Chemical Kinetics 2302333

A Computational Approach with Python and SymPy

Viwat Vchirawongkwin

Contents

Listings

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.
- LaTeX output support for publishing high-quality equations.

```python
from sympy import symbols, Eq, solve, pprint

# Define symbols
k, A = symbols('k A')

# Define a simple rate equation
rate_eq = Eq(A - k*A, 1)
pprint(rate_eq)

# Solve for A
solution = solve(rate_eq, A)
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:

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

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:

```
pip install jupyter
```

Listing 1.2: Installing Jupyter Notebook

To launch Jupyter Notebook:

```
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:

```
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:

```python
def reaction_rate(k, A):
    return k * A

print(reaction_rate(0.1, 2.0))
```

Listing 1.5: Defining a Function

Defining Symbols and Equations:

```python
from sympy import symbols, Eq, pprint

# Define symbols
A, A0, k, t = symbols('A A_0 k t')

# Define a first-order rate law
rate_law = Eq(A, A0 * symbols('e')**(-k * t))
pprint(rate_law)
```

Listing 1.6: Defining Symbols and Equations

Substituting Values:

```python
# Substitute values into the rate law
A_initial = 1.0  # Initial concentration
k_value = 0.2    # Rate constant
t_value = 5.0    # Time

# Evaluate the equation
result = rate_law.subs({A0: A_initial, k: k_value, t: t_value})
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:

```python
from sympy import symbols, Function, Eq, dsolve, Derivative,
    pprint

```

```python
# Define the symbols and function
t, k, A0 = symbols('t k A0')
A = Function('A')

# Define the differential equation dA/dt = -k*A
rate_eq = Eq(Derivative(A(t), t), -k * A(t))
pprint(rate_eq)

# Solve the ODE with the initial condition A(0) = A0
solution = dsolve(rate_eq, A(t), ics={A(0): A0})
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.

```python
from sympy import symbols, sin, init_printing

# Call init_printing to enable pretty printing
init_printing()

x = symbols('x')
expression = sin(x) / x

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.

```python
from sympy import symbols

A, B, t, k = symbols('A B t k')
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., $A(t)$.

```python
from sympy import Function

A = Function('A')(t)
print(A)  # Output: A(t)
```

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

3. Eq

Purpose: Define symbolic equations.

```python
from sympy import Eq, pprint

rate_law = Eq(A.diff(t), -k * A)
print(rate_law)  # Output: Derivative(A, t) = -k*A
pprint(rate_law)
```

Listing 1.12: Creating a symbolic equation using Eq.

4. Derivative

Purpose: Compute the derivative of an expression.

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

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

5. dsolve

Purpose: Solve differential equations symbolically.

```python
from sympy import dsolve

solution = dsolve(Eq(A.diff(t), -k * A), A)
print(solution)  # Output: A(t) = C1*exp(-k*t)
pprint(solution)
```

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

6. solve

Purpose: Solve algebraic equations.

```python
from sympy import solve

result = solve(Eq(2 * A, B), A)
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).

```python
from sympy import symbols, integrate, pprint, exp

# Define symbols: t is the upper limit; tau is the integration
    variable.
tau, t, k, A0 = symbols('tau t k A0')

# Define the time-dependent concentration expression A(tau)
A_expr = A0 * exp(-k * tau)

# Compute the definite integral of k*A(tau) from tau=0 to tau=t
integral_definite = integrate(k * A_expr, (tau, 0, t))

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.

```python
from sympy import Matrix

M = Matrix([[1, 2], [3, 4]])
print(M)  # Output: Matrix([[1, 2], [3, 4]])
pprint(M)
```

Listing 1.17: Creating a 2x2 matrix using Matrix.

9. lambdify

Purpose: Convert symbolic expressions to numerical functions.

```python
from sympy import symbols, Function, Eq, dsolve, Derivative, exp,
    lambdify, pprint

# Define the symbols and the function
t, k, A0 = symbols('t k A0')
A = Function('A')

```

```python
# Define the differential equation: dA/dt = -k*A(t)
rate_eq = Eq(Derivative(A(t), t), -k * A(t))
pprint(rate_eq)

# Solve the ODE with the initial condition A(0) = A0
solution = dsolve(rate_eq, A(t), ics={A(0): A0})
pprint(solution)

# Extract the expression for A(t), which is A0*exp(-k*t)
A_expr = solution.rhs

# If you want to evaluate numerically, you need to substitute
    numerical values.
# For example, let A0 = 10 and k = 0.5.
A_expr_numeric = A_expr.subs({A0: 10, k: 0.5})

# Now lambdify the numerical expression with respect to t.
numerical_func = lambdify(t, A_expr_numeric, 'numpy')

# Evaluate A(t) at t = 5
result = numerical_func(5)
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.

```python
from sympy import symbols, Function

t, B = symbols('t␣B')
A = Function('A')

# Use A(t) instead of A
expression = A(t) + B

# To substitute, you must substitute the entire function call A(t)
substituted = expression.subs(A(t), 1)
print(substituted)  # Output: 1 + B
```

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

11. plot

Purpose: Visualize symbolic expressions.

```python
from sympy import symbols, Function, Eq, dsolve, Derivative, exp
from sympy.plotting import plot
from sympy import pprint
```

```python

# Define the symbols and the function
t, k, A0 = symbols('t k A0')
A = Function('A')

# Define the ODE: dA/dt = -k * A(t)
rate_eq = Eq(Derivative(A(t), t), -k * A(t))
pprint(rate_eq)

# Solve the ODE with an initial condition A(0) = A0
solution = dsolve(rate_eq, A(t), ics={A(0): A0})
pprint(solution)

# Extract the right-hand side expression: A(t) = A0*exp(-k*t)
A_expr = solution.rhs

# Substitute numerical values (for example: A0 = 1, k = 0.2)
A_expr_numeric = A_expr.subs({A0: 1, k: 0.2})

# Plot the numerical expression over t from 0 to 10
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. LaTeX

Purpose: Convert expressions into LaTeX format.

```python
from sympy import symbols, Function, latex

# Define the symbol for time
t = symbols('t')

# Define A as a function of t
A = Function('A')(t)

pprint(A.diff(t))

# Get the LaTeX representation of the derivative dA/dt
latex_expr = latex(A.diff(t))
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.

```python
from sympy import simplify, expand, factor, collect, pprint

```

```python
expr = (A + B)**2
pprint(simplify(expr))  # Simplify expression
pprint(expand(expr))    # Expand expression
pprint(factor(expr))    # Factorize expression
```

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

14. Piecewise

Purpose: Define piecewise functions.

```python
from sympy import Piecewise

piecewise_expr = Piecewise((A, A > 0), (0, True))
print(piecewise_expr)
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.

```python
from sympy import limit

limit_value = limit(A / t, t, 0)
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.

```python
from sympy import symbols, Function, Eq, dsolve, Derivative,
    series, pprint

# Define the symbols and the function
t, k, A0 = symbols('t k A0')
A = Function('A')

# Define the differential equation dA/dt = -k*A(t)
rate_eq = Eq(Derivative(A(t), t), -k * A(t))

# Solve the ODE with the initial condition A(0) = A0
solution = dsolve(rate_eq, A(t), ics={A(0): A0})
print("Solution of the ODE:")
pprint(solution)

# Extract the expression for A(t)
```

```python
A_expr = solution.rhs

# Compute the Taylor series expansion of A(t) about t = 0 up to
    order 3 (i.e., including terms up to t^2)
taylor_series = series(A_expr, t, 0, 3)
print("\nTaylor␣series␣expansion␣of␣A(t)␣up␣to␣order␣2:")
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:

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

   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:

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

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:

$$\frac{d[A]}{dt} = -k[A], \tag{2.2}$$

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

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

Second-Order Reaction

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

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

which, upon integration, gives:

$$\frac{1}{[A]} - \frac{1}{[A]_0} = kt. \tag{2.5}$$

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:

$$A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B. \tag{2.6}$$

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

$$r(t) = k_1 C_A(t) - k_2 C_B(t). \tag{2.7}$$

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

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

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

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

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:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C. \tag{2.10}$$

The corresponding rate equations are:

$$\frac{d[A]}{dt} = -k_1[A], \tag{2.11}$$

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

$$\frac{d[C]}{dt} = k_2[B]. \tag{2.13}$$

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:

$$A \xrightarrow{k_1} B, \quad A \xrightarrow{k_2} C. \tag{2.14}$$

The kinetic equations for parallel reactions are:

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

$$\frac{d[B]}{dt} = k_1[A], \tag{2.16}$$

$$\frac{d[C]}{dt} = k_2[A]. \tag{2.17}$$

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:

$$\mathrm{CH_3COCH_3} + \mathrm{I_2} \longrightarrow \mathrm{CH_3COCH_2I} + \mathrm{HI}. \tag{2.18}$$

This reaction proceeds in two stages:

1. **Enolization (Rate-Determining Step):**

$$\mathrm{CH_3COCH_3} + \mathrm{H_3O^+} \xrightarrow{k} \mathrm{CH_3COHCH_3} + \mathrm{H_2O}, \tag{2.19}$$

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:

$$\frac{dC_A}{dt} = k\, C_A\, C_{\mathrm{H_3O^+}}. \tag{2.20}$$

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

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

which gives rise to kinetic curves with characteristic bends due to the autocatalytic 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.

```python
from sympy import symbols, Eq, solve, integrate, pprint, latex, sqrt

# Define symbols for time (t), rate constant (k), and reaction order (n)
t, k, n = symbols('t k n')

# Define concentration variables: C_A (at time t) and initial concentration C_A0
C_A = symbols('C_A')
C_A0 = symbols('C_A0')

# Dictionary to store solutions for different reaction orders
solutions = {}

# Loop over different reaction orders: 0, 1, 2, and 3
for order in [0, 1, 2, 3]:
    # Separate variables:  C_A^(-order) dC_A = -k   dt
    lhs_integrated = integrate(C_A**(-order), C_A)
    rhs_integrated = integrate(-k, t)

    # Form the general integrated solution with an integration constant C1
    general_solution = Eq(lhs_integrated, rhs_integrated + symbols('C1'))

    # Use the initial condition C_A(0) = C_A0 to solve for C1
    C1_value = solve(general_solution.subs(t, 0).subs(C_A, C_A0), symbols('C1'))
    if C1_value:
        C1_value = C1_value[0]  # Extract the solution
        solution = solve(general_solution.subs(symbols('C1'), C1_value), C_A)
        if solution:
            solutions[order] = Eq(C_A, solution[0])  # Store the solution

```

```python
# Display the solutions for each reaction order
for order, solution in solutions.items():
    print(f"Solution␣for␣reaction␣order␣n␣=␣{order}:")
    pprint(solution)
    print("LaTeX␣format:")
    print(latex(solution))
    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`.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define numerical parameters
k_value = 1       # Rate constant
C_A0_value = 1    # Initial concentration
time = np.linspace(0, 1, 100)  # Time points

# Define analytical functions for each reaction order
def C_A_0order(t):
    return np.maximum(C_A0_value - k_value * t, 0)

def C_A_1order(t):
    return C_A0_value * np.exp(-k_value * t)

def C_A_2order(t):
    return C_A0_value / (1 + k_value * C_A0_value * t)

def C_A_3order(t):
    return C_A0_value / np.sqrt(1 + 2 * k_value * C_A0_value**2 *
        t)

# Plot the curves
plt.figure(figsize=(8, 6))
plt.plot(time, C_A_0order(time), label='n␣=␣0', linestyle='solid')
plt.plot(time, C_A_1order(time), label='n␣=␣1', linestyle='dotted'
    )
plt.plot(time, C_A_2order(time), label='n␣=␣2', linestyle='dashed'
    )
plt.plot(time, C_A_3order(time), label='n␣=␣3', linestyle='dashdot
    ')

plt.xlabel('Time␣(t)')
plt.ylabel('Concentration␣$C_A$')
plt.title('Kinetic␣Curves␣for␣Different␣Reaction␣Orders')
plt.legend()
plt.grid(True)
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

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

$$A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B, \tag{2.22}$$

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

$$r(t) = k_1 C_A(t) - k_2 C_B(t), \tag{2.23}$$

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

$$C_A(t) = C_{A0} - x(t), \tag{2.24}$$

$$C_B(t) = C_{B0} + x(t), \tag{2.25}$$

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:

$$\frac{dx(t)}{dt} = k_1 \Big[C_{A0} - x(t)\Big] - k_2 \Big[C_{B0} + x(t)\Big]. \tag{2.26}$$

Its solution is

$$x(t) = \frac{k_1 C_{A0} + k_2 C_{B0}}{k_1 + k_2} \Big[1 - e^{-(k_1+k_2)t}\Big], \tag{2.27}$$

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

$$A + B \underset{k_2}{\overset{k_1}{\rightleftharpoons}} C + D, \tag{2.28}$$

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

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

$$[A](t) = a_0 - x(t), \tag{2.29}$$
$$[B](t) = b_0 - x(t), \tag{2.30}$$
$$[C](t) = c_0 + x(t), \tag{2.31}$$
$$[D](t) = d_0 + x(t), \tag{2.32}$$

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

$$r_{\text{forward}} = k_1\big(a_0 - x(t)\big)\big(b_0 - x(t)\big), \tag{2.33}$$
$$r_{\text{reverse}} = k_2\big(c_0 + x(t)\big)\big(d_0 + x(t)\big). \tag{2.34}$$

Thus, the net rate of reaction is

$$\frac{dx}{dt} = k_1\big(a_0 - x\big)\big(b_0 - x\big) - k_2\big(c_0 + x\big)\big(d_0 + x\big). \tag{2.35}$$

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

$$\frac{dx}{dt} = -\gamma\, x^2 + \lambda\, x + \delta, \tag{2.36}$$

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. (2.36) 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,
$$\frac{du}{\gamma\left(\alpha^2 + u^2\right)} = dt.$$
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.

```python
from sympy import symbols, Function, Eq, dsolve, pprint, exp,
    solve
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Define symbols
t, k1, k2, A0, B0 = symbols('t k1 k2 A0 B0')
x = Function('x')(t)

# Define the differential equation
dx_dt = Eq(x.diff(t), k1 * (A0 - x) - k2 * (B0 + x))

# Solve the differential equation
sol_x = dsolve(dx_dt, x)
C1 = symbols('C1')

# Determine the integration constant using x(0)=0
C1_value = solve(sol_x.rhs.subs(t, 0) - 0, C1)[0]

# Substitute back to obtain x(t)
x_solution = sol_x.rhs.subs(C1, C1_value)
```

```python

# Express concentrations in terms of x(t)
C_A = A0 - x_solution
C_B = B0 + x_solution

print("\nExplicit␣Solution␣for␣x(t):")
pprint(x_solution)
print("\nSolution␣for␣C_A(t):")
pprint(C_A)
print("\nSolution␣for␣C_B(t):")
pprint(C_B)

# Convert symbolic expressions to numerical functions
from sympy import lambdify
x_func = lambdify((t, k1, k2, A0, B0), x_solution, 'numpy')

# Numerical ODE model
def reversible_reaction(y, t, k1, k2, A0, B0):
    x = y[0]
    dx_dt = k1 * (A0 - x) - k2 * (B0 + x)
    return [dx_dt]

# Define parameters and initial conditions
A0_val, B0_val = 1.0, 0.2
k1_val, k2_val = 0.45, 0.12
t_vals = np.linspace(0, 10, 100)
y0 = [0]  # x(0)=0

# Solve numerically using odeint
sol_x_numeric = odeint(reversible_reaction, y0, t_vals, args=(
    k1_val, k2_val, A0_val, B0_val))

# Compute concentrations from numerical x(t)
C_A_numeric = A0_val - sol_x_numeric[:, 0]
C_B_numeric = B0_val + sol_x_numeric[:, 0]

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(t_vals, C_A_numeric, label=r'$C_A(t)$', linestyle='solid'
    )
plt.plot(t_vals, C_B_numeric, label=r'$C_B(t)$', linestyle='dotted
    ')
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Reversible␣Reaction␣Kinetics')
plt.legend()
plt.grid(True)
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.

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Define symbols
t, gamma, lambda_, delta = sp.symbols('t gamma lambda delta', real
    =True, positive=True)
a0, b0, c0, d0, k1, k2 = sp.symbols('a0 b0 c0 d0 k1 k2', real=True
    , positive=True)

# Define x(t) as the extent of reaction
x = sp.Function('x')(t)

# Define the differential equation
ode = sp.Eq(sp.diff(x, t), -gamma * x**2 + lambda_ * x + delta)

# Solve the ODE with x(0)=0
sol = sp.dsolve(ode, x, ics={x.subs(t, 0): 0})
simplified_sol = sp.simplify(sol.rhs)

# Convert the solution to a numerical function
x_func = sp.lambdify((t, gamma, lambda_, delta), simplified_sol, '
    numpy')

# Define parameter values
A0_val, B0_val, C0_val, D0_val = 0.06, 0.05, 0.04, 0.03
k1_val, k2_val = 0.5, 0.25

# Calculate gamma, lambda, and delta
gamma_val = k1_val + k2_val
lambda_val = k1_val * (A0_val + B0_val) - k2_val * (C0_val +
    D0_val)
delta_val = k1_val * A0_val * B0_val - k2_val * C0_val * D0_val

# Define time array
t_vals = np.linspace(0, 60, 300)
x_vals = x_func(t_vals, gamma_val, lambda_val, delta_val)
```

```python

# Define numerical ODE model for the second-order reaction
def second_order_reversible(y, t, k1, k2, A0, B0, C0, D0):
    x = y[0]
    dx_dt = k1 * (A0 - x) * (B0 - x) - k2 * (C0 + x) * (D0 + x)
    return [dx_dt]

# Solve numerically with odeint
y0 = [0]  # x(0)=0
sol_x_numeric = odeint(second_order_reversible, y0, t_vals, args=(
    k1_val, k2_val, A0_val, B0_val, C0_val, D0_val))

# Compute concentrations
C_A_numeric = A0_val - sol_x_numeric[:, 0]
C_B_numeric = B0_val - sol_x_numeric[:, 0]
C_C_numeric = C0_val + sol_x_numeric[:, 0]
C_D_numeric = D0_val + sol_x_numeric[:, 0]

# Plot the concentration profiles
plt.figure(figsize=(8, 6))
plt.plot(t_vals, C_A_numeric, label=r'$A_0 - x(t)$', linestyle='
    solid', color='black')
plt.plot(t_vals, C_B_numeric, label=r'$B_0 - x(t)$', linestyle='
    dotted', color='black')
plt.plot(t_vals, C_C_numeric, label=r'$C_0 + x(t)$', linestyle='
    dashed', color='black')
plt.plot(t_vals, C_D_numeric, label=r'$D_0 + x(t)$', linestyle='
    dashdot', color='black')
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Second-Order Reversible Reaction Kinetics')
plt.legend()
plt.grid(True)
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.

2.5.2 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

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C. \tag{2.37}$$

At $t = 0$, the initial concentrations are

$$C_A(0) = C_{A0}, \quad C_B(0) = 0, \quad C_C(0) = 0. \tag{2.38}$$

The rate equations governing this system are

$$\frac{d[A]}{dt} = -k_1[A], \tag{2.39}$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B], \tag{2.40}$$

$$\frac{d[C]}{dt} = k_2[B]. \tag{2.41}$$

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

$$C_A(t) = C_{A0}\, e^{-k_1 t}. \tag{2.42}$$

Solving for the Intermediate $B(t)$

Substituting $C_A(t)$ into Eq. (2.40) gives

$$\frac{dC_B}{dt} + k_2 C_B = k_1 C_{A0}\, e^{-k_1 t}. \tag{2.43}$$

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

$$\mu(t) = e^{\int k_2 dt} = e^{k_2 t}. \tag{2.44}$$

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

$$e^{k_2 t}\frac{dC_B}{dt} + k_2 e^{k_2 t} C_B = k_1 C_{A0}\, e^{(k_2 - k_1)t}. \tag{2.45}$$

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

$$\frac{d}{dt}\left(C_B\, e^{k_2 t}\right) = k_1 C_{A0}\, e^{(k_2 - k_1)t}. \tag{2.46}$$

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

$$C_B(t)\, e^{k_2 t} = \int_0^t k_1 C_{A0}\, e^{(k_2-k_1)\tau}\, d\tau. \tag{2.47}$$

Evaluating the integral gives the solution

$$C_B(t) = C_{A0}\, \frac{k_1}{k_2 - k_1} \left(e^{-k_1 t} - e^{-k_2 t} \right), \tag{2.48}$$

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)

$$C_{A0} = C_A(t) + C_B(t) + C_C(t), \tag{2.49}$$

we solve for the concentration of the final product C:

$$C_C(t) = C_{A0} - C_A(t) - C_B(t). \tag{2.50}$$

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

$$C_C(t) = C_{A0} \left[1 - e^{-k_1 t} - \frac{k_1}{k_2 - k_1} \left(e^{-k_1 t} - e^{-k_2 t} \right) \right], \tag{2.51}$$

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 $C_{B,\mathbf{max}}$

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

$$\frac{dC_B}{dt} = 0. \tag{2.52}$$

Using the rate equation

$$\frac{dC_B}{dt} = k_1 C_{A0}\, e^{-k_1 t} - k_2 C_B(t), \tag{2.53}$$

and solving for t leads to

$$t_{\max} = \frac{\ln(k_2/k_1)}{k_2 - k_1}. \tag{2.54}$$

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

$$C_{B,\max} = C_{A0}\,\frac{k_1}{k_2 - k_1}\left[\left(\frac{k_2}{k_1}\right)^{-\frac{k_1}{k_2-k_1}} - \left(\frac{k_2}{k_1}\right)^{-\frac{k_2}{k_2-k_1}}\right]. \tag{2.55}$$

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:

$$A \xrightarrow{k_1} B, \quad B + C \xrightarrow{k_2} P. \tag{2.56}$$

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

$$\frac{dC_A}{dt} = -k_1 C_A, \tag{2.57}$$

$$\frac{dC_B}{dt} = k_1 C_A - k_2\, C_B\, C_C. \tag{2.58}$$

The solution for $C_A(t)$ remains

$$C_A(t) = C_{A0}\, e^{-k_1 t}. \tag{2.59}$$

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

$$\frac{dC_B}{dt} + k_2\, C_C\, C_B = k_1 C_{A0}\, e^{-k_1 t}. \tag{2.60}$$

Following the integrating factor method with

$$\mu(t) = e^{\int k_2 C_C\, dt} = e^{k_2 C_C\, t}, \tag{2.61}$$

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

$$\frac{d}{dt}\left(C_B\, e^{k_2 C_C\, t}\right) = k_1 C_{A0}\, e^{(k_2 C_C - k_1)t}. \tag{2.62}$$

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

$$C_B(t)\, e^{k_2 C_C\, t} = \int_0^t k_1 C_{A0}\, e^{(k_2 C_C - k_1)\tau}\, d\tau. \tag{2.63}$$

Evaluating the integral leads to

$$C_B(t) = C_{A0}\, \frac{k_1}{k_2 C_C - k_1} \left(e^{-k_1 t} - e^{-k_2 C_C\, t}\right), \tag{2.64}$$

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`.

```python
from sympy import symbols, Function, Eq, dsolve, pprint, exp,
    solve

# Define symbols and functions
t, k1, k2, C_A0 = symbols('t k1 k2 C_A0')
C_A = Function('C_A')(t)
C_B = Function('C_B')(t)
C_C = Function('C_C')(t)
```

```

# Differential equations for the basic mechanism
eq1 = Eq(C_A.diff(t), -k1 * C_A)
eq2 = Eq(C_B.diff(t), k1 * C_A - k2 * C_B)

# Solve for C_A(t)
sol_A = dsolve(eq1, C_A)
C_A_solution = sol_A.rhs.subs('C1', C_A0)

# Substitute C_A(t) into eq2 and solve for C_B(t)
eq2_subs = eq2.subs(C_A, C_A_solution)
sol_B = dsolve(eq2_subs, C_B)
C1_B = symbols('C1_B')
general_solution_B = sol_B.rhs.subs('C1', C1_B)
C1_B_value = solve(general_solution_B.subs(t, 0), C1_B)[0]
C_B_solution = general_solution_B.subs(C1_B, C1_B_value)

print("Solution␣for␣C_A(t):")
pprint(C_A_solution)
print("\nSolution␣for␣C_B(t):")
pprint(C_B_solution)
```

Listing 2.5: Symbolic Solution with SymPy

```
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt

# Define the model for the basic consecutive reaction A -> B -> C
def model(y, t, k1, k2):
    A, B, C = y
    dA_dt = -k1 * A
    dB_dt = k1 * A - k2 * B
    dC_dt = k2 * B
    return [dA_dt, dB_dt, dC_dt]

# Initial conditions and parameters
A0, B0, C0 = 1.0, 0.0, 0.0
k1, k2 = 1.0/2.0, 1.0/3.0
time = np.linspace(0, 10, 100)

# Compute t_max for the intermediate B
t_max = np.log(k2 / k1) / (k2 - k1)

# Solve the ODE system
y0 = [A0, B0, C0]
sol = odeint(model, y0, time, args=(k1, k2))

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(time, sol[:, 0], label='[A]', linestyle='solid')
plt.plot(time, sol[:, 1], label='[B]', linestyle='dotted')
plt.plot(time, sol[:, 2], label='[C]', linestyle='dashed')
plt.axvline(x=t_max, linestyle="--", color="red", label=r"$t_{\max
    }$")
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Consecutive Reaction Kinetics')
plt.legend()
plt.grid(True)
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:

$$A \xrightarrow{k_1} B, \quad B + C \xrightarrow{k_2} P. \tag{2.65}$$

In this simulation, we assume the following initial conditions:

- $A_0 = 1.0$ (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)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Define reaction rate constants
k1 = 0.5   # Rate constant for A -> B
k2 = 0.1   # Rate constant for B + C -> P

# Initial concentrations (Reordered for better logical flow)
A0 = 1.0   # Initial concentration of A
B0 = 0.0   # Initially, no B is present
C0 = 0.9   # Initial concentration of C
P0 = 0.0   # Initially, no P is present

# Time range for the simulation
t = np.linspace(0, 20, 200)  # Time from 0 to 20 units

# Define the system of differential equations
def reaction_rates(y, t, k1, k2):
    A, B, C, P = y  # Now the variables follow the same order as
        the initial conditions
    dA_dt = -k1 * A                  # A -> B
    dB_dt = k1 * A - k2 * B * C       # B is formed and consumed
        in B + C -> P
    dC_dt = -k2 * B * C               # C is consumed in the
        reaction
    dP_dt = k2 * B * C                # P is formed from B + C
```

```python
    return [dA_dt, dB_dt, dC_dt, dP_dt]

# Solve the system numerically
initial_conditions = [A0, B0, C0, P0]  # Reordered for consistency
solution = odeint(reaction_rates, initial_conditions, t, args=(k1,
     k2))

# Extract the concentrations from the solution array
A_conc = solution[:, 0]
B_conc = solution[:, 1]
C_conc = solution[:, 2]
P_conc = solution[:, 3]  # Now C(t) is properly changing over time

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(t, A_conc, label=r'$A(t)$', linestyle='solid')
plt.plot(t, B_conc, label=r'$B(t)$', linestyle='dashdot')
plt.plot(t, C_conc, label=r'$C(t)$', linestyle='dotted', color="
    green")
plt.plot(t, P_conc, label=r'$P(t)$', linestyle='dashed', color="
    red")

# Labels and title
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Kinetics␣of␣a␣Complex␣Successive␣Reaction')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

Listing 2.7: Numerical Simulation of a Complex Successive Reaction

2.5.3 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:

$$A \xrightarrow{k_1} B, \quad A \xrightarrow{k_2} C, \tag{2.66}$$

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:

$$\frac{dC_A}{dt} = -(k_1 + k_2)\, C_A, \tag{2.67}$$

$$\frac{dC_B}{dt} = k_1\, C_A, \tag{2.68}$$

$$\frac{dC_C}{dt} = k_2\, C_A. \tag{2.69}$$

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

$$C_A(t) = C_{A0}\, e^{-(k_1+k_2)t}, \tag{2.70}$$

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

$$C_B(t) = \int_0^t k_1\, C_A(\tau)\, d\tau = k_1 C_{A0} \int_0^t e^{-(k_1+k_2)\tau} d\tau. \tag{2.71}$$

Evaluating the integral gives

$$C_B(t) = \frac{k_1}{k_1 + k_2}\, C_{A0} \left(1 - e^{-(k_1+k_2)t}\right). \tag{2.72}$$

Similarly, for $C_C(t)$:

$$C_C(t) = \frac{k_2}{k_1 + k_2}\, C_{A0} \left(1 - e^{-(k_1+k_2)t}\right). \tag{2.73}$$

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

$$\frac{C_B(t)}{C_C(t)} = \frac{k_1}{k_2}. \tag{2.74}$$

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:

$$\mathrm{CH_3X + H_2O \xrightarrow{k_1} CH_3OH + H^+ + X^-}, \tag{2.75}$$

$$CH_3X + OH^- \xrightarrow{k_2} CH_3OH. \tag{2.76}$$

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:

$$\frac{dx(t)}{dt} = k_1(a - x(t)) + k_2(a - x(t))(b - x(t)). \tag{2.77}$$

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

By introducing the substitutions:

$$\alpha = k_1 + k_2(b - a), \tag{2.78}$$

$$\beta = \frac{k_2 a}{k_1 + k_2 b}, \tag{2.79}$$

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:

```python
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt

# Define rate equations for parallel reactions
def parallel_reactions(y, t, k1, k2):
    A, B, C = y
    dA_dt = -(k1 + k2) * A
    dB_dt = k1 * A
    dC_dt = k2 * A
    return [dA_dt, dB_dt, dC_dt]

# Initial conditions
A0, B0, C0 = 1.0, 0.0, 0.0  # Initially, only A is present
k1, k2 = 0.81, 0.27          # Rate constants for the competing
    pathways
time = np.linspace(0, 10, 100)  # Time range for the simulation

# Solve the system numerically
y0 = [A0, B0, C0]
solution = odeint(parallel_reactions, y0, time, args=(k1, k2))
```

```python

# Extract the concentration profiles
A_conc = solution[:, 0]
B_conc = solution[:, 1]
C_conc = solution[:, 2]

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(time, A_conc, label=r'$C_A(t)$', linestyle='solid')
plt.plot(time, B_conc, label=r'$C_B(t)$', linestyle='dotted')
plt.plot(time, C_conc, label=r'$C_C(t)$', linestyle='dashed')
plt.xlabel('Time␣(t)')
plt.ylabel('Concentration')
plt.title('Parallel␣Reaction␣Kinetics')
plt.legend()
plt.grid(True)
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 ($\mathrm{CH_3X}$) reacts via two competing pathways:

$$\mathrm{CH_3X + H_2O \xrightarrow{k_1} CH_3OH} \quad \text{(Pathway 1)}$$

$$\mathrm{CH_3X + OH^- \xrightarrow{k_2} CH_3OH} \quad \text{(Pathway 2)}$$

In this model, the variable $x(t)$ represents the extent of reaction (i.e., the fraction of $\mathrm{CH_3X}$ 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 $\mathrm{CH_3X}$ (remaining reactant) and the two products formed via each pathway. Finally, it plots the kinetic curves.

```python
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt

```

```python
# Define the system of differential equations for the parallel reaction.
# This function models a second-order parallel reaction:
#     CH3X + H2O  ->[k1] CH3OH   (Pathway 1)
#     CH3X + OH^-  ->[k2] CH3OH   (Pathway 2)
def parallel_reaction_kinetics(y, t, k1, k2, C_A0, C_B0):
    """
␣␣␣␣Parameters:
␣␣␣␣␣␣␣␣y␣␣␣␣:␣list␣␣->␣[x],␣where␣x␣is␣the␣extent␣of␣reaction␣at␣time␣t.
␣␣␣␣␣␣␣␣t␣␣␣␣:␣float␣->␣Time.
␣␣␣␣␣␣␣␣k1␣␣␣:␣float␣->␣Rate␣constant␣for␣the␣hydrolysis␣pathway␣(with␣H2O).
␣␣␣␣␣␣␣␣k2␣␣␣:␣float␣->␣Rate␣constant␣for␣the␣reaction␣with␣OH^-.
␣␣␣␣␣␣␣␣C_A0␣:␣float␣->␣Initial␣concentration␣of␣CH3X.
␣␣␣␣␣␣␣␣C_B0␣:␣float␣->␣Initial␣concentration␣of␣OH^-.

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣A␣list␣containing␣the␣derivative␣dx/dt.
␣␣␣␣"""
    x = y[0]  # x(t) is the reacted fraction of CH3X.

    # Rate equation for the extent of reaction, considering both pathways.
    dx_dt = k1 * (C_A0 - x) + k2 * (C_A0 - x) * (C_B0 - x)

    return [dx_dt]

# Initial Conditions
C_A0 = 1.0  # Initial concentration of CH3X
C_B0 = 0.5  # Initial concentration of OH^-
x0   = [0]  # Initially, no reaction has occurred

# Rate Constants
k1 = 0.3  # Rate constant for hydrolysis with water
k2 = 0.1  # Rate constant for reaction with OH^-

# Time Grid for the Simulation
time = np.linspace(0, 10, 100)  # Simulate from t = 0 to t = 10 seconds

# Solve the ODE System Numerically
solution = odeint(parallel_reaction_kinetics, x0, time, args=(k1, k2, C_A0, C_B0))

# Extract the Concentration Profiles
x_t   = solution[:, 0]           # Extent of reaction
C_A_t = C_A0 - x_t               # Remaining CH3X concentration
# The following approximations assign portions of the reacted CH3X
# to the two pathways based on the relative rate constants.
C_B_t = (k1 / (k1 + k2 * C_B0)) * x_t   # CH3OH from the water pathway
C_C_t = (k2 * C_A0 / (k1 + k2 * C_B0)) * x_t  # CH3OH from the OH^- pathway
```

```python

# Plot the Results
plt.figure(figsize=(8, 6))
plt.plot(time, C_A_t, label=r'$C_A(t)$ - CH3X', linestyle='solid',
     color='blue')
plt.plot(time, C_B_t, label=r'$C_B(t)$ - CH3OH (Water Pathway)',
    linestyle='dotted', color='green')
plt.plot(time, C_C_t, label=r'$C_C(t)$ - CH3OH (OH$^-$ Pathway)',
    linestyle='dashed', color='red')

# Graph Formatting
plt.xlabel('Time (t) [s]')
plt.ylabel('Concentration [M]')
plt.title('Second-Order Parallel Reaction Kinetics')
plt.legend()
plt.grid(True)
plt.show()
```

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

Interpretation of the Results

- The concentration of $\mathrm{CH_3X}$ ($C_A(t)$) 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 $\mathrm{OH^-}$.
- 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.

2.5.4 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:

$$\mathrm{CH_3COCH_3 + I_2 \longrightarrow CH_3COCH_2I + HI}. \tag{2.80}$$

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:

$$\mathrm{CH_3COCH_3 + H_3O^+ \overset{k}{\rightleftharpoons} CH_3COHCH_3 + H_2O}. \tag{2.81}$$

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:

$$\frac{dC_A}{dt} = kC_A C_{\mathrm{H_3O^+}}, \tag{2.82}$$

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:

$$\frac{dx(t)}{dt} = k\Big(C_{A0} - x(t)\Big)\Big(C_{B0} + x(t)\Big), \tag{2.83}$$

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

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

$$C_B(t) = C_{B0} + x(t) = \frac{C_{A0} + C_{B0}}{1 + \frac{C_{A0}}{C_{B0}} e^{-k(C_{A0}+C_{B0})t}}. \tag{2.84}$$

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:

$$t_{\max} = \frac{\ln(C_{A0}/C_{B0})}{k(C_{A0} + C_{B0})}. \tag{2.85}$$

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 $C_B(t)$:

```python
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp  # Import sympy for symbolic computations

# Define the ODE for the self-catalyzed reaction
def self_catalyzed(y, t, k, CA0, CB0):
    x = y[0]  # Reacted fraction of acetone
    dx_dt = k * (CA0 - x) * (CB0 + x)
    return [dx_dt]

# Initial conditions
CA0, CB0 = 0.8, 0.001  # Initial concentrations of acetone and
    hydronium ions
x0 = [0]               # Initial reacted fraction
k = 0.2                # Rate constant

# Define the time points for integration
time = np.linspace(0, 100, 200)

# Solve the differential equation
solution = odeint(self_catalyzed, x0, time, args=(k, CA0, CB0))

# Extract the data
x_t = solution[:, 0]
CB_t = CB0 + x_t  # Concentration of hydronium ions (product B)
CA_t = CA0 - x_t  # Concentration of acetone
reaction_rate = k * CA_t * CB_t  # Reaction rate

# Symbolic computation of t_max using sympy
t_sym, CA0_sym, CB0_sym, k_sym = sp.symbols('t CA0 CB0 k')
t_max_expr = sp.log(CA0_sym / CB0_sym) / (k_sym * (CA0_sym +
    CB0_sym))  # Expression for t_max

# Pretty print the symbolic expression for t_max
print("\nSymbolic expression for t_max:")
sp.pprint(t_max_expr)

# Evaluate t_max numerically
t_max_value = t_max_expr.subs({CA0_sym: CA0, CB0_sym: CB0, k_sym:
    k}).evalf()
print(f"\nNumerical value of t_max = {t_max_value:.4f}")

# Create the figure and subplots
fig, axes = plt.subplots(2, 1, figsize=(8, 10))

# First plot: Concentration profiles
axes[0].plot(time, CA_t, label=r'$C_A(t)$', linestyle='solid')
axes[0].plot(time, CB_t, label=r'$C_B(t)$', linestyle='dashed')
```

```python
axes[0].set_xlabel('Time (t)')
axes[0].set_ylabel('Concentration')
axes[0].set_title('Self-Catalyzed Reaction Kinetics')
axes[0].legend()
axes[0].grid(True)

# Second plot: Reaction rate vs. time
axes[1].plot(time, reaction_rate, label=r'$k C_A(t) C_B(t)$',
    linestyle='solid')
axes[1].axvline(x=float(t_max_value), color='red', linestyle='
    dashed', label=r'$t_{\max}$')
axes[1].set_xlabel('Time (t)')
axes[1].set_ylabel('Reaction Rate')
axes[1].set_title('Reaction Rate vs Time')
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
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 $[A] = [A]_0 e^{-kt}$.

```python
from sympy import symbols, Function, Eq, dsolve, pprint

t, k = symbols('t k')
```

```python
A = Function('A')(t)
deq = Eq(A.diff(t), -k * A)
sol = dsolve(deq, A)
pprint(sol)
```

Advanced Symbolic Derivations

In more complex reactions, symbolic computation allows us to derive exact solutions for different kinetic models, such as **second-order reactions** and **consecutive reactions**.

```python
from sympy import symbols, Function, Eq, dsolve, exp, simplify,
    pprint

# Define symbols
t, k1, k2, CA0, CB0, CC0 = symbols('t␣k1␣k2␣CA0␣CB0␣CC0')
CA = Function('CA')(t)
CB = Function('CB')(t)
CC = Function('CC')(t)

# First-Order Reaction: dCA/dt = -k1 * CA
eq1 = Eq(CA.diff(t), -k1 * CA)
sol_CA = dsolve(eq1, CA, ics={CA.subs(t, 0): CA0})

# Second-Order Reaction: dCA/dt = -k1 * CA * CB
eq2 = Eq(CA.diff(t), -k1 * CA * CB)
sol_CA_2nd = dsolve(eq2, CA)  # Solution without initial condition

# Consecutive Reaction: A -> B -> C
eq_consec_B = Eq(CB.diff(t), k1 * CA - k2 * CB)
sol_CB = dsolve(eq_consec_B.subs(CA, sol_CA.rhs), CB, ics={CB.subs
    (t, 0): 0})

eq_consec_C = Eq(CC.diff(t), k2 * CB)
sol_CC = dsolve(eq_consec_C.subs(CB, sol_CB.rhs), CC, ics={CC.subs
    (t, 0): 0})

# Display solutions
print("\nFirst-Order␣Decay␣Solution:")
pprint(sol_CA)

print("\nSecond-Order␣Reaction␣Solution:")
pprint(sol_CA_2nd)

print("\nConsecutive␣Reaction␣Solutions:")
pprint(sol_CB)
pprint(sol_CC)
```

2.6.2 Activity 2: Numerical Integration of a Multi-Step Kinetic Models

Many chemical reactions occur through multiple steps, either in a **consecutive** (sequential) manner or as **parallel** (competing) reactions. The goal of this activity is to:

- Model **consecutive** and **parallel** reaction mechanisms using differential equations.
- Solve these equations numerically using Python.
- Visualize concentration changes over time.

```python
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt

# Define rate equations for consecutive reactions
def consecutive_reaction(y, t, k1, k2):
    A, B, C = y
    dA_dt = -k1 * A
    dB_dt = k1 * A - k2 * B
    dC_dt = k2 * B
    return [dA_dt, dB_dt, dC_dt]

# Define rate equations for parallel reactions
def parallel_reaction(y, t, k1, k2):
    A, B, C = y
    dA_dt = -(k1 + k2) * A
    dB_dt = k1 * A
    dC_dt = k2 * A
    return [dA_dt, dB_dt, dC_dt]

# Initial conditions
A0, B0, C0 = 1.0, 0.0, 0.0  # Only A is initially present
k1, k2 = 0.5, 0.3
time = np.linspace(0, 10, 100)

# Solve consecutive reaction
sol_consec = odeint(consecutive_reaction, [A0, B0, C0], time, args
    =(k1, k2))

# Solve parallel reaction
sol_parallel = odeint(parallel_reaction, [A0, B0, C0], time, args
    =(k1, k2))

# Plot results
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(time, sol_consec[:, 0], label='[A]', linestyle='solid')
plt.plot(time, sol_consec[:, 1], label='[B]', linestyle='dotted')
plt.plot(time, sol_consec[:, 2], label='[C]', linestyle='dashed')
plt.title('Consecutive␣Reaction')
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(time, sol_parallel[:, 0], label='[A]', linestyle='solid')
plt.plot(time, sol_parallel[:, 1], label='[B]', linestyle='dotted'
    )
plt.plot(time, sol_parallel[:, 2], label='[C]', linestyle='dashed'
    )
plt.title('Parallel␣Reaction')
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

2.6.3 Activity 3: Second-Order Reaction with Two Reactants

Overview: Consider a second-order reaction:

$$A + B \xrightarrow{k} \text{Products}, \tag{2.86}$$

with the rate equation:

$$\frac{dx}{dt} = k(A_0 - x)(B_0 - x), \tag{2.87}$$

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

```python
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, Eq, integrate, solve, pprint, lambdify

# Define symbols and parameters
t, k = symbols('t␣k', real=True, positive=True)
A0, B0 = symbols('A0␣B0', real=True, positive=True)
x = symbols('x', real=True, positive=True)
C1 = symbols('C1')

# Integrate the separated variables: 1/[(A0-x)(B0-x)] dx = k dt
lhs_integrated = integrate(1/((A0 - x)*(B0 - x)), x)
rhs_integrated = integrate(k, t)
integrated_eq = Eq(lhs_integrated, rhs_integrated + C1)

# Determine the integration constant using x(0) = 0
C1_value = solve(integrated_eq.subs(t, 0).subs(x, 0), C1)[0]
final_equation = integrated_eq.subs(C1, C1_value)

# Solve explicitly for x(t)
```

```
x_solution = solve(final_equation, x)[0]
print("Final␣Integrated␣Rate␣Law:")
pprint(Eq(x, x_solution))
```

Step 2: Plotting the Kinetic Curves

```
# Convert the symbolic solution to a numerical function
x_numeric = lambdify((t, A0, B0, k), x_solution, "numpy")

# Set numerical parameters
k_value = 0.02
A0_value = 0.08
B0_value = 0.06
time = np.linspace(0, 10000, 200)

# Compute the extent of reaction and reactant concentrations
x_values = x_numeric(time, A0_value, B0_value, k_value)
A_values = A0_value - x_values
B_values = B0_value - x_values

# Ensure concentrations remain non-negative
A_values[A_values < 0] = 0
B_values[B_values < 0] = 0

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(time, A_values, label='[A]␣(Reactant␣A)')
plt.plot(time, B_values, label='[B]␣(Reactant␣B)')
plt.xlabel('Time␣(s)')
plt.ylabel('Concentration␣(mol/L)')
plt.title('Second-Order␣Kinetics␣with␣Two␣Reactants')
plt.legend()
plt.grid(True)
plt.show()
```

2.6.4 Activity 4: Modeling Self-Catalyzed Reaction Kinetics

Introduction to Self-Catalysis

A **self-catalyzed reaction** is a chemical process in which one of the products acts as a catalyst, accelerating the reaction as it proceeds. This type of reaction is common in organic chemistry, particularly in acid- or base-catalyzed reactions where the product influences the reaction rate.

The following Python script solves the self-catalyzed reaction using numerical integration and visualizes the concentration profiles and reaction rate.

```
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols, solve, ln, pprint

```

```python
# Define ODE model
def self_catalyzed(y, t, k, CA0, CB0):
    x = y[0]  # Reacted fraction
    dx_dt = k * (CA0 - x) * (CB0 + x)
    return [dx_dt]

# Initial conditions
CA0, CB0 = 0.8, 0.001  # Initial concentrations
x0 = [0]  # Initial reacted fraction
k = 0.2  # Rate constant
time = np.linspace(0, 100, 100)

# Solve ODE numerically
solution = odeint(self_catalyzed, x0, time, args=(k, CA0, CB0))
x_t = solution[:, 0]
CA_t = CA0 - x_t
CB_t = CB0 + x_t

# Calculate t_max using SymPy
t_sym, k_sym, CA0_sym, CB0_sym = symbols('t k CA0 CB0')
t_max_expr = ln(CA0_sym / CB0_sym) / (k_sym * (CA0_sym + CB0_sym))
t_max_value = t_max_expr.subs({CA0_sym: CA0, CB0_sym: CB0, k_sym:
    k}).evalf()

# Print symbolic expression
print("\nExpression for t_max:")
pprint(t_max_expr)

print("\nComputed value for t_max:", t_max_value)

# Plot concentration curves
plt.figure(figsize=(10, 5))

# Plot CA(t) and CB(t)
plt.subplot(1, 2, 1)
plt.plot(time, CA_t, label=r'$C_A(t)$', linestyle='solid')
plt.plot(time, CB_t, label=r'$C_B(t)$', linestyle='dashed')
plt.xlabel('Time (t)')
plt.ylabel('Concentration')
plt.title('Self-Catalyzed Reaction')
plt.legend()
plt.grid(True)

# Plot rate k * CA * CB
rate_t = k * CA_t * CB_t
plt.subplot(1, 2, 2)
plt.plot(time, rate_t, label=r'$k C_A C_B$', linestyle='solid')
plt.axvline(x=t_max_value, linestyle="--", color="red", label=r"
    $t_{\max}$")
plt.xlabel('Time (t)')
plt.ylabel('Reaction Rate')
plt.title('Reaction Rate vs Time')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

Assignments

Task: Implement a Python script that models the consecutive reaction $A \to B \to C$. 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 $A \to B \to C$.
- 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:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C \tag{3.1}$$

The rate equations are:

$$\frac{d[A]}{dt} = -k_1[A] \tag{3.2}$$
$$\frac{d[B]}{dt} = k_1[A] - k_2[B] \tag{3.3}$$
$$\frac{d[C]}{dt} = k_2[B] \tag{3.4}$$

Representing this system in matrix form:

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

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

$$\alpha = \begin{bmatrix}-1 & 1 & 0\\ 0 & -1 & 1\end{bmatrix} \tag{3.6}$$

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

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

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

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

Explicitly calculating:

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

Expanding the matrix multiplication:

$$\frac{d[A]}{dt} = -k_1[A] \tag{3.10}$$
$$\frac{d[B]}{dt} = k_1[A] - k_2[B] \tag{3.11}$$
$$\frac{d[C]}{dt} = k_2[B] \tag{3.12}$$

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:

$$2A \rightleftharpoons 2B + C, \quad A + B \rightleftharpoons 2D, \quad B + D \rightleftharpoons E \tag{3.13}$$

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

$$\begin{aligned} k_1, k_2 : &\quad 2A \rightleftharpoons 2B + C \\ k_3, k_4 : &\quad A + B \rightleftharpoons 2D \\ k_5, k_6 : &\quad B + D \rightleftharpoons E \end{aligned}$$

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

$$\alpha = \begin{bmatrix} -2 & 2 & 1 & 0 & 0 \\ 2 & -2 & -1 & 0 & 0 \\ -1 & -1 & 0 & 2 & 0 \\ 1 & 1 & 0 & -2 & 0 \\ 0 & -1 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 & -1 \end{bmatrix} \tag{3.14}$$

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

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

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

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

```python
import sympy as sp

# Define symbols
A, B, C, D, E = sp.symbols('A B C D E')
k1, k2, k3, k4, k5, k6 = sp.symbols('k1 k2 k3 k4 k5 k6')

# Define rate equations
r = sp.Matrix([
    k1*A**2,
    k2*B**2*C,
    k3*A*B,
    k4*D**2,
    k5*B*D,
    k6*E
])

# Stoichiometric matrix
alpha = sp.Matrix([[-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]])

# Compute concentration derivatives
dC_dt = alpha.T * r
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:

$$\frac{d}{dt}\mathbb{C}(t) = \mathbb{K}\,\mathbb{C}(t), \tag{3.17}$$

where:

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

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

$$\mathbb{C}(t) = e^{\mathbb{K}t}\,\mathbb{C}_0, \tag{3.18}$$

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

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

Consider the following kinetic scheme:

$$B \xleftarrow{k_1} A \xrightarrow{k_2} C \underset{k_4}{\overset{k_3}{\rightleftharpoons}} D. \tag{3.19}$$

This reaction leads to the system of coupled differential equations:

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

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

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

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:

$$\mathbb{K} = \mathbb{X}\,\Lambda\,\mathbb{X}^{-1}, \tag{3.22}$$

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:

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

where

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

Thus, the concentration profile at time t is:

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

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:

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

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:

$$\mathbb{K}\,\mathbb{X} = \mathbb{X}\,\Lambda. \tag{3.27}$$

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:

```python
import sympy as sp

# Define symbols
t = sp.Symbol('t')
C0 = sp.Matrix([sp.Symbol('C0_1'), 0, 0, 0])
k1, k2, k3, k4 = sp.symbols('k1␣k2␣k3␣k4')

# Define the rate constant matrix
K = sp.Matrix([[-(k1 + k2), 0, 0, 0],
               [k1, 0, 0, 0],
                [k2, 0, -k3, k4],
                [0, 0, k3, -k4]])

# Compute eigenvalues and eigenvectors
eigen_data = K.eigenvects()
eigenvalues = [ev[0] for ev in eigen_data]
eigenvectors = []
for ev in eigen_data:
    for vec in ev[2]:  # Include all eigenvectors for degenerate
        cases
        eigenvectors.append(vec)

# Ensure the eigenvector matrix is square
if len(eigenvectors) == K.shape[0]:
    X = sp.Matrix.hstack(*eigenvectors)  # Construct eigenvector
        matrix
    e_Lambda_t = sp.diag(*[sp.exp(ev[0] * t) for ev in eigen_data
        for _ in ev[2]])

    # Compute matrix exponential using similarity transformation
    exp_Kt = X * e_Lambda_t * X.inv()

    # Compute concentration evolution
    C_t = exp_Kt * C0

    # Display final concentration evolution
    print("\nConcentration␣Evolution:")
    sp.pprint(C_t)
else:
    print("Error:␣Eigenvector␣matrix␣is␣not␣square.␣Direct␣
        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:

$$\mathrm{A} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{B} \xrightarrow{k_3} \mathrm{C}, \tag{3.28}$$

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:

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

Solving for C_B yields:

$$C_B = \frac{k_1}{k_2 + k_3} C_A. \tag{3.30}$$

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

$$r_P = k_3 C_B = \frac{k_1 k_3}{k_2 + k_3} C_A. \tag{3.31}$$

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:

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

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Define rate constants and initial concentration
k1, k2, k3 = 0.15, 0.07, 10
A0 = 0.1

# Define the system of ODEs
def reaction_system(t, y):
    A, B = y
    dA_dt = -k1 * A
    dB_dt = k1 * A - (k2 + k3) * B
    return [dA_dt, dB_dt]

# Solve the ODEs over the interval from 0 to 2 seconds
t_span = [0, 2]
t_eval = np.linspace(0, 2, 100)
sol = solve_ivp(reaction_system, t_span, [A0, 0], t_eval=t_eval)

# Extract the concentration profiles for A and B
A, B = sol.y
t = sol.t

# Compute the steady-state prediction for the rate of product
    formation
rP_steady = (k1 * k3 / (k2 + k3)) * A

# Plot the numerical and steady-state predicted rates
plt.figure(figsize=(8, 5))
plt.plot(t, k3 * B, label=r'$k_3␣B(t)$', color='black')
plt.plot(t, rP_steady, '--', label=r'$\frac{k_1␣k_3}{k_2␣+␣k_3}␣A(
    t)$', color='black')
plt.xlabel('Time␣(s)')
plt.ylabel('Rate␣(mol/L/s)')
plt.legend()
plt.grid()
plt.show()
```

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

The simulation shows that the numerical solution for $k_3 B(t)$ (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:

$$A + A \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} A^* + A. \tag{3.33}$$

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

   - *Deexcitation via collision:*

$$A^* + A \xrightarrow{k_2} A + A, \tag{3.34}$$

   - *Conversion to Product:*

$$A^* \xrightarrow{k_3} P. \tag{3.35}$$

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

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

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

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

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.

```python
import sympy as sp
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# ---------------------------------------------------------------
# 1) SYMBOLIC DERIVATION: Solve for [A*] under the steady-state
    condition.
#    The steady-state condition is: d[A*]/dt = 0, i.e.,
#       k1*[A]^2 = (k2*[A] + k3)*[A*]
# ---------------------------------------------------------------

# Define symbolic variables
A_sym, Astar_sym = sp.symbols('A Astar', positive=True)
k1_sym, k2_sym, k3_sym = sp.symbols('k1 k2 k3', positive=True)

# Write the steady-state equation for A*
eq_ss = sp.Eq(k1_sym*A_sym**2, Astar_sym * (k2_sym*A_sym + k3_sym)
    )
# Solve for [A*]
Astar_ss_expr = sp.solve(eq_ss, Astar_sym)[0]

print("Symbolic steady-state solution for [A*]:")
sp.pprint(Astar_ss_expr)

# ---------------------------------------------------------------
# 2) NUMERICAL INTEGRATION: Solve the full ODE system.
#
#    d[A]/dt    = -k1*A^2 + k2*A*A*
#    d[A*]/dt   =  k1*A^2 - k2*A*A* - k3*A*
#    d[P]/dt    =  k3*A*
# ---------------------------------------------------------------

# Define numerical values for the rate constants
k1_val = 1.0
k2_val = 10.0
k3_val = 1.0

# Initial conditions: [A]=1.0, [A*]=0.0, [P]=0.0
A0 = 1.0
Astar0 = 0.0
P0 = 0.0

def lindemann_odes(t, y):
    A, Astar, P = y
    dA_dt     = -k1_val*A**2 + k2_val*A*Astar
    dAstar_dt =  k1_val*A**2 - k2_val*A*Astar - k3_val*Astar
    dP_dt     =  k3_val*Astar
    return [dA_dt, dAstar_dt, dP_dt]

# Time span and points for evaluation
t_span = (0, 5)
t_eval = np.linspace(0, 5, 200)
sol = solve_ivp(lindemann_odes, t_span, [A0, Astar0, P0], t_eval=
    t_eval)
t_vals = sol.t
A_vals = sol.y[0]
Astar_vals = sol.y[1]
P_vals = sol.y[2]

# ---------------------------------------------------------------
# 3) EVALUATE THE STEADY-STATE EXPRESSION:
#    Convert the symbolic expression for [A*] into a function and
#    evaluate it using the numerical [A] from the ODE solution.
# ---------------------------------------------------------------

Astar_ss_func = sp.lambdify(
    (A_sym, k1_sym, k2_sym, k3_sym),
    Astar_ss_expr,
    'numpy'
)
Astar_ss_vals = Astar_ss_func(A_vals, k1_val, k2_val, k3_val)

# ---------------------------------------------------------------
# 4) PLOTTING THE RESULTS:
#    Compare the numerical solution for [A*] with the steady-state
#    approximation.
# ---------------------------------------------------------------
plt.figure(figsize=(8,6))
plt.plot(t_vals, A_vals,       'b-', label='[A](t)␣(numeric)')
plt.plot(t_vals, Astar_vals,   'r-', label='[A*](t)␣(numeric)')
plt.plot(t_vals, P_vals,       'g-', label='[P](t)␣(numeric)')
plt.plot(t_vals, Astar_ss_vals,'r--', label='[A*](t)␣steady-state'
    )
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Lindemann␣Mechanism:␣Comparison␣with␣Steady-State␣
    Approximation')
plt.legend()
plt.grid(True)
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:

$$\begin{aligned}
C_2H_6 &\xrightarrow{k_1} 2\ CH_3{}^{\bullet},\\
CH_3{}^{\bullet} + C_2H_6 &\xrightarrow{k_2} CH_4 + C_2H_5{}^{\bullet},\\
C_2H_5{}^{\bullet} &\xrightarrow{k_3} C_2H_4 + H^{\bullet},\\
C_2H_6 + H^{\bullet} &\xrightarrow{k_4} C_2H_5{}^{\bullet} + H_2,\\
2\ C_2H_5{}^{\bullet} &\xrightarrow{k_5} C_4H_{10},\\
2\ C_2H_5{}^{\bullet} &\xrightarrow{k_6} C_2H_4 + C_2H_6.
\end{aligned}$$

In this mechanism, the radicals $CH_3{}^{\bullet}$, $C_2H_5{}^{\bullet}$, 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:

$$\text{Production: } 2k_1[C_2H_6], \qquad \text{Consumption: } k_2[CH_3{}^{\bullet}][C_2H_6].$$

At steady state, the net rate is zero:

$$2k_1[C_2H_6] - k_2[CH_3{}^{\bullet}][C_2H_6] = 0.$$

Dividing both sides by $[C_2H_6]$ (assuming it is nonzero) immediately yields

$$[CH_3{}^{\bullet}] = \frac{2k_1}{k_2}. \tag{3.38}$$

2. The $C_2H_5{}^{\bullet}$ Radical:

It is convenient to introduce the following shorthand notation:

$$C_1 = [C_2H_6], \quad C_4 = [C_2H_5{}^{\bullet}], \quad C_5 = [H^{\bullet}].$$

The $C_2H_5{}^{\bullet}$ radical is generated by:

- Reaction (2): at a rate
$$r_2 = k_2\,[\mathrm{CH_3}^\bullet]\,C_1.$$
Substituting the expression from (3.38) gives
$$r_2 = k_2\left(\frac{2k_1}{k_2}\right)C_1 = 2k_1C_1.$$
- 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 $\mathrm{C_2H_5}^\bullet$ at a rate
$$2(k_5+k_6)C_4^2.$$

Thus, the steady-state balance for $\mathrm{C_2H_5}^\bullet$ is
$$2k_1C_1 + k_4C_1C_5 - k_3C_4 - 2(k_5+k_6)C_4^2 = 0.$$

3. The $\mathrm{H}^\bullet$ Radical:

The hydrogen radical is produced in Reaction (3) and consumed in Reaction (4):
$$k_3C_4 - k_4C_1C_5 = 0.$$
Solving for C_5 gives
$$C_5 = \frac{k_3C_4}{k_4C_1}. \tag{3.39}$$

Eliminating C_5 in the $\mathrm{C_2H_5}^\bullet$ balance:

Substitute the expression for C_5 from (3.39) into the steady-state equation for $\mathrm{C_2H_5}^\bullet$:
$$2k_1C_1 + k_4C_1\left(\frac{k_3C_4}{k_4C_1}\right) - k_3C_4 - 2(k_5+k_6)C_4^2 = 0.$$

Notice that the term $k_4C_1(k_3C_4/(k_4C_1))$ simplifies to k_3C_4 and cancels with the $-k_3C_4$. This leaves
$$2k_1C_1 = 2(k_5+k_6)C_4^2,$$
so that
$$[\mathrm{C_2H_5}^\bullet] = C_4 = \sqrt{\frac{k_1C_1}{k_5+k_6}} = \sqrt{\frac{k_1[\mathrm{C_2H_6}]}{k_5+k_6}}. \tag{3.40}$$

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

$$[H^\bullet] = C_5 = \frac{k_3}{k_4}\sqrt{\frac{k_1}{(k_5+k_6)C_1}} = \frac{k_3}{k_4}\sqrt{\frac{k_1}{(k_5+k_6)[C_2H_6]}}. \tag{3.41}$$

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

$$[CH_3{}^\bullet] = \frac{2k_1}{k_2}, \quad [C_2H_5{}^\bullet] = \sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}}, \quad [H^\bullet] = \frac{k_3}{k_4}\sqrt{\frac{k_1}{(k_5+k_6)[C_2H_6]}}.$$

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): $r_1 = k_1[C_2H_6]$,
- Reaction (2): $r_2 = 2k_1[C_2H_6]$ (using the result for $[CH_3{}^\bullet]$ from (3.38)),
- Reaction (4): $r_4 = k_4[C_2H_6][H^\bullet] = k_3\sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}}$ (after substituting (3.41)).

It is regenerated in Reaction (6) at a rate

$$r_6 = k_6[C_2H_5{}^\bullet]^2 = \frac{k_1k_6}{k_5+k_6}[C_2H_6].$$

Thus, the net rate of consumption of ethane is

$$\begin{aligned}\frac{d[C_2H_6]}{dt} &= -r_1 - r_2 - r_4 + r_6 \\ &= -k_1[C_2H_6] - 2k_1[C_2H_6] - k_3\sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}} + \frac{k_1k_6}{k_5+k_6}[C_2H_6] \\ &= -\left(3k_1 - \frac{k_1k_6}{k_5+k_6}\right)[C_2H_6] - k_3\sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}}. \end{aligned} \tag{3.42}$$

Methane (CH_4)**:**

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

$$\frac{d[CH_4]}{dt} = 2k_1[C_2H_6]. \tag{3.43}$$

Ethylene (C_2H_4)**:**

Ethylene is formed via two pathways:

- Reaction (3): $r_3 = k_3\sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}}$,
- Reaction (6): $r_6 = \frac{k_1k_6}{k_5+k_6}[C_2H_6]$.

Thus,

$$\frac{d[C_2H_4]}{dt} = k_3\sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}} + \frac{k_1k_6}{k_5+k_6}[C_2H_6]. \tag{3.44}$$

Hydrogen (H_2):

Hydrogen is produced in Reaction (4):

$$\frac{d[H_2]}{dt} = k_3\sqrt{\frac{k_1[C_2H_6]}{k_5+k_6}}. \tag{3.45}$$

Butane (C_4H_{10}):

Butane is formed by the dimerization of two $C_2H_5^\bullet$ radicals in Reaction (5):

$$\frac{d[C_4H_{10}]}{dt} = \frac{k_1k_5}{k_5+k_6}[C_2H_6]. \tag{3.46}$$

For convenience, we define the following constants:

$$\alpha = 3k_1 - \frac{k_1k_6}{k_5+k_6}, \qquad \beta = k_3\sqrt{\frac{k_1}{k_5+k_6}}, \tag{3.47}$$

so that Equation (3.42) may be compactly written as

$$\frac{d[C_2H_6]}{dt} = -\alpha[C_2H_6] - \beta\sqrt{[C_2H_6]}. \tag{3.48}$$

Once the time evolution of $[C_2H_6]$ is determined (e.g., by numerical integration of (3.48)), the product concentrations can be obtained from the integrated forms:

$$[CH_4](t) = 2k_1\int_0^t [C_2H_6](u)\,du, \tag{3.49}$$

$$[C_2H_4](t) = \beta\int_0^t \sqrt{[C_2H_6](u)}\,du + \frac{k_1k_6}{k_5+k_6}\int_0^t [C_2H_6](u)\,du, \tag{3.50}$$

$$[H_2](t) = \beta\int_0^t \sqrt{[C_2H_6](u)}\,du, \tag{3.51}$$

$$[C_4H_{10}](t) = \frac{k_1k_5}{k_5+k_6}\int_0^t [C_2H_6](u)\,du. \tag{3.52}$$

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:

```python
import sympy as sp

# 1) Define symbols for rate constants and concentrations.
#    Notation:
#      C1 : [C2H6]
#      C2 : [CH3*]
#      C3 : [CH4]
#      C4 : [C2H5*]
#      C5 : [H*]
#      C6 : [C2H4]
#      C7 : [H2]
#      C8 : [C4H10]
k1, k2, k3, k4, k5, k6 = sp.symbols('k1␣k2␣k3␣k4␣k5␣k6', positive=True)
C1, C2, C3, C4, C5, C6, C7, C8 = sp.symbols('C1␣C2␣C3␣C4␣C5␣C6␣C7␣C8', positive=True)

# 2) Define the reaction rates.
# Reaction scheme:
#  R1: C2H6 -> 2 CH3*
#  R2: CH3* + C2H6 -> CH4 + C2H5*
#  R3: C2H5* -> C2H4 + H*
#  R4: C2H6 + H* -> C2H5* + H2
#  R5: 2 C2H5* -> C4H10
#  R6: 2 C2H5* -> C2H4 + C2H6
r1 = k1 * C1                          # Reaction 1
r2 = k2 * C2 * C1                     # Reaction 2
r3 = k3 * C4                          # Reaction 3
r4 = k4 * C1 * C5                     # Reaction 4
r5 = k5 * C4**2                       # Reaction 5
r6 = k6 * C4**2                       # Reaction 6
r_vec = sp.Matrix([r1, r2, r3, r4, r5, r6])

# 3) Define the stoichiometric matrix.
# Rows correspond to the species:
#   0: C2H6, 1: CH3*, 2: CH4, 3: C2H5*, 4: H*, 5: C2H4, 6: H2, 7: C4H10
alpha = sp.Matrix([
    [-1, -1,  0, -1,  0, +1],   # C2H6
    [ 2, -1,  0,  0,  0,  0],   # CH3*
    [ 0, +1,  0,  0,  0,  0],   # CH4
    [ 0, +1, -1, +1, -2, -2],   # C2H5*
    [ 0,  0, +1, -1,  0,  0],   # H*
    [ 0,  0, +1,  0,  0, +1],   # C2H4
    [ 0,  0,  0, +1,  0,  0],   # H2
    [ 0,  0,  0,  0, +1,  0]    # C4H10
])

```

```python
# 4) Compute the net production rates for each species.
net_rates = alpha * r_vec
# The entries of net_rates correspond to:
#   d[C2H6]/dt, d[CH3*]/dt, d[CH4]/dt, d[C2H5*]/dt, d[H*]/dt, d[C2H4]/dt, d[H2]/dt, d[C4H10]/dt

# 5) Apply the steady-state approximation for the radicals:
#    Set net rates for CH3* (row 1), C2H5* (row 3), and H* (row 4) equal to zero.
eq_CH3  = sp.Eq(net_rates[1], 0)  # CH3*
eq_C2H5 = sp.Eq(net_rates[3], 0)  # C2H5*
eq_H    = sp.Eq(net_rates[4], 0)  # H*
sol_radicals = sp.solve((eq_CH3, eq_C2H5, eq_H), (C2, C4, C5), dict=True)

print("Steady-state␣solutions␣for␣the␣radical␣species:")
sp.pprint(sol_radicals)
print("\n")
# Expected solutions:
#   [CH3*] = 2*k1/k2,
#   [C2H5*] = sqrt(k1 * C1/(k5 + k6)),
#   [H*]    = (k3/k4) * sqrt(k1/(C1*(k5+k6))).

# 6) Substitute the steady-state radical expressions into the net production rates
net_rates_ss = sp.simplify(net_rates.subs(sol_radicals[0]))

# 7) Display the differential equations for all species.
species = {
    0: r"[C2H6]",
    1: r"[CH3*]",
    2: r"[CH4]",
    3: r"[C2H5*]",
    4: r"[H*]",
    5: r"[C2H4]",
    6: r"[H2]",
    7: r"[C4H10]"
}

print("Differential␣equations␣after␣applying␣the␣steady-state␣approximation:\n")
for i in range(net_rates_ss.shape[0]):
    print(f"d{species[i]}/dt␣=")
    sp.pprint(net_rates_ss[i])
    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:

$$[\mathrm{CH_3^\bullet}] = \frac{2k_1}{k_2}, \quad [\mathrm{C_2H_5^\bullet}] = \sqrt{\frac{k_1\,C1}{k_5+k_6}}, \quad [\mathrm{H^\bullet}] = \frac{k_3}{k_4}\sqrt{\frac{k_1}{(k_5+k_6)\,C1}},$$

confirming our analytical derivations.

Solving for the Ethane Concentration $[\mathrm{C_2H_6}](t)$ Recall that the differential equation for ethane is given by

$$\frac{d[\mathrm{C_2H_6}]}{dt} = -\alpha[\mathrm{C_2H_6}] - \beta\sqrt{[\mathrm{C_2H_6}]},$$

where

$$\alpha = 3k_1 - \frac{k_1k_6}{k_5+k_6}, \qquad \beta = k_3\sqrt{\frac{k_1}{k_5+k_6}}.$$

To obtain an analytical expression for $[\mathrm{C_2H_6}](t)$, we perform the substitution

$$u(t) = \sqrt{[\mathrm{C_2H_6}](t)} \quad \Longrightarrow \quad [\mathrm{C_2H_6}](t) = u(t)^2.$$

Differentiating yields

$$\frac{d[\mathrm{C_2H_6}]}{dt} = 2u(t)\frac{du}{dt}.$$

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:

```python
import sympy as sp

# Define symbols and the function for ethane concentration C(t)
t, C0, alpha, beta = sp.symbols('t␣C0␣alpha␣beta', positive=True)
C = sp.Function('C')(t)

# Original ODE: dC/dt = -alpha * C - beta * sqrt(C)
ode = sp.Eq(sp.diff(C, t), -alpha * C - beta * sp.sqrt(C))

# Substitute: let u(t) = sqrt(C(t)), so that C(t) = u(t)**2.
# Then, dC/dt = 2*u(t)*du/dt.
# Substituting into the ODE:
#   2*u*du/dt = -alpha*u**2 - beta*u.
# Dividing by u (assuming u>0) yields:
```

```python
#   du/dt = - (alpha/2)*u - beta/2.
u = sp.Function('u')(t)
ode_u = sp.Eq(sp.diff(u, t), -alpha/2 * u - beta/2)

# Solve the ODE for u(t) with the initial condition: u(0) = sqrt(
    C0)
solution_u = sp.dsolve(ode_u, u, ics={u.subs(t, 0): sp.sqrt(C0)})

# The solution for C(t) is then [u(t)]**2.
solution_C = sp.simplify(solution_u.rhs**2)

print("Ethane␣concentration␣[C2H6](t)␣in␣terms␣of␣alpha␣and␣beta:"
    )
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:

$$k_1 = 4.26 \times 10^{16} \exp\left(-\frac{44579}{T}\right),$$

$$k_2 = 1.65 \times 10^{9} \left(\frac{T}{298}\right)^{4.25} \exp\left(-\frac{3890}{T}\right),$$

$$k_3 = 8.85 \times 10^{12} \exp\left(-\frac{19469}{T}\right),$$

$$k_4 = 1.71 \times 10^{12} \left(\frac{T}{298}\right)^{2.32} \exp\left(-\frac{3414}{T}\right),$$

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 $R = 8.3154\,\mathrm{J/(mol\,K)}$. (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.

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Define rate constants as functions of temperature T
def rate_constants(T):
    k1 = 4.26e16 * np.exp(-44579 / T)
    k2 = 1.65e9 * (T / 298) ** 4.25 * np.exp(-3890 / T)
    k3 = 8.85e12 * np.exp(-19469 / T)
    k4 = 1.71e12 * (T / 298) ** 2.32 * np.exp(-3414 / T)
    k5 = 1.15e13  # constant
    k6 = 1.45e12  # constant
    alpha = k1 * (3 * k5 + 2 * k6) / (k5 + k6)
    beta = k3 * np.sqrt(k1 / (k5 + k6))
    return k1, k2, k3, k4, k5, k6, alpha, beta

# Define a function to calculate [C2H6](t) from the analytical
    solution
def ethane_concentration(t, C1_0, alpha, beta):
    # The analytical solution obtained by substituting u(t) = sqrt
        (C(t))
    # into the transformed ODE leads to:
    return ((-beta/alpha) + np.exp(-alpha/2 * t)*np.sqrt(C1_0) +
        np.exp(-alpha/2 * t)*(beta/alpha))**2

# Function to perform numerical integration (trapezoidal rule)
def integrate_species(t_vals, func, *args):
    return np.array([np.trapz(func(t_vals[:i], *args), t_vals[:i])
         for i in range(1, len(t_vals)+1)])

# Simulation parameters
T = 1100  # Temperature in K
k1, k2, k3, k4, k5, k6, alpha, beta = rate_constants(T)
p = 2e5   # Pressure in Pa
R = 8.3154  # Gas constant in J/(mol K)
C1_0 = p / (R * T * 1e6)  # Adjust units if necessary

time_eval = np.linspace(0, 0.5, 200)  # Time in seconds
C1 = ethane_concentration(time_eval, C1_0, alpha, beta)
C3 = 2 * k1 * integrate_species(time_eval, ethane_concentration,
    C1_0, alpha, beta)
# For ethylene and hydrogen, we assume proper expressions based on
     Equations (\ref{eq:C2H4_int}) and (\ref{eq:H2_int})
C6 = beta * integrate_species(time_eval,
            lambda u, C1_0, alpha, beta: np.sqrt(
                ethane_concentration(u, C1_0, alpha, beta))
            + (k6 / (k3**2)) * ethane_concentration(u, C1_0, alpha,
                beta),
            C1_0, alpha, beta)
C7 = beta * integrate_species(time_eval,
            lambda u, C1_0, alpha, beta: np.sqrt(
                ethane_concentration(u, C1_0, alpha, beta)),
            C1_0, alpha, beta)
C8 = (beta**2 * k5 / k3**2) * integrate_species(time_eval,
    ethane_concentration, C1_0, alpha, beta)

# Plot the results
fig, ax1 = plt.subplots(figsize=(8, 6))
ax2 = ax1.twinx()

ax1.set_ylim([0, 2.5e-5])  # Left y-axis: Ethane, Ethylene,
    Hydrogen
ax2.set_ylim([0, 1e-6])    # Right y-axis: Methane, Butane

ax1.plot(time_eval, C1, label='Ethane(t)', color='black',
    linestyle='-')
ax1.plot(time_eval, C6, label='Ethylene(t)', color='black',
    linestyle='dashed')
ax1.plot(time_eval, C7, label='Hydrogen(t)', color='red',
    linestyle='dashdot')

ax2.plot(time_eval, C3, label='Methane(t)', color='blue',
    linestyle='dotted')
ax2.plot(time_eval, C8, label='Butane(t)', color='green',
    linestyle=':')

ax1.set_xlabel('Time␣(s)')
ax1.set_ylabel('Concentration␣(Ethane,␣Ethylene,␣Hydrogen)')
ax2.set_ylabel('Concentration␣(Methane,␣Butane)')
ax1.set_title('Kinetic␣Curves␣at␣T␣=␣1100␣K')

ax1.legend(loc='upper␣right')
ax2.legend(loc='upper␣left')
ax1.grid()
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

$$\mathrm{A} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{B} \xrightarrow{k_3} \mathrm{C}.$$

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

$$\mathrm{A} \rightleftharpoons \mathrm{B},$$

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 $B \to C$, 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

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

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

$$\mathrm{E} + \mathrm{S} \rightleftharpoons \mathrm{ES},$$

the equilibrium constant is

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

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

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

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.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Define rate constants
k1 = 1.0    # Forward rate constant for A -> B
```

```python
k2 = 0.8    # Reverse rate constant for B -> A
k3 = 0.01   # Slow rate constant for B -> C

# Initial concentrations (in M)
A0 = 0.1
B0 = 0.0
C0 = 0.0

# Define the system of ODEs
def reaction_system(t, y):
    A, B, C = y
    dA_dt = -k1 * A + k2 * B
    dB_dt = k1 * A - (k2 + k3) * B
    dC_dt = k3 * B
    return [dA_dt, dB_dt, dC_dt]

# Time span for simulation
t_span = (0, 15)
t_eval = np.linspace(0, 15, 200)

# Solve the ODEs
solution = solve_ivp(reaction_system, t_span, [A0, B0, C0], t_eval
    =t_eval)
t = solution.t
A_sol, B_sol, C_sol = solution.y

# Compute the quasi-equilibrium approximation for [B]
B_quasi = (k1 / k2) * A_sol

# Plot the results
plt.figure(figsize=(8, 5))
plt.plot(t, B_sol, label=r'Numerical␣$B(t)$', linestyle='-',
    linewidth=2)
plt.plot(t, B_quasi, label=r'Quasi-Equilibrium␣$\frac{k_1}{k_2}A(t
    )$', linestyle='dotted', linewidth=2)
plt.xlabel('Time␣(s)')
plt.ylabel('Concentration␣(M)')
plt.title('Verification␣of␣the␣Quasi-Equilibrium␣Approximation')
plt.legend()
plt.grid(True)
plt.show()
```

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

This script simulates the reaction

$$\mathrm{A} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{B} \xrightarrow{k_3} \mathrm{C},$$

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.

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Define symbolic variables
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)
KM = sp.symbols('KM', positive=True, real=True)

# Step 1: Equilibrium equation for ES complex formation
equilibrium_eq = sp.Eq(k1 * C_S * (C_E0 - C_ES), k_neg1 * C_ES)

# Step 2: Solve for C_ES
C_ES_solution = sp.solve(equilibrium_eq, C_ES)[0]

# Step 3: Define the rate equation
r_P_expr = k2 * C_ES_solution
sp.pprint(r_P_expr)

# Step 4: Express in terms of the Michaelis constant
r_P_MM = k2 * C_E0 * C_S / (C_S + k_neg1/k1)
r_P_final = sp.simplify(r_P_MM.subs(k_neg1/k1, KM))

print("The -MichaelisMenten rate expression is:")
sp.pretty_print(r_P_final)

# -----------------------------------------
# Numerical Evaluation and Plotting
# -----------------------------------------

rP_func = sp.lambdify((C_S, C_E0, k2, KM), r_P_final, 'numpy')

C_E0_val = 1.214e-7   # Total enzyme concentration (M)
k2_val   = 10         # Rate constant k2 (1/s)
KM_val   = 4.815e-4   # Michaelis constant (M)

C_S_values = np.linspace(1e-5, 0.015, 100)
rP_values = rP_func(C_S_values, C_E0_val, k2_val, KM_val)

# Plot 1: Reaction Rate vs. Substrate Concentration
plt.figure(figsize=(6, 4))
plt.plot(C_S_values, rP_values, 'k-', linewidth=2)
plt.axhline(y=k2_val * C_E0_val, linestyle='--', color='gray',
    label=r'$r_{max}$')
plt.xlabel(r'Substrate Concentration, $C_S$ (M)')
plt.ylabel(r'Reaction Rate, $r_P$ (M/s)')
plt.title('Reaction Rate vs. Substrate Concentration')
plt.legend()
plt.grid(True)
plt.show()

# Plot 2: -LineweaverBurk Plot
inv_rP = 1 / rP_values
inv_CS = 1 / C_S_values
plt.figure(figsize=(6, 4))
plt.plot(inv_CS, inv_rP, 'k-', linewidth=2)
plt.xlabel(r'$1/C_S$␣(1/M)')
plt.ylabel(r'$1/r_P$␣(s/M)')
plt.title('-LineweaverBurk␣Plot')
plt.grid(True)
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:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C. \tag{3.53}$$

Activity 1: Numerical Integration with SciPy

```python
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Define the reaction rate constants
k1 = 1.0  # Rate constant for A -> B
```

```python
k2 = 0.5  # Rate constant for B -> C

# Define the system of ODEs
def reaction_system(t, y, k1, k2):
    A, B, C = y
    dA_dt = -k1 * A
    dB_dt = k1 * A - k2 * B
    dC_dt = k2 * B
    return [dA_dt, dB_dt, dC_dt]

# Initial concentrations
A0 = 1.0
B0 = 0.0
C0 = 0.0
y0 = [A0, B0, C0]

# Time span for the simulation
t_span = (0, 10)
t_eval = np.linspace(0, 10, 200)

# Solve the ODE system
solution = solve_ivp(reaction_system, t_span, y0, args=(k1, k2),
    t_eval=t_eval)

# Plot the results
plt.figure(figsize=(8, 5))
plt.plot(solution.t, solution.y[0], label='[A]')
plt.plot(solution.t, solution.y[1], label='[B]')
plt.plot(solution.t, solution.y[2], label='[C]')
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Numerical Integration of A -> B -> C')
plt.legend()
plt.grid(True)
plt.show()
```

Activity 2: Analytical Solution Using Matrix Exponential

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import expm

# Define the reaction rate constants
k1 = 1.0  # Rate constant for A -> B
k2 = 0.5  # Rate constant for B -> C

# Define the rate constant matrix for the system
K = np.array([[-k1,    0,   0],
              [ k1,  -k2,   0],
              [  0,   k2,   0]])

# Initial concentration vector
```

```python
C0 = np.array([1.0, 0.0, 0.0])

# Time points for evaluation
t_points = np.linspace(0, 10, 200)
concentrations = np.zeros((3, len(t_points)))

# Compute the analytical solution using the matrix exponential at
    each time point
for i, t in enumerate(t_points):
    exp_Kt = expm(K * t)
    concentrations[:, i] = exp_Kt.dot(C0)

# Plot the analytical solution
plt.figure(figsize=(8, 5))
plt.plot(t_points, concentrations[0, :], label='[A]␣Analytical')
plt.plot(t_points, concentrations[1, :], label='[B]␣Analytical')
plt.plot(t_points, concentrations[2, :], label='[C]␣Analytical')
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.title('Analytical␣Solution␣via␣Matrix␣Exponential')
plt.legend()
plt.grid(True)
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:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C \xrightarrow{k_3} D. \tag{3.54}$$

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.