3 = k_3 \sqrt{\frac{k_1 [\text{C}_2\text{H}_6]}{k_5 + k_6}}$,
- Reaction (6): $r_6 = \frac{k_1 k_6}{k_5 + k_6} [\text{C}_2\text{H}_6]$.

Thus,

$$\frac{d[\text{C}_2\text{H}_4]}{dt} = k_3 \sqrt{\frac{k_1 [\text{C}_2\text{H}_6]}{k_5 + k_6}} + \frac{k_1 k_6}{k_5 + k_6} [\text{C}_2\text{H}_6]. \quad (3.44)$$

Hydrogen (H_2):

Hydrogen is produced in Reaction (4):

$$\frac{d[\text{H}_2]}{dt} = k_3 \sqrt{\frac{k_1 [\text{C}_2\text{H}_6]}{k_5 + k_6}}. \quad (3.45)$$

Butane (C_4H_{10}):

Butane is formed by the dimerization of two $\text{C}_2\text{H}_5^\bullet$ radicals in Reaction (5):

$$\frac{d[\text{C}_4\text{H}_{10}]}{dt} = \frac{k_1 k_5}{k_5 + k_6} [\text{C}_2\text{H}_6]. \quad (3.46)$$

For convenience, we define the following constants:

$$\alpha = 3k_1 - \frac{k_1 k_6}{k_5 + k_6}, \quad \beta = k_3 \sqrt{\frac{k_1}{k_5 + k_6}}, \quad (3.47)$$

so that Equation (3.42) may be compactly written as

$$\frac{d[\text{C}_2\text{H}_6]}{dt} = -\alpha [\text{C}_2\text{H}_6] - \beta \sqrt{[\text{C}_2\text{H}_6]}. \quad (3.48)$$

Once the time evolution of $[\text{C}_2\text{H}_6]$ is determined (e.g., by numerical integration of (3.48)), the product concentrations can be obtained from the integrated forms:

$$[\text{CH}_4](t) = 2k_1 \int_0^t [\text{C}_2\text{H}_6](u) du, \quad (3.49)$$

$$[\text{C}_2\text{H}_4](t) = \beta \int_0^t \sqrt{[\text{C}_2\text{H}_6](u)} du + \frac{k_1 k_6}{k_5 + k_6} \int_0^t [\text{C}_2\text{H}_6](u) du, \quad (3.50)$$

$$[\text{H}_2](t) = \beta \int_0^t \sqrt{[\text{C}_2\text{H}_6](u)} du, \quad (3.51)$$

$$[\text{C}_4\text{H}_{10}](t) = \frac{k_1 k_5}{k_5 + k_6} \int_0^t [\text{C}_2\text{H}_6](u) du. \quad (3.52)$$

Symbolic Verification via Python/Sympy To confirm these results, the following Python script using the Sympy library sets up the reaction rates and stoichiometric matrix, applies the steady-state approximation to the radicals, and solves for their concentrations symbolically:

```

1 import sympy as sp
2
3 # 1) Define symbols for rate constants and concentrations.
4 #   Notation:
5 #       C1 : [C2H6]
6 #       C2 : [CH3*]
7 #       C3 : [CH4]
8 #       C4 : [C2H5*]
9 #       C5 : [H*]
10 #       C6 : [C2H4]
11 #       C7 : [H2]
12 #       C8 : [C4H10]
13 k1, k2, k3, k4, k5, k6 = sp.symbols('k1_k2_k3_k4_k5_k6', positive=
    True)
14 C1, C2, C3, C4, C5, C6, C7, C8 = sp.symbols('C1_C2_C3_C4_C5_C6_C7_
    C8', positive=True)
15
16 # 2) Define the reaction rates.
17 # Reaction scheme:
18 # R1: C2H6 -> 2 CH3*
19 # R2: CH3* + C2H6 -> CH4 + C2H5*
20 # R3: C2H5* -> C2H4 + H*
21 # R4: C2H6 + H* -> C2H5* + H2
22 # R5: 2 C2H5* -> C4H10
23 # R6: 2 C2H5* -> C2H4 + C2H6
24 r1 = k1 * C1                                # Reaction 1
25 r2 = k2 * C2 * C1                            # Reaction 2
26 r3 = k3 * C4                                # Reaction 3
27 r4 = k4 * C1 * C5                            # Reaction 4
28 r5 = k5 * C4**2                             # Reaction 5
29 r6 = k6 * C4**2                             # Reaction 6
30 r_vec = sp.Matrix([r1, r2, r3, r4, r5, r6])
31
32 # 3) Define the stoichiometric matrix.
33 # Rows correspond to the species:
34 # 0: C2H6, 1: CH3*, 2: CH4, 3: C2H5*, 4: H*, 5: C2H4, 6: H2, 7:
    C4H10
35 alpha = sp.Matrix([
36     [-1, -1, 0, -1, 0, +1], # C2H6
37     [2, -1, 0, 0, 0, 0], # CH3*
38     [0, +1, 0, 0, 0, 0], # CH4
39     [0, +1, -1, +1, -2, -2], # C2H5*
40     [0, 0, +1, -1, 0, 0], # H*
41     [0, 0, +1, 0, 0, +1], # C2H4
42     [0, 0, 0, +1, 0, 0], # H2
43     [0, 0, 0, 0, +1, 0] # C4H10
44 ])
45

```

```

46 # 4) Compute the net production rates for each species.
47 net_rates = alpha * r_vec
48 # The entries of net_rates correspond to:
49 #   d[C2H6]/dt, d[CH3*]/dt, d[CH4]/dt, d[C2H5*]/dt, d[H*]/dt, d[
      C2H4]/dt, d[H2]/dt, d[C4H10]/dt
50
51 # 5) Apply the steady-state approximation for the radicals:
52 #   Set net rates for CH3* (row 1), C2H5* (row 3), and H* (row 4)
      equal to zero.
53 eq_CH3 = sp.Eq(net_rates[1], 0) # CH3*
54 eq_C2H5 = sp.Eq(net_rates[3], 0) # C2H5*
55 eq_H = sp.Eq(net_rates[4], 0) # H*
56 sol_radicals = sp.solve((eq_CH3, eq_C2H5, eq_H), (C2, C4, C5),
      dict=True)
57
58 print("Steady-state solutions for the radical species:")
59 sp.pprint(sol_radicals)
60 print("\n")
61 # Expected solutions:
62 #   [CH3*] = 2*k1/k2,
63 #   [C2H5*] = sqrt(k1 * C1/(k5 + k6)),
64 #   [H*] = (k3/k4) * sqrt(k1/(C1*(k5+k6))).
65
66 # 6) Substitute the steady-state radical expressions into the net
      production rates
67 net_rates_ss = sp.simplify(net_rates.subs(sol_radicals[0]))
68
69 # 7) Display the differential equations for all species.
70 species = {
71     0: r"[C2H6]",
72     1: r"[CH3*]",
73     2: r"[CH4]",
74     3: r"[C2H5*]",
75     4: r"[H*]",
76     5: r"[C2H4]",
77     6: r"[H2]",
78     7: r"[C4H10]"
79 }
80
81 print("Differential equations after applying the steady-state
      approximation:\n")
82 for i in range(net_rates_ss.shape[0]):
83     print(f"d{species[i]}/dt=")
84     sp.pprint(net_rates_ss[i])
85     print()

```

Listing 3.5: Symbolic Verification of the Steady-State Approximation in Ethane Thermal Cracking

When you execute this script (for example, in a Jupyter notebook),

Sympy returns the symbolic steady-state solutions:

$$[\text{CH}_3\bullet] = \frac{2k_1}{k_2}, \quad [\text{C}_2\text{H}_5\bullet] = \sqrt{\frac{k_1 C_1}{k_5 + k_6}}, \quad [\text{H}\bullet] = \frac{k_3}{k_4} \sqrt{\frac{k_1}{(k_5 + k_6) C_1}},$$

confirming our analytical derivations.

Solving for the Ethane Concentration $[\text{C}_2\text{H}_6](t)$ Recall that the differential equation for ethane is given by

$$\frac{d[\text{C}_2\text{H}_6]}{dt} = -\alpha[\text{C}_2\text{H}_6] - \beta\sqrt{[\text{C}_2\text{H}_6]},$$

where

$$\alpha = 3k_1 - \frac{k_1 k_6}{k_5 + k_6}, \quad \beta = k_3 \sqrt{\frac{k_1}{k_5 + k_6}}.$$

To obtain an analytical expression for $[\text{C}_2\text{H}_6](t)$, we perform the substitution

$$u(t) = \sqrt{[\text{C}_2\text{H}_6](t)} \implies [\text{C}_2\text{H}_6](t) = u(t)^2.$$

Differentiating yields

$$\frac{d[\text{C}_2\text{H}_6]}{dt} = 2u(t) \frac{du}{dt}.$$

Substituting into the ODE gives

$$2u \frac{du}{dt} = -\alpha u^2 - \beta u.$$

Dividing by u (assuming $u > 0$) we have

$$\frac{du}{dt} = -\frac{\alpha}{2}u - \frac{\beta}{2}.$$

This first-order linear ODE for $u(t)$ can be solved symbolically using Python/Sympy. The following script performs the integration:

```

1 import sympy as sp
2
3 # Define symbols and the function for ethane concentration C(t)
4 t, C0, alpha, beta = sp.symbols('t C0 alpha beta', positive=True)
5 C = sp.Function('C')(t)
6
7 # Original ODE: dC/dt = -alpha * C - beta * sqrt(C)
8 ode = sp.Eq(sp.diff(C, t), -alpha * C - beta * sp.sqrt(C))
9
10 # Substitute: let u(t) = sqrt(C(t)), so that C(t) = u(t)**2.
11 # Then, dC/dt = 2*u(t)*du/dt.
12 # Substituting into the ODE:
13 # 2*u*du/dt = -alpha*u**2 - beta*u.
14 # Dividing by u (assuming u>0) yields:

```

```

15 # du/dt = - (alpha/2)*u - beta/2.
16 u = sp.Function('u')(t)
17 ode_u = sp.Eq(sp.diff(u, t), -alpha/2 * u - beta/2)
18
19 # Solve the ODE for u(t) with the initial condition: u(0) = sqrt(
    C0)
20 solution_u = sp.dsolve(ode_u, u, ics={u.subs(t, 0): sp.sqrt(C0)})
21
22 # The solution for C(t) is then [u(t)]**2.
23 solution_C = sp.simplify(solution_u.rhs**2)
24
25 print("Ethane concentration [C2H6](t) in terms of alpha and beta:")
26 sp.pprint(solution_C)

```

Listing 3.6: Symbolic Evaluation of Ethane Concentration

When you execute this script, it outputs the expression for $[C_2H_6](t)$ in terms of α , β , and the initial concentration C_0 . This symbolic solution clearly illustrates how the concentration decays over time under the influence of both first-order (linear) and square-root (nonlinear) terms.

Data for Rate Constants, Temperature, and Pressure Before we present the numerical simulation code, note that the rate constants in this mechanism are defined by Arrhenius-type expressions. For example:

$$\begin{aligned}
 k_1 &= 4.26 \times 10^{16} \exp\left(-\frac{44579}{T}\right), \\
 k_2 &= 1.65 \times 10^9 \left(\frac{T}{298}\right)^{4.25} \exp\left(-\frac{3890}{T}\right), \\
 k_3 &= 8.85 \times 10^{12} \exp\left(-\frac{19469}{T}\right), \\
 k_4 &= 1.71 \times 10^{12} \left(\frac{T}{298}\right)^{2.32} \exp\left(-\frac{3414}{T}\right),
 \end{aligned}$$

with k_5 and k_6 given as constants. In our simulation, the temperature is set to $T = 1100$ K and the pressure is $p = 2 \times 10^5$ Pa. These values are used to compute the rate constants as well as the initial ethane concentration, which (assuming ideal gas behavior) is calculated from:

$$C_{1,0} = \frac{p}{RT},$$

where $R = 8.3154$ J/(mol K). (Note that a unit conversion may be required depending on the concentration units used.)

Numerical Simulation and Kinetic Curves of the Stable Species

In addition to the symbolic analysis, we can numerically simulate the kinetic curves of the stable species (ethane, methane, ethylene, hydrogen, and butane) using Python. The following code uses `numpy`, `scipy`, and `matplotlib` to compute and plot these curves. The rate constants are defined as functions of temperature, and numerical integration (using the trapezoidal rule) is used to obtain the integrated concentrations.

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.integrate import solve_ivp
5
6 # Define rate constants as functions of temperature T
7 def rate_constants(T):
8     k1 = 4.26e16 * np.exp(-44579 / T)
9     k2 = 1.65e9 * (T / 298) ** 4.25 * np.exp(-3890 / T)
10    k3 = 8.85e12 * np.exp(-19469 / T)
11    k4 = 1.71e12 * (T / 298) ** 2.32 * np.exp(-3414 / T)
12    k5 = 1.15e13 # constant
13    k6 = 1.45e12 # constant
14    alpha = k1 * (3 * k5 + 2 * k6) / (k5 + k6)
15    beta = k3 * np.sqrt(k1 / (k5 + k6))
16    return k1, k2, k3, k4, k5, k6, alpha, beta
17
18 # Define a function to calculate [C2H6](t) from the analytical
    solution
19 def ethane_concentration(t, C1_0, alpha, beta):
20     # The analytical solution obtained by substituting  $u(t) = \sqrt{C(t)}$ 
    into the transformed ODE leads to:
21     return ((-beta/alpha) + np.exp(-alpha/2 * t)*np.sqrt(C1_0) +
22            np.exp(-alpha/2 * t)*(beta/alpha))**2
23
24 # Function to perform numerical integration (trapezoidal rule)
25 def integrate_species(t_vals, func, *args):
26     return np.array([np.trapz(func(t_vals[:i], *args), t_vals[:i])
27                        for i in range(1, len(t_vals)+1)])
28
29 # Simulation parameters
30 T = 1100 # Temperature in K
31 k1, k2, k3, k4, k5, k6, alpha, beta = rate_constants(T)
32 p = 2e5 # Pressure in Pa
33 R = 8.3154 # Gas constant in J/(mol K)
34 C1_0 = p / (R * T * 1e6) # Adjust units if necessary
35
36 time_eval = np.linspace(0, 0.5, 200) # Time in seconds
37 C1 = ethane_concentration(time_eval, C1_0, alpha, beta)
38 C3 = 2 * k1 * integrate_species(time_eval, ethane_concentration,
    C1_0, alpha, beta)
39
40 # For ethylene and hydrogen, we assume proper expressions based on
    Equations (\ref{eq:C2H4_int}) and (\ref{eq:H2_int})
41 C6 = beta * integrate_species(time_eval,

```

```

40         lambda u, C1_0, alpha, beta: np.sqrt(
41             ethane_concentration(u, C1_0, alpha, beta))
42             + (k6 / (k3**2)) * ethane_concentration(u, C1_0, alpha,
43                 beta),
44             C1_0, alpha, beta)
45 C7 = beta * integrate_species(time_eval,
46     lambda u, C1_0, alpha, beta: np.sqrt(
47         ethane_concentration(u, C1_0, alpha, beta)),
48         C1_0, alpha, beta)
49 C8 = (beta**2 * k5 / k3**2) * integrate_species(time_eval,
50     ethane_concentration, C1_0, alpha, beta)
51
52 # Plot the results
53 fig, ax1 = plt.subplots(figsize=(8, 6))
54 ax2 = ax1.twinx()
55 ax1.set_ylim([0, 2.5e-5]) # Left y-axis: Ethane, Ethylene,
56     Hydrogen
57 ax2.set_ylim([0, 1e-6]) # Right y-axis: Methane, Butane
58
59 ax1.plot(time_eval, C1, label='Ethane(t)', color='black',
60     linestyle='-')
61 ax1.plot(time_eval, C6, label='Ethylene(t)', color='black',
62     linestyle='dashed')
63 ax1.plot(time_eval, C7, label='Hydrogen(t)', color='red',
64     linestyle='dashdot')
65
66 ax2.plot(time_eval, C3, label='Methane(t)', color='blue',
67     linestyle='dotted')
68 ax2.plot(time_eval, C8, label='Butane(t)', color='green',
69     linestyle=':')
70
71 ax1.set_xlabel('Time(s)')
72 ax1.set_ylabel('Concentration(Ethane, Ethylene, Hydrogen)')
73 ax2.set_ylabel('Concentration(Methane, Butane)')
74 ax1.set_title('Kinetic Curves at T=1100K')
75
76 ax1.legend(loc='upper_right')
77 ax2.legend(loc='upper_left')
78 ax1.grid()
79 plt.show()

```

Listing 3.7: Numerical Simulation of Kinetic Curves for Ethane Cracking

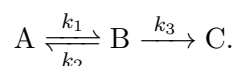
Summary By applying the steady-state approximation, we reduce a complex chain-reaction mechanism to algebraic expressions for the radicals and derive differential equations for the stable species. The symbolic verification using Python/Sympy confirms our analytical derivations, while the numerical simulation code demonstrates how the concentrations of ethane, methane, ethylene, hydrogen, and butane evolve with time at a given temperature. This integrated approach provides both theoretical insight and

practical tools for understanding the dynamics of ethane thermal cracking.

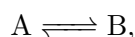
The Quasi-Equilibrium Approximation in Enzymatic Reaction Kinetics

In many chemical reactions—especially in enzyme catalysis—an intermediate species quickly reaches equilibrium with its precursor before slowly converting into the final product. This separation of time scales allows us to use the *quasi-equilibrium* (or *rapid equilibrium*) approximation to greatly simplify the kinetic analysis.

Example 1: Consecutive Reaction Consider the consecutive reaction



Here, the reversible conversion between A and B is much faster than the slow conversion of B to C . As a result, the intermediate B rapidly achieves an equilibrium concentration with A . For the fast, reversible step



the equilibrium constant is defined as

$$K = \frac{[B]_{\text{eq}}}{[A]_{\text{eq}}} = \frac{k_1}{k_2},$$

which implies that

$$[B] \approx \frac{k_1}{k_2} [A].$$

Since the formation of C is controlled by the slow step $B \rightarrow C$, the rate of product formation becomes

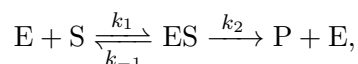
$$r_C = k_3[B] \approx k_3 \frac{k_1}{k_2} [A].$$

For this approximation to be valid, the slow step must satisfy

$$k_3 \ll k_1 \quad \text{and} \quad k_3 \ll k_2,$$

ensuring that B remains in quasi-equilibrium with A .

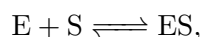
Example 2: Enzyme Kinetics Many enzyme-catalyzed reactions proceed via a fast formation of an enzyme–substrate complex, followed by a slower conversion to product. A classic mechanism is given by



where:

- E is the enzyme,
- S is the substrate,
- ES is the enzyme–substrate complex, and
- P is the product.

Under the quasi-equilibrium assumption, the binding of S to E rapidly reaches equilibrium. For the reaction



the equilibrium constant is

$$K_{\text{eq}} = \frac{[ES]}{[E][S]} = \frac{k_1}{k_{-1}},$$

which can be rearranged to express the concentration of the complex:

$$[ES] = \frac{k_1}{k_{-1}} [E] [S].$$

Since the total enzyme concentration is conserved,

$$[E]_0 = [E] + [ES],$$

the free enzyme concentration can be written as

$$[E] = [E]_0 - [ES].$$

Substituting this into the equilibrium expression, we have

$$[ES] = \frac{k_1}{k_{-1}} ([E]_0 - [ES]) [S].$$

Rearranging to solve for $[ES]$:

$$[ES] \left(1 + \frac{k_1}{k_{-1}} [S] \right) = \frac{k_1}{k_{-1}} [E]_0 [S],$$

$$[ES] = \frac{\frac{k_1}{k_{-1}} [E]_0 [S]}{1 + \frac{k_1}{k_{-1}} [S]}.$$

Defining the **Michaelis constant** as

$$K_M = \frac{k_{-1}}{k_1},$$

this expression simplifies to

$$[ES] = \frac{[E]_0 [S]}{K_M + [S]}.$$

Since the conversion of ES to P is the slow, rate-determining step, the rate of product formation is given by

$$r_P = k_2 [ES] = \frac{k_2 [E]_0 [S]}{K_M + [S]}.$$

This is the well-known **Michaelis–Menten equation**, which describes the dependence of the reaction rate on the substrate concentration. In particular:

- At low $[S]$ ($[S] \ll K_M$), the rate is approximately

$$r_P \approx \frac{k_2 [E]_0}{K_M} [S],$$

showing a first-order dependence on S .

- At high $[S]$ ($[S] \gg K_M$), the rate saturates at a maximum value:

$$r_{\max} = k_2 [E]_0,$$

implying zero-order kinetics.

The Michaelis–Menten equation can be linearized using the **Lineweaver–Burk transformation**:

$$\frac{1}{r_P} = \frac{1}{r_{\max}} + \frac{K_M}{r_{\max}} \frac{1}{[S]},$$

which is especially useful for determining the kinetic parameters K_M and r_{\max} from experimental data. Another useful rearrangement is:

$$r_P = r_{\max} - K_M \frac{r_P}{[S]}.$$

Numerical Verification of the Quasi-Equilibrium Approximation

The following Python script demonstrates how the intermediate B in the consecutive reaction stabilizes at its quasi-equilibrium concentration. The numerical solution is compared with the quasi-equilibrium approximation.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import solve_ivp
4
5 # Define rate constants
6 k1 = 1.0      # Forward rate constant for A -> B
```

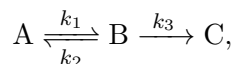
```

7 k2 = 0.8      # Reverse rate constant for B -> A
8 k3 = 0.01     # Slow rate constant for B -> C
9
10 # Initial concentrations (in M)
11 A0 = 0.1
12 B0 = 0.0
13 C0 = 0.0
14
15 # Define the system of ODEs
16 def reaction_system(t, y):
17     A, B, C = y
18     dA_dt = -k1 * A + k2 * B
19     dB_dt = k1 * A - (k2 + k3) * B
20     dC_dt = k3 * B
21     return [dA_dt, dB_dt, dC_dt]
22
23 # Time span for simulation
24 t_span = (0, 15)
25 t_eval = np.linspace(0, 15, 200)
26
27 # Solve the ODEs
28 solution = solve_ivp(reaction_system, t_span, [A0, B0, C0], t_eval
29                      =t_eval)
30 t = solution.t
31 A_sol, B_sol, C_sol = solution.y
32
33 # Compute the quasi-equilibrium approximation for [B]
34 B_quasi = (k1 / k2) * A_sol
35
36 # Plot the results
37 plt.figure(figsize=(8, 5))
38 plt.plot(t, B_sol, label=r'Numerical_{$B(t)$}', linestyle='-',
39          linewidth=2)
40 plt.plot(t, B_quasi, label=r'Quasi-Equilibrium_{$\frac{k_1}{k_2}A(t)$}',
41          linestyle='dotted', linewidth=2)
42 plt.xlabel('Time_{$s$}')
43 plt.ylabel('Concentration_{$M$}')
44 plt.title('Verification_of_the_Quasi-Equilibrium_Approximation')
45 plt.legend()
46 plt.grid(True)
47 plt.show()

```

Listing 3.8: Quasi-equilibrium principle simulation using SciPy and Matplotlib

This script simulates the reaction



and shows that $B(t)$ quickly reaches and stays near the quasi-equilibrium prediction:

$$[B] \approx \frac{k_1}{k_2} [A].$$

Numerical Verification of the Michaelis–Menten Equation The next Python script uses sympy to derive the Michaelis–Menten equation symbolically and then visualizes the reaction rate behavior using numerical values.

```

1 import sympy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Define symbolic variables
6 C_ES, C_E, C_E0, C_S, k1, k_neg1, k2 = sp.symbols('C_ES C_E C_E0 C_S k1 k_neg1 k2', positive=True, real=True)
7 KM = sp.symbols('KM', positive=True, real=True)
8
9 # Step 1: Equilibrium equation for ES complex formation
10 equilibrium_eq = sp.Eq(k1 * C_S * (C_E0 - C_ES), k_neg1 * C_ES)
11
12 # Step 2: Solve for C_ES
13 C_ES_solution = sp.solve(equilibrium_eq, C_ES)[0]
14
15 # Step 3: Define the rate equation
16 r_P_expr = k2 * C_ES_solution
17 sp.pprint(r_P_expr)
18
19 # Step 4: Express in terms of the Michaelis constant
20 r_P_MM = k2 * C_E0 * C_S / (C_S + k_neg1/k1)
21 r_P_final = sp.simplify(r_P_MM.subs(k_neg1/k1, KM))
22
23 print("The MichaelisMenten rate expression is:")
24 sp.pretty_print(r_P_final)
25
26 # -----
27 # Numerical Evaluation and Plotting
28 # -----
29
30 rP_func = sp.lambdify((C_S, C_E0, k2, KM), r_P_final, 'numpy')
31
32 C_E0_val = 1.214e-7 # Total enzyme concentration (M)
33 k2_val = 10 # Rate constant k2 (1/s)
34 KM_val = 4.815e-4 # Michaelis constant (M)
35
36 C_S_values = np.linspace(1e-5, 0.015, 100)
37 rP_values = rP_func(C_S_values, C_E0_val, k2_val, KM_val)
38
39 # Plot 1: Reaction Rate vs. Substrate Concentration
40 plt.figure(figsize=(6, 4))
41 plt.plot(C_S_values, rP_values, 'k-', linewidth=2)
42 plt.axhline(y=k2_val * C_E0_val, linestyle='--', color='gray',
43            label=r'$r_{\max}$')
44 plt.xlabel(r'Substrate Concentration, C_S (M)')
45 plt.ylabel(r'Reaction Rate, r_P (M/s)')
46 plt.title('Reaction Rate vs. Substrate Concentration')
47 plt.legend()

```

```

47 plt.grid(True)
48 plt.show()
49
50 # Plot 2: -LineweaverBurk Plot
51 inv_rP = 1 / rP_values
52 inv_CS = 1 / C_S_values
53 plt.figure(figsize=(6, 4))
54 plt.plot(inv_CS, inv_rP, 'k-', linewidth=2)
55 plt.xlabel(r'$1/C_S$ (1/M)')
56 plt.ylabel(r'$1/r_P$ (s/M)')
57 plt.title('-LineweaverBurk Plot')
58 plt.grid(True)
59 plt.show()

```

Listing 3.9: Derivation and visualization of Michaelis–Menten kinetics

This script first derives the Michaelis–Menten expression symbolically and then plots:

- The reaction rate r_P versus the substrate concentration C_S , and
- The corresponding Lineweaver–Burk plot, which linearizes the kinetic data.

Summary The quasi-equilibrium approximation is a powerful method for simplifying the analysis of multi-step reactions—especially in enzyme kinetics. By assuming that intermediate species rapidly equilibrate with their precursors, we obtain compact analytical expressions like the Michaelis–Menten equation that capture the key behavior of complex catalytic systems. Both theoretical derivation and numerical validation confirm the usefulness of this approximation.

3.6 Hands-On Activities

In this section, you will implement Python codes to analyze a simple consecutive reaction both numerically (using ODE integration) and analytically (using the matrix exponential). Consider the reaction:



Activity 1: Numerical Integration with SciPy

```

1 import numpy as np
2 from scipy.integrate import solve_ivp
3 import matplotlib.pyplot as plt
4
5 # Define the reaction rate constants
6 k1 = 1.0 # Rate constant for A -> B

```

```

7 k2 = 0.5 # Rate constant for B -> C
8
9 # Define the system of ODEs
10 def reaction_system(t, y, k1, k2):
11     A, B, C = y
12     dA_dt = -k1 * A
13     dB_dt = k1 * A - k2 * B
14     dC_dt = k2 * B
15     return [dA_dt, dB_dt, dC_dt]
16
17 # Initial concentrations
18 A0 = 1.0
19 B0 = 0.0
20 C0 = 0.0
21 y0 = [A0, B0, C0]
22
23 # Time span for the simulation
24 t_span = (0, 10)
25 t_eval = np.linspace(0, 10, 200)
26
27 # Solve the ODE system
28 solution = solve_ivp(reaction_system, t_span, y0, args=(k1, k2),
29                      t_eval=t_eval)
30
31 # Plot the results
32 plt.figure(figsize=(8, 5))
33 plt.plot(solution.t, solution.y[0], label='[A]')
34 plt.plot(solution.t, solution.y[1], label='[B]')
35 plt.plot(solution.t, solution.y[2], label='[C]')
36 plt.xlabel('Time')
37 plt.ylabel('Concentration')
38 plt.title('Numerical Integration of A -> B -> C')
39 plt.legend()
40 plt.grid(True)
41 plt.show()

```

Activity 2: Analytical Solution Using Matrix Exponential

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.linalg import expm
4
5 # Define the reaction rate constants
6 k1 = 1.0 # Rate constant for A -> B
7 k2 = 0.5 # Rate constant for B -> C
8
9 # Define the rate constant matrix for the system
10 K = np.array([[ -k1,  0,  0],
11               [ k1, -k2,  0],
12               [  0,  k2,  0]])
13
14 # Initial concentration vector

```

```

15 C0 = np.array([1.0, 0.0, 0.0])
16
17 # Time points for evaluation
18 t_points = np.linspace(0, 10, 200)
19 concentrations = np.zeros((3, len(t_points)))
20
21 # Compute the analytical solution using the matrix exponential at
22 # each time point
23 for i, t in enumerate(t_points):
24     exp_Kt = expm(K * t)
25     concentrations[:, i] = exp_Kt.dot(C0)
26
27 # Plot the analytical solution
28 plt.figure(figsize=(8, 5))
29 plt.plot(t_points, concentrations[0, :], label='[A] Analytical')
30 plt.plot(t_points, concentrations[1, :], label='[B] Analytical')
31 plt.plot(t_points, concentrations[2, :], label='[C] Analytical')
32 plt.xlabel('Time')
33 plt.ylabel('Concentration')
34 plt.title('Analytical Solution via Matrix Exponential')
35 plt.legend()
36 plt.grid(True)
37 plt.show()

```

In these activities, you observe the time evolution of species concentrations using both numerical integration and the analytical matrix exponential method. Compare the results to verify consistency between the two approaches.

3.7 Assignment

Compare the matrix method with direct numerical integration results for a three-step reaction:



Solve the system analytically using the matrix exponential method and numerically using ODE integration. Discuss the stability and steady-state behavior of the reaction network.

3.8 Conclusion

Multi-step reaction kinetics is a fundamental concept in chemical kinetics. The matrix method provides a powerful tool for solving complex reaction networks. By mastering both analytical and numerical techniques, researchers can develop predictive kinetic models that are useful in both research and industrial applications.