# PRD 19: Chemical Reaction Network Theory Autocatalysis

Pure Thought AI Challenge 19

Pure Thought AI Challenges Project

January 18, 2026

**Abstract**

This document presents a comprehensive Product Requirement Document (PRD) for implementing a pure-thought computational challenge. The problem can be tackled using only symbolic mathematics, exact arithmetic, and fresh code—no experimental data or materials databases required until final verification. All results must be accompanied by machine-checkable certificates.

## Contents

**Domain**: Chemistry  Dynamical Systems
**Timeline**: 6-9 months
**Difficulty**: High
**Prerequisites**: Differential equations, graph theory, dynamical systems, algebraic topology, linear programming

---

## 0.1 1. Problem Statement

### 0.1.1 Scientific Context

**Chemical Reaction Networks (CRNs)** provide a mathematical framework for modeling molecular interactions as directed graphs with associated rate laws. Since the pioneering work of Guldberg and Waage (1864) on mass action kinetics, CRN theory has evolved into a sophisticated branch of applied mathematics with deep connections to dynamical systems, graph theory, and algebraic geometry.

The fundamental insight is that chemical dynamics can be encoded in combinatorial-algebraic structures:

- **Species** S = A, B, C, ... represent molecular types
- **Complexes** C = y, y, ... represent linear combinations of species (e.g., A+B, 2C, )
- **Reactions** R  C×C represent transformations (e.g., A+B → 2C with rate constant k)
- **Stoichiometry matrix** S $^{s\ddot{O}r}encodeshowreactionschangespeciesconcentrations$
- **Mass action ODEs** dx/dt = S · v(x, k) govern temporal evolution

**Deficiency Theory** (Feinberg-Horn-Jackson, 1972-1987) provides the cornerstone result: for networks with **deficiency = 0** (where = n - - s, with n = number of complexes, = linkage classes, s = rank of stoichiometry matrix), if the network is weakly reversible, there exists a unique positive equilibrium within each stoichiometric compatibility class, and this equilibrium is locally asymptotically stable. Networks with > 0 can exhibit multistationarity, oscillations, and chaos.

**Autocatalysis** — chemical reactions where products catalyze their own formation — is central to the origin of life, metabolic cycles, and self-replicating systems. **Reflexively Autocatalytic and Food-generated (RAF) sets** (Hordijk  Steel, 2004) formalize the idea of self-sustaining chemical systems: a subset of reactions is RAF if every reaction is catalyzed by a molecule produced within the set, starting from a "food set" of available substrates.

**Multistationarity** (multiple stable steady states) underlies biochemical switches, cell differentiation, and decision-making in regulatory networks. Determining whether a given CRN admits multiple positive equilibria is algorithmically challenging, but can be attacked using:

- **Gröbner bases** to solve polynomial steady-state equations
- **Sign conditions** on Jacobian determinants (Feinberg's deficiency theorems)
- **Injectivity criteria** (Banaji-Pantea, Craciun-Feinberg)

**Persistence** ensures that if all species start with positive concentrations, they remain positive for all time (no extinction). This is crucial for biological realism and can be certified using conservation laws, siphon analysis, and linear programming.

### 0.1.2 Core Question

**Can we algorithmically determine stability, autocatalysis, multistationarity, and persistence for CRNs using ONLY graph-theoretic analysis, symbolic algebra, and linear programming — without numerical simulation of ODEs?**

Specifically:

- Given a CRN, compute its **deficiency** and predict equilibrium behavior via Feinberg-Horn-Jackson theory
- Identify **autocatalytic cycles** and **RAF sets** using graph algorithms
- Find **all positive steady states** by solving polynomial equations with Gröbner bases
- Certify **persistence** via conservation laws, siphons, and P/T-invariants
- Detect **multistationarity** using algebraic criteria (discriminant analysis, sign patterns)
- Generate **machine-checkable certificates** for each property

### 0.1.3 Why This Matters

- **Origin of Life**: Autocatalytic sets provide the simplest models for self-replicating chemical systems that could bootstrap life from prebiotic chemistry
- **Systems Biology**: CRN models capture gene regulatory networks, signaling cascades, and metabolic pathways; deficiency theory predicts bistability in cell fate decisions
- **Synthetic Biology**: Designing engineered genetic circuits requires understanding when networks exhibit desired behaviors (oscillations, switches, robust homeostasis)
- **Drug Discovery**: Pharmacodynamic models are CRNs; persistence guarantees that therapies won't cause complete extinction of cell populations

### 0.1.4 Pure Thought Advantages

- **Exact symbolic methods**: Gröbner bases find ALL steady states (not just numerically accessible ones); deficiency is computed exactly from graph structure
- **Certificates**: Persistence can be certified via LP duality; injectivity via Jacobian sign patterns; these are machine-verifiable
- **No parameter dependence**: Graph-theoretic properties (deficiency, linkage classes, siphons) are independent of rate constants; results hold for entire parameter families
- **Scalability**: Symbolic computation handles networks with hundreds of species; numerical ODE solvers struggle with stiff systems and missed bifurcations

## 0.2 2. Mathematical Formulation

### 0.2.1 CRN Structure

**Definition (Chemical Reaction Network)**: A CRN is a triple (S, C, R) where:

- S = X, ..., $X_s isa finite set of$ **species**
- C $^s isa finite set of$ **complexes**$(yC isa non-negative integer vector)$
- R C × C is a set of **reactions** (ordered pairs (y, y') written y → y')

**Example** (Brusselator):

- Species: S = A, B, X, Y, C, D
- Complexes: C = A, 2X+Y, B+X, X, 3X, Y+C, D,
- Reactions: A→X, 2X+Y→3X, B+X→Y+C, X→D

**Stoichiometry Matrix**: S $^{sÖr} where r = |R| and S_{ik} = (stoich of species i in product) - (stoich in reactant) for reaction k.$

**Mass Action Kinetics**:

$$\frac{dx}{dt} = S \cdot v(x) \quad \text{where} \quad vk(x) = \kappa k(x) = \kappa k(x) = \kappa_{k(x) = \kappa k} \prod$$

### 0.2.2 Deficiency Theory

**Reaction Graph**: G = (C, R) is a directed graph with vertex set C (complexes) and edge set R (reactions).

**Linkage Classes**: Weakly connected components of G (denoted ).

**Stoichiometric Subspace**: S = Im(S) $^s (ranks = dim S)$.

**Deficiency**:

$$\delta = n - \ell - s$$

where n = |C| (number of complexes), = number of linkage classes, s = rank(S).

**Weakly Reversible**: Every linkage class is strongly connected (for every reaction y→y', there's a path y' y in the same linkage class).

**Feinberg-Horn-Jackson Theorem**: If = 0 and the network is weakly reversible, then:

- There exists a unique positive equilibrium x* in each stoichiometric compatibility class (x + S).
- x* is locally asymptotically stable within its class.

### 0.2.3 Conservation Laws and Stoichiometric Compatibility

**Conservation Laws**: Vectors c ker($S^T$) $satisfy c^T S = 0, implying c^T x(t) = c^T x(0) for all t.$

**Stoichiometric Compatibility Class**:

$[x0] = \{x \in \mathbb{R}0] = \{x \in \mathbb{R}0] = \{x \in \mathbb{R}_{0]} = \{x \in \mathbb{R} > 0^s : x = x_0 + S\eta \text{ for some } \eta \in \mathbb{R}^r\}$ *All trajectories starting at x remain in [x].* **0.2.3 Autocatalyt**

**RAF Set** (Hordijk-Steel): A subset R' R of reactions is **reflexively autocatalytic and food-generated** if:

- Every reaction in R' is catalyzed by at least one molecule that is either in the food set F or produced by R' itself
- Every reactant for reactions in R' is either in F or produced by R'

**Graph Algorithm**: RAF detection reduces to finding a subgraph where every edge (reaction) has an incoming edge from a catalyzed species.

### 0.2.4 Multistationarity

**Injectivity**: A network is **injective** on a region  if the map x  S · v(x) is injective. Injective networks have at most one positive equilibrium per stoichiometric class.

**Criteria**:

- **Deficiency-One Algorithm** (Feinberg): For =1 networks, multistationarity can be ruled out by checking sign patterns of certain determinants
- **Gröbner Basis**: Steady-state equation S · v(x)=0 is a polynomial system; number of positive real roots gives number of equilibria

### 0.2.5 Persistence

**Definition**: A CRN is **persistent** if for all x(0)  $^{s}$>0, *we have lim inf*ß$x_i(t) > 0 for all species i$.

**Siphon**: A subset Z  S of species is a **siphon** if every reaction producing a species in Z consumes at least one species in Z. If a siphon is emptied (all concentrations zero), it stays empty.

**Persistence Theorem**: A CRN is persistent if and only if no siphon can be emptied from positive initial conditions.

### 0.2.6 Certificate Specification

A **valid certificate** for a CRN analysis must include:

- **Deficiency Certificate**:
- Adjacency matrix of reaction graph G
- List of weakly connected components (linkage classes)
- Stoichiometry matrix S and its rank (via row echelon form)
- Deficiency  = n -  - s
- **Steady State Certificate** (for each equilibrium x*):
- Polynomial system S · v(x) = 0 with Gröbner basis
- Rational (or algebraic) coordinates of x*
- Verification that S · v(x*) = 0 (residual < $10^{-50}$)

- Positivity: $x^*_i > 0 for all i$
- **Persistence Certificate**:
- List of all minimal siphons Z, ..., $Z_k$

- For each siphon $Z_j, proofthatit'snon-emptiable$ :
- Conservation law c with $c^T(speciesinZ_j) > 0$
- Or: Show that every reaction emptying Z$j$ *requires positive concentration outside* $Z$j
- **Multistationarity Certificate** (if multiple equilibria exist):
- Two distinct positive solutions x, $x$ with residuals $< 10^{-50}$
- Proof they're in same stoichiometric class: x - $x$ Im(S)
- **Autocatalysis Certificate**:
- List of cycles in reaction graph containing net production
- RAF set R' with catalysis graph showing each reaction catalyzed by product of R'

**Export Format**: JSON with exact rational arithmetic (numerator/denominator), Gröbner bases in SymPy string format.

---

## 0.3 3. Implementation Approach

### 0.3.1 Phase 1: CRN Construction and Graph Analysis (Months 1-2)

**Goal**: Build CRN data structures, compute reaction graph, identify linkage classes.

```
import networkx as nx
import numpy as np
from sympy import *
from typing import List, Dict, Tuple, Set
from fractions import Fraction

class Complex:
    """Represents a chemical complex as a dict {species:
       stoichiometry}."""
    def __init__(self, composition: Dict[str, int]):
        self.composition = {k: v for k, v in composition.items() if v >
           0}

    def __hash__(self):
        return hash(frozenset(self.composition.items()))

    def __eq__(self, other):
        return self.composition == other.composition

    def __repr__(self):
        if not self.composition:
            return "   "
        terms = [f"{v}{k}" if v > 1 else k for k, v in
           sorted(self.composition.items())]
        return "+".join(terms)

    def to_vector(self, species_list: List[str]) -> np.ndarray:
        """Convert to stoichiometry vector."""
```

```
        return np.array([self.composition.get(s, 0) for s in
            species_list], dtype=int)


class Reaction:
    """Represents a reaction: reactant     product with rate
        constant."""
    def __init__(self, reactant: Complex, product: Complex, rate:
        Symbol):
        self.reactant = reactant
        self.product = product
        self.rate = rate

    def __repr__(self):
        return f"{self.reactant}     {self.product} (rate {self.rate})"

    def stoichiometry(self, species_list: List[str]) -> np.ndarray:
        """Net change in species concentrations."""
        return self.product.to_vector(species_list) -
            self.reactant.to_vector(species_list)


class ChemicalReactionNetwork:
    """
    CRN with species, complexes, reactions.
    """
    def __init__(self, species: List[str]):
        self.species = species
        self.complexes: List[Complex] = []
        self.reactions: List[Reaction] = []
        self.complex_to_index: Dict[Complex, int] = {}

    def add_reaction(self, reactant: Complex, product: Complex, rate:
        Symbol):
        """Add reaction and register complexes."""
        if reactant not in self.complex_to_index:
            self.complex_to_index[reactant] = len(self.complexes)
            self.complexes.append(reactant)

        if product not in self.complex_to_index:
            self.complex_to_index[product] = len(self.complexes)
            self.complexes.append(product)

        self.reactions.append(Reaction(reactant, product, rate))

    def stoichiometry_matrix(self) -> np.ndarray:
        """Build s    r stoichiometry matrix."""
        s = len(self.species)
        r = len(self.reactions)
        S = np.zeros((s, r), dtype=int)

        for k, rxn in enumerate(self.reactions):
            S[:, k] = rxn.stoichiometry(self.species)

        return S

```

```
    def reaction_graph(self) -> nx.DiGraph:
        """Build directed graph G = (C, R)."""
        G = nx.DiGraph()
        for i in range(len(self.complexes)):
            G.add_node(i, label=str(self.complexes[i]))

        for rxn in self.reactions:
            i = self.complex_to_index[rxn.reactant]
            j = self.complex_to_index[rxn.product]
            G.add_edge(i, j, rate=rxn.rate)

        return G

    def linkage_classes(self) -> List[Set[int]]:
        """Compute weakly connected components."""
        G = self.reaction_graph()
        return [set(comp) for comp in nx.weakly_connected_components(G)]

    def compute_deficiency(self) -> int:
        """
           = n -     - s
        n = number of complexes
            = number of linkage classes
        s = rank of stoichiometry matrix
        """
        n = len(self.complexes)
            = len(self.linkage_classes())

        S = self.stoichiometry_matrix()
        s = np.linalg.matrix_rank(S)

           = n -     - s

        return

    def is_weakly_reversible(self) -> bool:
        """Check if every linkage class is strongly connected."""
        G = self.reaction_graph()

        for linkage_class in self.linkage_classes():
            subgraph = G.subgraph(linkage_class)
            if not nx.is_strongly_connected(subgraph):
                return False

        return True


# Example: Brusselator
def brusselator_crn() -> ChemicalReactionNetwork:
    """
    A     X (rate k1)
    2X + Y     3X (rate k2)
    B + X     Y + C (rate k3)
    X     D (rate k4)
    """
    species = ['A', 'B', 'X', 'Y', 'C', 'D']
```

```
    crn = ChemicalReactionNetwork(species)

    k1, k2, k3, k4 = symbols('k1 k2 k3 k4', positive=True, real=True)

    A = Complex({'A': 1})
    B = Complex({'B': 1})
    X = Complex({'X': 1})
    Y = Complex({'Y': 1})
    C = Complex({'C': 1})
    D = Complex({'D': 1})
    X2Y = Complex({'X': 2, 'Y': 1})
    X3 = Complex({'X': 3})
    BX = Complex({'B': 1, 'X': 1})
    YC = Complex({'Y': 1, 'C': 1})

    crn.add_reaction(A, X, k1)
    crn.add_reaction(X2Y, X3, k2)
    crn.add_reaction(BX, YC, k3)
    crn.add_reaction(X, D, k4)

    return crn


def print_crn_info(crn: ChemicalReactionNetwork):
    """Display CRN structure."""
    print("Species:", crn.species)
    print("Complexes:", [str(c) for c in crn.complexes])
    print("Reactions:")
    for rxn in crn.reactions:
        print(f"  {rxn}")

    print(f"\nStoichiometry Matrix S:")
    S = crn.stoichiometry_matrix()
    print(S)

    print(f"\nDeficiency     = {crn.compute_deficiency()}")
    print(f"Weakly reversible: {crn.is_weakly_reversible()}")
    print(f"Linkage classes: {crn.linkage_classes()}")
```

**Test Case**: Verify Brusselator has  = 1, two linkage classes.

---

### 0.3.2 Phase 2: Deficiency Theory and FHJ Theorem (Months 2-3)

**Goal**: Implement Feinberg-Horn-Jackson predictions for equilibrium behavior.

```
def conservation_laws(crn: ChemicalReactionNetwork) -> np.ndarray:
    """
    Compute basis for ker(S^T).
    These are conservation laws: c^T x(t) = constant.
    """
    S = crn.stoichiometry_matrix()
    # Symbolic null space for exact arithmetic
    S_sym = Matrix(S.T)
    kernel = S_sym.nullspace()

```

```
    if not kernel:
        return np.array([]).reshape(0, len(crn.species))

    # Convert to numpy (rational)
    C = np.array([[Fraction(int(val.p), int(val.q)) for val in vec]
                  for vec in kernel], dtype=object)

    return C


def stoichiometric_subspace_dimension(crn: ChemicalReactionNetwork) ->
    int:
    """Dimension of Im(S) = rank(S)."""
    S = crn.stoichiometry_matrix()
    return np.linalg.matrix_rank(S)


def feinberg_horn_jackson_analysis(crn: ChemicalReactionNetwork) ->
    Dict:
    """
    Apply FHJ deficiency theorems.

    Returns:
        - deficiency
        - prediction: "unique equilibrium", "multistationarity
          possible", etc.
    """
       = crn.compute_deficiency()
    weakly_rev = crn.is_weakly_reversible()

    analysis = {
        'deficiency':   ,
        'weakly_reversible': weakly_rev,
        'num_linkage_classes': len(crn.linkage_classes()),
        'stoichiometric_subspace_dim':
            stoichiometric_subspace_dimension(crn)
    }

    if    == 0 and weakly_rev:
        analysis['prediction'] = "Unique positive equilibrium per
            stoichiometric class (FHJ Theorem)"
        analysis['stability'] = "Locally asymptotically stable"
    elif    == 0 and not weakly_rev:
        analysis['prediction'] = "At most one equilibrium per class,
            but may have none"
        analysis['stability'] = "Unknown"
    elif    == 1:
        analysis['prediction'] = "Deficiency-one: apply advanced
            criteria for multistationarity"
        analysis['stability'] = "Possible multistationarity or limit
            cycles"
    else:
        analysis['prediction'] = f"Deficiency {  } > 1: complex
            dynamics possible"
        analysis['stability'] = "Multistationarity, oscillations, chaos
            possible"
```

```
    return analysis


# Example application
if __name__ == "__main__":
    crn = brusselator_crn()
    analysis = feinberg_horn_jackson_analysis(crn)

    print("=== Feinberg-Horn-Jackson Analysis ===")
    for key, value in analysis.items():
        print(f"{key}: {value}")

    print("\n=== Conservation Laws ===")
    C = conservation_laws(crn)
    if C.size > 0:
        for i, c in enumerate(C):
            terms = [f"{c[j]}  {crn.species[j]}" for j in
                range(len(crn.species)) if c[j] != 0]
            print(f"Conservation law {i+1}: {' + '.join(terms)} =
                constant")
    else:
        print("No conservation laws (stoichiometric subspace is
            full-dimensional)")
```

**Certificate**: Export deficiency, linkage classes, conservation laws as JSON with exact rational arithmetic.

---

### 0.3.3 Phase 3: Autocatalysis and RAF Set Detection (Months 3-4)

**Goal**: Identify autocatalytic cycles and reflexively autocatalytic sets.

```
def find_autocatalytic_cycles(crn: ChemicalReactionNetwork) ->
    List[List[int]]:
    """
    Find cycles in reaction graph where net production > 0 for some
        species.

    Autocatalytic cycle: cycle C in reaction graph with  _ {rxn     C}
        stoich > 0.
    """
    G = crn.reaction_graph()
    S = crn.stoichiometry_matrix()

    autocatalytic = []

    # Find all simple cycles
    for cycle_complexes in nx.simple_cycles(G):
        # Get reactions in this cycle
        cycle_edges = []
        for i in range(len(cycle_complexes)):
            u = cycle_complexes[i]
            v = cycle_complexes[(i+1) % len(cycle_complexes)]
            # Find reaction index
            for k, rxn in enumerate(crn.reactions):
```

```
                if (crn.complex_to_index[rxn.reactant] == u and
                    crn.complex_to_index[rxn.product] == v):
                    cycle_edges.append(k)
                    break

        # Net stoichiometry
        net_change = np.sum(S[:, cycle_edges], axis=1)

        # Autocatalytic if any species has net production
        if np.any(net_change > 0):
            autocatalytic.append({
                'cycle_complexes': [crn.complexes[i] for i in
                    cycle_complexes],
                'cycle_reactions': [crn.reactions[k] for k in
                    cycle_edges],
                'net_production': {crn.species[i]: int(net_change[i])
                                   for i in range(len(crn.species)) if
                                       net_change[i] != 0}
            })

    return autocatalytic


def detect_raf_set(crn: ChemicalReactionNetwork, food_set: Set[str]) ->
    List[Set[int]]:
    """
    Find RAF sets: subsets of reactions that are reflexively
        autocatalytic
    and food-generated.

    Algorithm:
    1. Build catalysis graph: reaction r is catalyzed by species s
    2. Iteratively add reactions whose catalysts are available (from
        food or prior reactions)
    3. Check if closure is self-sustaining
    """
    # For simplicity, assume every species catalyzes all reactions it
        appears in as a product
    # (Real biochemical networks have explicit catalysis annotations)

    def products_of_reactions(reaction_indices: Set[int]) -> Set[str]:
        """Species produced by given reactions."""
        produced = set(food_set)
        for k in reaction_indices:
            rxn = crn.reactions[k]
            for species in rxn.product.composition.keys():
                produced.add(species)
        return produced

    def can_run(reaction_index: int, available_species: Set[str]) ->
        bool:
        """Can this reaction run given available species (as
            reactants)?"""
        rxn = crn.reactions[reaction_index]
        return all(s in available_species for s in
            rxn.reactant.composition.keys())
```

```
    # Iterative closure
    current_raf = set()
    available = set(food_set)

    changed = True
    while changed:
        changed = False
        for k in range(len(crn.reactions)):
            if k not in current_raf and can_run(k, available):
                current_raf.add(k)
                available.update(products_of_reactions({k}))
                changed = True

    # Check reflexivity: every reaction in RAF is catalyzed by product
       of RAF
    # (Simplified: we assume a reaction is autocatalytic if its
       products include its reactants)
    raf_reactions = [crn.reactions[k] for k in current_raf]

    return [current_raf] if current_raf else []


# Example: Formose reaction (autocatalytic sugar synthesis)
def formose_reaction_crn() -> ChemicalReactionNetwork:
    """
    Simplified formose reaction:
    2 CH2O      C2H4O2 (glycolaldehyde)
    CH2O + C2H4O2      C3H6O3 (glyceraldehyde)
    C2H4O2 + C2H4O2      C4H8O4 (erythrose)
    C4H8O4      2 C2H4O2 (autocatalytic feedback)
    """
    species = ['CH2O', 'C2H4O2', 'C3H6O3', 'C4H8O4']
    crn = ChemicalReactionNetwork(species)

    k1, k2, k3, k4 = symbols('k1 k2 k3 k4', positive=True)

    CH2O = Complex({'CH2O': 1})
    C2H4O2 = Complex({'C2H4O2': 1})
    C3H6O3 = Complex({'C3H6O3': 1})
    C4H8O4 = Complex({'C4H8O4': 1})
    CH2O_2 = Complex({'CH2O': 2})
    C2H4O2_2 = Complex({'C2H4O2': 2})
    CH2O_C2H4O2 = Complex({'CH2O': 1, 'C2H4O2': 1})

    crn.add_reaction(CH2O_2, C2H4O2, k1)
    crn.add_reaction(CH2O_C2H4O2, C3H6O3, k2)
    crn.add_reaction(C2H4O2_2, C4H8O4, k3)
    crn.add_reaction(C4H8O4, C2H4O2_2, k4)  # Autocatalytic

    return crn
```

**Output**: List of autocatalytic cycles with net production; RAF sets showing self-sustaining reaction subnetworks.

### 0.3.4 Phase 4: Steady State Analysis with Gröbner Bases (Months 4-6)

**Goal**: Find all positive equilibria by solving polynomial steady-state equations.

```
from sympy import groebner, solve, symbols, simplify
from sympy.polys.polytools import poly

def mass_action_odes(crn: ChemicalReactionNetwork) -> List[Expr]:
    """
    Symbolic mass action ODEs: dx/dt = S    v(x).

    Returns list of expressions for d[X_i]/dt.
    """
    # Symbolic concentration variables
    x = symbols(' '.join([f'x_{s}' for s in crn.species]),
        positive=True, real=True)
    x_dict = {crn.species[i]: x[i] for i in range(len(crn.species))}

    S = Matrix(crn.stoichiometry_matrix())

    # Rate vector v
    v = []
    for rxn in crn.reactions:
        rate_expr = rxn.rate
        for species, stoich in rxn.reactant.composition.items():
            rate_expr *= x_dict[species]**stoich
        v.append(rate_expr)

    v_vec = Matrix(v)

    # dx/dt = S    v
    dx_dt = S * v_vec

    return [simplify(dx_dt[i]) for i in range(len(crn.species))], x


def find_positive_equilibria(crn: ChemicalReactionNetwork,
                             use_groebner: bool = True) -> List[Dict]:
    """
    Solve dx/dt = 0 for positive real solutions.

    Uses Gr bner bases for exact symbolic solutions.
    """
    dx_dt, x = mass_action_odes(crn)

    # Steady state equations
    equations = [expr for expr in dx_dt]

    print(f"Solving {len(equations)} polynomial equations in {len(x)}
        variables...")

    if use_groebner:
        # Gr bner basis (can be slow for large systems)
        try:
            G = groebner(equations, x, order='lex')
            print(f"Gr bner basis computed: {len(G)} polynomials")

```

```
            # Solve reduced system
            solutions = solve(G, x, dict=True)
        except Exception as e:
            print(f"Gr bner basis failed: {e}, falling back to direct solve")
            solutions = solve(equations, x, dict=True)
    else:
        solutions = solve(equations, x, dict=True)

    # Filter for positive real solutions
    positive_equilibria = []

    for sol in solutions:
        # Check if all values are positive and real
        try:
            is_positive = all(val.is_positive for val in sol.values() if val.is_real)
            is_real = all(val.is_real for val in sol.values())

            if is_positive and is_real:
                # Convert to numerical values for verification
                sol_numeric = {str(var): complex(val.evalf()) for var, val in sol.items()}

                # Verify it's a true solution
                residual = verify_equilibrium(crn, sol)

                positive_equilibria.append({
                    'symbolic': sol,
                    'numeric': sol_numeric,
                    'residual': residual
                })
        except Exception as e:
            print(f"Skipping solution due to evaluation error: {e}")
            continue

    return positive_equilibria


def verify_equilibrium(crn: ChemicalReactionNetwork, solution: Dict) -> float:
    """
    Verify that solution satisfies dx/dt = 0.

    Returns maximum absolute residual.
    """
    dx_dt, x = mass_action_odes(crn)

    # Substitute solution
    residuals = [abs(eq.subs(solution).evalf()) for eq in dx_dt]

    return max(residuals)


# Example: Find equilibria for simple bistable network
def schloegl_crn() -> ChemicalReactionNetwork:
```

```
    """
    Schl gl model (canonical bistable system):
    A + 2X     3X (rate k1)
    3X     A + 2X (rate k2)
    B     X (rate k3)
    X     C (rate k4)

    For suitable parameters, has two stable equilibria.
    """
    species = ['A', 'B', 'X', 'C']
    crn = ChemicalReactionNetwork(species)

    k1, k2, k3, k4 = symbols('k1 k2 k3 k4', positive=True)

    A = Complex({'A': 1})
    B = Complex({'B': 1})
    X = Complex({'X': 1})
    C = Complex({'C': 1})
    A_2X = Complex({'A': 1, 'X': 2})
    X3 = Complex({'X': 3})
    A_X2 = Complex({'A': 1, 'X': 2})

    crn.add_reaction(A_2X, X3, k1)
    crn.add_reaction(X3, A_X2, k2)
    crn.add_reaction(B, X, k3)
    crn.add_reaction(X, C, k4)

    return crn
```

**Certificate**: For each equilibrium, export symbolic solution, numerical values (high precision), and residual verification.

---

### 0.3.5 Phase 5: Persistence via Siphon Analysis (Months 6-7)

**Goal**: Certify that no species goes extinct from positive initial conditions.

```
def find_siphons(crn: ChemicalReactionNetwork) -> List[Set[str]]:
    """
    Find all minimal siphons.

    Siphon Z: every reaction producing a species in Z consumes a
       species in Z.
    If Z is empty, it stays empty.
    """
    # Brute force: check all subsets (exponential, only feasible for
       small networks)
    from itertools import combinations

    all_species = set(crn.species)
    siphons = []

    for size in range(1, len(crn.species) + 1):
        for subset in combinations(crn.species, size):
            Z = set(subset)

```

```
            if is_siphon(crn, Z):
                # Check minimality
                is_minimal = True
                for s in Z:
                    if is_siphon(crn, Z - {s}):
                        is_minimal = False
                        break

                if is_minimal:
                    siphons.append(Z)

    return siphons


def is_siphon(crn: ChemicalReactionNetwork, Z: Set[str]) -> bool:
    """
    Check if Z is a siphon:
    For every reaction producing a species in Z, at least one reactant
       is in Z.
    """
    for rxn in crn.reactions:
        produces_Z = any(s in Z for s in rxn.product.composition.keys())

        if produces_Z:
            consumes_Z = any(s in Z for s in
               rxn.reactant.composition.keys())
            if not consumes_Z:
                return False  # Violates siphon condition

    return True


def certify_persistence_via_conservation(crn: ChemicalReactionNetwork)
   -> Dict:
    """
    Certify persistence using conservation laws.

    A siphon Z cannot be emptied if there exists conservation law c with
    c^T    (species in Z) > 0 and c_i     0 for all i.
    """
    C = conservation_laws(crn)
    siphons = find_siphons(crn)

    non_emptiable_siphons = []
    emptiable_siphons = []

    for Z in siphons:
        # Check if any conservation law prevents emptying Z
        is_protected = False

        for c in C:
            # Sum of conservation law coefficients over species in Z
            Z_sum = sum(c[i] for i, s in enumerate(crn.species) if s in
               Z)

            # If Z_sum > 0 and all c_i >= 0, then Z cannot be emptied
```

```
            if Z_sum > 0 and all(c[i] >= 0 for i in
               range(len(crn.species))):
                is_protected = True
                break

        if is_protected:
            non_emptiable_siphons.append(Z)
        else:
            emptiable_siphons.append(Z)

    is_persistent = (len(emptiable_siphons) == 0)

    return {
        'persistent': is_persistent,
        'total_siphons': len(siphons),
        'non_emptiable_siphons': non_emptiable_siphons,
        'emptiable_siphons': emptiable_siphons,
        'conservation_laws': C
    }


# Example persistence check
if __name__ == "__main__":
    crn = brusselator_crn()
    persistence = certify_persistence_via_conservation(crn)

    print("=== Persistence Analysis ===")
    print(f"Network is persistent: {persistence['persistent']}")
    print(f"Total siphons found: {persistence['total_siphons']}")
    print(f"Non-emptiable siphons:
       {persistence['non_emptiable_siphons']}")
    print(f"Emptiable siphons: {persistence['emptiable_siphons']}")
```

**Certificate**: List all minimal siphons; for each non-emptiable siphon, provide conservation law proving it can't be emptied.

---

### 0.3.6 Phase 6: Certificate Generation and Export (Months 7-9)

**Goal**: Generate machine-checkable certificates for all CRN properties.

```
import json
from fractions import Fraction

def export_crn_certificate(crn: ChemicalReactionNetwork,
                           equilibria: List[Dict],
                           persistence_data: Dict,
                           autocatalytic_cycles: List[Dict],
                           output_file: str):
    """
    Export complete CRN analysis as JSON certificate.
    """
    # Deficiency data
    deficiency_cert = {
        'num_complexes': len(crn.complexes),
        'num_linkage_classes': len(crn.linkage_classes()),
```

```
        'stoichiometric_subspace_dim':
            stoichiometric_subspace_dimension(crn),
        'deficiency': crn.compute_deficiency(),
        'weakly_reversible': crn.is_weakly_reversible()
    }

    # Stoichiometry matrix (as list of lists with exact integers)
    S = crn.stoichiometry_matrix().tolist()

    # Conservation laws (exact rational)
    C = conservation_laws(crn)
    conservation_cert = []
    for c in C:
        conservation_cert.append([str(val) for val in c])  # Fraction
            strings

    # Equilibria (symbolic + numeric)
    equilibria_cert = []
    for eq in equilibria:
        eq_cert = {
            'symbolic': {str(k): str(v) for k, v in
                eq['symbolic'].items()},
            'numeric': {k: str(v) for k, v in eq['numeric'].items()},
            'residual': float(eq['residual'])
        }
        equilibria_cert.append(eq_cert)

    # Persistence
    persistence_cert = {
        'persistent': persistence_data['persistent'],
        'siphons': [list(Z) for Z in
            persistence_data['non_emptiable_siphons'] +
                    persistence_data['emptiable_siphons']],
        'emptiable_siphons': [list(Z) for Z in
            persistence_data['emptiable_siphons']]
    }

    # Autocatalysis
    autocatalysis_cert = []
    for cycle in autocatalytic_cycles:
        autocatalysis_cert.append({
            'cycle_complexes': [str(c) for c in
                cycle['cycle_complexes']],
            'net_production': cycle['net_production']
        })

    # Complete certificate
    certificate = {
        'crn_name': 'Analyzed CRN',
        'species': crn.species,
        'reactions': [str(rxn) for rxn in crn.reactions],
        'stoichiometry_matrix': S,
        'deficiency_analysis': deficiency_cert,
        'conservation_laws': conservation_cert,
        'equilibria': equilibria_cert,
        'persistence': persistence_cert,
```

```
        'autocatalytic_cycles': autocatalysis_cert,
        'certificate_version': '1.0',
        'verification': {
            'all_equilibria_verified': all(eq['residual'] < 1e-10 for
               eq in equilibria),
            'persistence_certified': persistence_data['persistent']
        }
    }

    with open(output_file, 'w') as f:
        json.dump(certificate, f, indent=2)

    print(f"Certificate exported to {output_file}")
    return certificate


# Generate certificate for example network
if __name__ == "__main__":
    crn = brusselator_crn()

    print("Analyzing Brusselator CRN...")

    # Compute all properties
    equilibria = find_positive_equilibria(crn, use_groebner=False)  #
       May be slow
    persistence = certify_persistence_via_conservation(crn)
    cycles = find_autocatalytic_cycles(crn)

    # Export
    cert = export_crn_certificate(
        crn,
        equilibria,
        persistence,
        cycles,
        'brusselator_certificate.json'
    )

    print("\n=== Certificate Summary ===")
    print(f"Deficiency: {cert['deficiency_analysis']['deficiency']}")
    print(f"Equilibria found: {len(cert['equilibria'])}")
    print(f"Persistent: {cert['persistence']['persistent']}")
    print(f"Autocatalytic cycles: {len(cert['autocatalytic_cycles'])}")
```

**Verification Script**:

```
def verify_crn_certificate(cert_file: str) -> bool:
    """
    Verify all claims in a CRN certificate.
    """
    with open(cert_file, 'r') as f:
        cert = json.load(f)

    print("=== Verifying Certificate ===")

    # Check deficiency calculation
    n = cert['deficiency_analysis']['num_complexes']
        = cert['deficiency_analysis']['num_linkage_classes']
```

```
    s = cert['deficiency_analysis']['stoichiometric_subspace_dim']
      _claimed   = cert['deficiency_analysis']['deficiency']
      _verified   = n -     - s

    assert   _claimed   ==   _verified  , f"Deficiency mismatch:
       {  _claimed  }     {  _verified  }"
    print(f"    Deficiency verified:    = {  _claimed  }")

    # Check stoichiometry matrix rank
    S = np.array(cert['stoichiometry_matrix'])
    rank_verified = np.linalg.matrix_rank(S)
    assert rank_verified == s, f"Rank mismatch: {rank_verified}     {s}"
    print(f"    Stoichiometry rank verified: {s}")

    # Verify equilibria (check residuals)
    for i, eq in enumerate(cert['equilibria']):
        residual = eq['residual']
        assert residual < 1e-8, f"Equilibrium {i} has large residual:
           {residual}"
    print(f"    All {len(cert['equilibria'])} equilibria verified")

    # Persistence check
    if cert['persistence']['persistent']:
        assert len(cert['persistence']['emptiable_siphons']) == 0
        print("    Persistence verified (no emptiable siphons)")

    print("\ n   ALL CHECKS PASSED")
    return True
```

---

### 0.4 4. Example Starting Prompt

```
You are a mathematical chemist studying Chemical Reaction Network
   Theory. Your task is to
analyze the stability, autocatalysis, and multistationarity of reaction
   networks using ONLY
graph-theoretic methods, symbolic algebra, and linear programming
   NO numerical ODE simulation.

OBJECTIVE: Analyze the **Brusselator** reaction network and generate a
   complete certificate
of its dynamical properties.

NETWORK SPECIFICATION:
Species: {A, B, X, Y, C, D}
Reactions:
  R1: A     X (rate  k  )
  R2: 2X + Y     3X (rate  k  )  [Autocatalytic]
  R3: B + X     Y + C (rate  k  )
  R4: X     D (rate  k  )

PHASE 1 (Months 1-2): CRN Construction and Graph Analysis
- Implement Complex and Reaction classes
- Build stoichiometry matrix S (6 species    4 reactions)
```

```
- Construct reaction graph G with complexes as vertices
- Identify linkage classes (weakly connected components)
- Expected: 2 linkage classes, 8 complexes

PHASE 2 (Months 2-3): Deficiency Theory
- Compute deficiency    = n -     - s where n=8 complexes,    =2
    linkage classes
- Calculate rank(S) using row echelon form (expect s=5)
- Predict    = 8 - 2 - 5 = 1 (deficiency-one network)
- Check weak reversibility (expect FALSE: not all linkage classes
    strongly connected)
- Apply FHJ theorem:   =1, not weakly reversible     multistationarity
    possible

PHASE 3 (Months 3-4): Autocatalysis Detection
- Find all cycles in reaction graph
- For each cycle, compute net stoichiometry    S_k
- Identify autocatalytic cycle containing R2: 2X+Y     3X (net
    production of X)
- Check for RAF sets with food set F = {A, B}
- Expected: {R1, R2} forms RAF (R1 produces X, R2 autocatalyzes X)

PHASE 4 (Months 4-6): Steady State Analysis
- Formulate mass action ODEs symbolically:
  dX/dt =  k  [A] +  k  [X]  [Y] -  k  [B][X] -  k  [X]
  dY/dt =  k  [B][X] -  k  [X]  [Y]
  (Assuming A, B constant, ignoring C, D)
- Set dX/dt = dY/dt = 0
- Solve using SymPy:
  X* =  k  [A]/ k
  Y* =  k  [B]/ k      ( k  /( k  [A]))
- Verify steady state substitution yields residuals < 10^{-50}
- Check Jacobian eigenvalues for stability (expect Hopf bifurcation
    possible)

PHASE 5 (Months 6-7): Persistence Analysis
- Find all minimal siphons (subsets of species that stay empty once
    emptied)
- Expected siphons: {X, Y} (if X and Y both zero, they stay zero)
- Check conservation laws: compute ker(S^T)
- Expected: A + C + X = const, B + Y + C = const, C + D = const
- Certify persistence: conservation laws prevent {X,Y} siphon from
    being emptied
  if initial conditions have  X  ,  Y  > 0 and  A  ,  B  > 0

PHASE 6 (Months 7-9): Certificate Generation
- Export JSON with:
  * Deficiency    = 1
  * Stoichiometry matrix S (exact integers)
  * Equilibrium (X*, Y*) with symbolic expressions and residual <
      10^{-50}
  * Autocatalytic cycle: [R2] with net production {X: +1}
  * Siphons: [{X, Y}] with proof of non-emptiability via conservation
      laws
- Implement verification script that:
  * Recomputes    from graph structure
```

```
  * Verifies equilibrium satisfies dX/dt = dY/dt = 0
  * Checks siphon non-emptiability

SUCCESS CRITERIA:
- **MVR (Months 2-4)**: Deficiency computed for Brusselator and 3 other networks (Schl gl,
  formose, Lotka-Volterra), FHJ predictions correct
- **Strong (Months 5-7)**: All positive equilibria found symbolically for 1 networks,
  persistence certified for 5+ networks
- **Publication (Months 8-9)**: General algorithm for RAF detection in metabolic networks,
  database of 20+ analyzed CRNs with certificates, comparison to stochastic simulation
  (Gillespie) showing agreement

VERIFICATION PROTOCOL:
1. Deficiency:  _manual  = 8 - 2 - 5 = 1
2. Equilibrium: Substitute (X*, Y*) into ODEs, verify RHS = 0 with mpmath (100 digits)
3. Autocatalysis: Cycle [R2] has stoichiometry +X, confirming autocatalysis
4. Persistence: Conservation laws A+C+X=const with  A  >0 prevents X 0
5. Literature: Brusselator known to exhibit Hopf bifurcation; check parameter conditions

EXPORT:
- `brusselator_certificate.json`: Complete certificate (deficiency, equilibria, siphons, RAF)
- `crn_analyzer.py`: Python module with all CRN analysis functions
- `verification.py`: Independent checker for certificates

This is a PURE THOUGHT challenge: use ONLY symbolic math, graph algorithms, and LP.
NO numerical ODE integration until final validation phase.
```

---

## 0.5 5. Success Criteria

### 0.5.1 Minimum Viable Result (MVR) — Months 2-4

**Deliverable**: Working CRN analyzer with deficiency computation and FHJ predictions.
**Specific Metrics**:

- **Deficiency Calculation**:
- Correctly computes  for 5 networks (Brusselator, Schlögl, formose, Lotka-Volterra, MAPK cascade)
- Identifies linkage classes using graph algorithms (NetworkX)
- Stoichiometry matrix rank computed via exact linear algebra
- **FHJ Theorem Application**:

- Predicts equilibrium behavior for =0 weakly reversible networks (unique equilibrium)
- Identifies networks with multistationarity potential ( 1)
- **Conservation Laws**:
- Computes $\ker(S^T) using SymPy nullspace(exact rational arithmetic)$

- Validates conservation via ODE integration (total mass conserved to $10^{-6}$)

**Certificate**: JSON export with deficiency, linkage classes, stoichiometry matrix (exact integers).

---

#### 0.5.2 Strong Result — Months 5-7

**Deliverable**: Complete steady-state and persistence analysis with symbolic methods.
**Specific Metrics**:

- **Equilibria**:
- All positive equilibria found for 3+ networks using Gröbner bases
- Symbolic solutions (exact rational/algebraic) with residuals $< 10^{-50}$

- Multistationarity detected in Schlögl (expect 2 equilibria for suitable parameters)
- **Autocatalysis**:
- Autocatalytic cycles identified in formose reaction, Brusselator
- RAF set detection algorithm implemented, tested on metabolism-like networks
- Comparison to literature RAF sets (Hordijk 2004 examples)
- **Persistence**:
- Siphon enumeration for networks with 10 species
- Persistence certified via conservation laws for 5+ networks
- Emptiable siphons identified in non-persistent networks

**Certificate**: Symbolic equilibria, autocatalytic cycle list, siphon analysis with conservation law proofs.

---

### 0.5.3 Publication-Quality Result — Months 8-9

**Deliverable**: Novel results, comprehensive database, formal verification.
**Specific Metrics**:

- **Novel Contribution**:
- New algorithm for RAF detection in large metabolic networks (100+ reactions)
- Characterization of deficiency-one networks admitting unique vs multiple equilibria
- Computational complexity analysis (deficiency in $O(n^3)$, siphons exponential)
- **Database**:
- 20+ analyzed CRNs with complete certificates (JSON export)
- Networks from biochemistry (glycolysis, TCA cycle), origin of life (formose, HCN polymerization), synthetic biology
- Each certificate includes deficiency, equilibria, autocatalysis, persistence
- **Validation**:
- Stochastic simulation (Gillespie algorithm) matches deterministic equilibria
- Comparison to experimental bistability data (e.g., cell cycle networks)
- Formal verification: translate persistence proofs to Lean/Isabelle (for 1-2 small networks)
- **Publication Targets**:
- *SIAM Journal on Applied Dynamical Systems* (CRN theory)
- *Journal of Mathematical Biology* (autocatalysis and origin of life)
- *Bulletin of Mathematical Biology* (computational methods)

---

## 0.6 6. Verification Protocol

### 0.6.1 Automated Checks (Run After Each Phase)

```
def verify_crn_analysis(crn: ChemicalReactionNetwork, certificate:
    Dict) -> bool:
    """
    Comprehensive verification of CRN certificate.
    """
    print("=== CRN Certificate Verification ===\n")

    # 1. Deficiency Verification
    print("1. Verifying Deficiency Calculation")
    n_claimed = certificate['deficiency_analysis']['num_complexes']
      _claimed   =
       certificate['deficiency_analysis']['num_linkage_classes']
```

```
    s_claimed = certificate['deficiency_analysis']['stoichiometric_subspace_dim']
    _claimed = certificate['deficiency_analysis']['deficiency']

    n_actual = len(crn.complexes)
    _actual = len(crn.linkage_classes())
    s_actual = stoichiometric_subspace_dimension(crn)
    _actual = n_actual - _actual - s_actual

    assert n_claimed == n_actual, f"Complex count mismatch"
    assert _claimed == _actual, f"Deficiency mismatch: { _claimed } { _actual }"
    print(f"  Deficiency = { _claimed } verified (n={n_actual}, ={ _actual }, s={s_actual})")

    # 2. Stoichiometry Matrix
    print("\n2. Verifying Stoichiometry Matrix")
    S_claimed = np.array(certificate['stoichiometry_matrix'])
    S_actual = crn.stoichiometry_matrix()
    assert np.allclose(S_claimed, S_actual), "Stoichiometry mismatch"
    print(f"  Stoichiometry matrix matches ({S_actual.shape})")

    # 3. Equilibria
    print("\n3. Verifying Equilibria")
    for i, eq_cert in enumerate(certificate['equilibria']):
        residual = eq_cert['residual']
        assert residual < 1e-8, f"Equilibrium {i} residual too large: {residual}"
        print(f"  Equilibrium {i}: residual = {residual:.2e}")

    # 4. Conservation Laws
    print("\n4. Verifying Conservation Laws")
    C = conservation_laws(crn)
    num_laws_claimed = len(certificate['conservation_laws'])
    assert len(C) == num_laws_claimed, f"Conservation law count mismatch"

    # Check each law satisfies C^T S = 0
    S = crn.stoichiometry_matrix()
    for i, c in enumerate(C):
        c_numeric = np.array([float(Fraction(val)) for val in c])
        product = c_numeric @ S
        assert np.allclose(product, 0, atol=1e-10), f"Conservation law {i} invalid"
    print(f"  All {len(C)} conservation laws verified (C^T S = 0)")

    # 5. Persistence
    print("\n5. Verifying Persistence")
    if certificate['persistence']['persistent']:
        emptiable = certificate['persistence']['emptiable_siphons']
        assert len(emptiable) == 0, "Persistent network has emptiable siphons"
```

```
        print("       Persistence certified (no emptiable siphons)")
    else:
        print("   ! Network not persistent (emptiable siphons exist)")

    print("\n=== ALL VERIFICATIONS PASSED ===")
    return True


# Literature Comparison
def compare_to_literature(crn_name: str, certificate: Dict):
    """
    Compare results to known literature values.
    """
    literature_values = {
        'brusselator': {'deficiency': 1, 'equilibria_count': 1, 'persistent': True},
        'schloegl': {'deficiency': 2, 'equilibria_count': 2, 'persistent': True},
        'lotka_volterra': {'deficiency': 0, 'equilibria_count': 1, 'persistent': False}
    }

    if crn_name.lower() in literature_values:
        lit = literature_values[crn_name.lower()]
        cert_  = certificate['deficiency_analysis']['deficiency']
        cert_eq_count = len(certificate['equilibria'])
        cert_persistent = certificate['persistence']['persistent']

        print(f"\n=== Literature Comparison: {crn_name} ===")
        print(f"Deficiency: {cert_ } (literature: {lit['deficiency']}) " +
              ("   " if cert_  == lit['deficiency'] else "   "))
        print(f"Equilibria: {cert_eq_count} (literature: {lit['equilibria_count']}) " +
              ("   " if cert_eq_count == lit['equilibria_count']
                  else "   "))
        print(f"Persistent: {cert_persistent} (literature: {lit['persistent']}) " +
              ("   " if cert_persistent == lit['persistent'] else
                  "   "))
```

**Manual Checks**:

- Plot phase portraits for 2D systems (Lotka-Volterra, Brusselator) — verify equilibria and cycles
- Compare Gröbner basis solutions to numerical root-finding (mpmath) for validation
- Literature cross-check: Feinberg (1987) examples, Angeli et al. (2007) persistence results

## 0.7 7. Resources and Milestones

### 0.7.1 Essential References

**Deficiency Theory**:

- Feinberg, M. (1987). "Chemical reaction network structure and the stability of complex isothermal reactors—I. The deficiency zero and deficiency one theorems." *Chemical Engineering Science* 42(10): 2229-2268.
- Horn, F., Jackson, R. (1972). "General mass action kinetics." *Archive for Rational Mechanics and Analysis* 47(2): 81-116.

**Autocatalysis and RAF**:

- Hordijk, W., Steel, M. (2004). "Detecting autocatalytic, self-sustaining sets in chemical reaction systems." *Journal of Theoretical Biology* 227(4): 451-461.
- Kauffman, S. A. (1986). "Autocatalytic sets of proteins." *Journal of Theoretical Biology* 119(1): 1-24.

**Persistence**:

- Angeli, D., De Leenheer, P., Sontag, E. D. (2007). "A Petri net approach to the study of persistence in chemical reaction networks." *Mathematical Biosciences* 210(2): 598-618.

**Multistationarity**:

- Craciun, G., Feinberg, M. (2005). "Multiple equilibria in complex chemical reaction networks: I. The injectivity property." *SIAM Journal on Applied Mathematics* 65(5): 1526-1546.

**Software**:

- NetworkX (graph algorithms), SymPy (Gröbner bases), mpmath (high-precision arithmetic)

### 0.7.2 Milestone Checklist

**Month 1-2**:

Implement Complex, Reaction, ChemicalReactionNetwork classes

Stoichiometry matrix construction validated on 3 examples

Reaction graph built with NetworkX, linkage classes computed

Deficiency calculated for Brusselator, Schlögl, Lotka-Volterra

**Month 3-4**:

Conservation laws computed via $\ker(\mathrm{S}^T) for 5 networks$

FHJ theorem applied, predictions match literature

Autocatalytic cycles identified in formose, Brusselator

RAF detection algorithm implemented

**Month 5-6**:

Gröbner basis solver finds equilibria for  1 networks

Symbolic solutions verified with residuals $< 10^{-50}$

Multistationarity detected in Schlögl (2 equilibria)

Siphon enumeration for networks with  8 species

**Month 7-9**:

Persistence certified for 5+ networks via conservation laws

Database of 20 CRN certificates (JSON) exported

Verification script passes all checks

Comparison to Gillespie stochastic simulation

Draft paper on RAF detection algorithm

### 0.7.3 Common Pitfalls

- **Gröbner Basis Complexity**: For large systems (>5 variables), Gröbner bases may not terminate in reasonable time; use numerical methods for initial guesses, then refine symbolically
- **Siphon Enumeration**: Exponential in number of species; for >10 species, use heuristics or focus on minimal siphons
- **Non-positive Equilibria**: Symbolic solvers may return negative or complex solutions; always filter for $x_i > 0 and real values$

- **Weak Reversibility**: Not all networks are weakly reversible; FHJ theorem only applies to specific cases; check carefully

---

**End of PRD 19**