EpidCRN Package Documentation

Team

November 17, 2025

Contents

1 Package Overview

EpidCRN is a Mathematica package for epidemiological models (ME) (whose definition assumes the existence of a unique disease-free boundary fixed point =DFE), which aims to use also Chemical Reaction Network Theory (CRNT) methods. Models are entered as a pair formed by the reactions ”RN”, (which do not depend on the form of the rates), and their rates ”rts”. The first could be viewed as a definition/standardization of the model, and the second as a secondary feature, which may be chosen as polynomial (mass-action), or fractional, at the convenience of the author, and depending on the availability of data.

The main object of study are models with possibly several boundary fixed points besides the DFE (or minimal siphons, in CRN terminology), which we call multi-strain. The focus of our package is somewhat different than that of other similar packages, a major concern being the flexibility of operating symbolically whenever this is possible (this is a major concern in ME).Once the model is input, anything should be achievable, at the usual cost. For example, rational computations of R_0 are achieved instantaneously (for multi-strains, this is the max of individual R_i for each strain), but small non-rational R_0 may require further effort. Also, invasion numbers, which appear in stability conditions for the non DFE boundary fixed points, are instantaneous if rational, but may require work from the user, if the ”siphon” boundary fixed points are not rational. Numeric stability scans are also available, in cases where stability is not symbolic, but the user needs to adjust various tolerances, to make this work. In summary, the package aims to provide trivial computations and also less trivial ones, but the latter are at the cost of extra user’s time.

2 Package Structure

2.1 Modular Organization

The package is split into subpackages:

1. `EpidCRN.wl` - Main loader package, which contains all the usage statements. The following subpackages are all in the same directory with the loader.
2. `Core.wl` - Basic network analysis (`EpidCRN`Core``)
3. `CRNT.wl` - Chemical reaction network theory (`EpidCRN`CRNT``)
4. `Boundary.wl` - NGM and boundary analysis (`EpidCRN`Boundary``)
5. `Bifurcation.wl` - Hopf bifurcations and parameter scanning (`EpidCRN`Bifurcation``)
6. `Siphons.wl` - Siphon and persistence analysis (`EpidCRN`Siphons``)
7. `Utils.wl` - Utility functions (`EpidCRN`Utils``)
8. `Visualization.wl` - (`EpidCRN`Visualization``)

2.2 Dependency Chain

`Core` $\rightarrow$ `CRNT` $\rightarrow$ `Boundary` $\rightarrow$ `Bifurcation` $\rightarrow$ `Siphons`

3 Fundamental Functions Reference

3.1 Core Functions (EpidCRN‘Core‘)

3.1.1 extMat[reactions]

Master function extracting network structure.
Returns: {species, α, β, γ, R_v, RHS, deficiency}

1. **species**: List of species names as strings
2. α: Reactant stoichiometric matrix
3. β: Product stoichiometric matrix
4. γ: Net stoichiometric matrix ($\beta - \alpha$)
5. R_v: Rate vector template
6. **RHS**: Right-hand side of ODE system
7. **deficiency**: Network deficiency information

3.1.2 asoRea[RN]

Converts reaction network to association format.
Input: `RN = {"s"+"i"->2"i", "i"->"r", "r"->"s"}`
Returns: Association with ”Substrates”/”Products” keys

3.1.3 compToAsso[side]

Parses reaction side to association of species $\rightarrow$ coefficients.

3.1.4 extSpe[reactions]

Extracts species list from reaction network.

3.2 Siphon Analysis (EpidCRN‘Siphons‘)

3.2.1 minSiph[species, asoReactions]

CRITICAL FUNCTION: Computes minimal siphons (sets of species that become zero at DFE).
Input:

1. `species`: List of species names as strings
2. `asoReactions`: Output of `asoRea[RN]`

Returns: List of lists of species names as strings
Example: `{{"x2"}, {"B1","S1"}, {"B2","S2"}}`

```
{spe, al, be, gam, Rv, RHS, def} = extMat[RN];
mS = minSiph[spe, asoRea[RN]];
(* mS contains variable names, NOT indices *)
```

Critical Note: `minSiph` returns variable names as strings. Do NOT convert to indices in downstream functions.

3.3 CRNT Functions (EpidCRN‘CRNT‘)

3.3.1 getComE[RN_List]

Extract complexes and edges from reaction network.

```
getComE[RN_List] := Module[{complexes, edges},
  complexes = {};
  edges = {};
  Do[
    Module[{left, right},
      left = RN[[i, 1]];
      right = RN[[i, 2]];
      If[! MemberQ[complexes, left], AppendTo[complexes, left]];
      If[! MemberQ[complexes, right], AppendTo[complexes, right]];
      AppendTo[edges, {left, right}];
    ],
    {i, Length[RN]}
  ];
  {complexes, edges}
];
```

Input: Reaction network as list of {left, right} pairs
Returns: `{complexes, edges}`
Example: `getComE[{{a,b},{b,c},{c,a}}]` → `{{a, b, c}, {{a, b}, {b, c}, {c, a}}}`

3.3.2 IaFHJ[vert_, edg_]

Incidence matrix analysis for FHJ graphs.

```
IaFHJ[vert_, edg_] := Module[{gg, oU, taF},
  gg[a_, b_] := Which[
    a === b[[1]], -1,
    a === b[[2]], 1,
    True, 0
  ];
  oU = Table[
    gg[vert[[i]], edg[[j]]],
    {i, Length[vert]},
    {j, Length[edg]}
  ];
  taF = TableForm[
    oU,
    TableHeadings -> {vert, edg},
    TableAlignments -> {Right, Top}
  ];
  {oU, taF}
];
```

Input: Vertices and edges lists
Returns: `{matrix (n_complexes × n_reactions), tableForm}`

3.3.3 IkFHJ[vert_, edg_, tk_]

Ik matrix computation for FHJ analysis.

```
IkFHJ[vert_, edg_, tk_] := Module[{tri, gg, oU},
  tri = MapThread[Append, {edg, tk}];
  gg[a_, b_] := Which[
    a === b[[1]], b[[3]],
    a === b[[2]], 0,
    True, 0
  ];
  oU = Table[
    gg[vert[[i]], tri[[j]]],
    {i, Length[vert]},
    {j, Length[tri]}
  ] // Transpose
];
```

Input: Vertices, edges, and rate constants
Returns: Matrix (n_reactions × n_complexes)

3.3.4 SpeComInc[spec_, comp_]

Species-complex incidence matrix.

```
SpeComInc[spec_, comp_] := Coefficient[#, spec] & /@ comp;
```

Input: Species list and complexes list
Returns: Coefficient matrix
Example: `SpeComInc[{x, y}, {x + y, 2 x, y}]` → `{{1, 1}, {2, 0}, {0, 1}}`

3.3.5 lapK[RN_, rates_]

Main Laplacian computation function.

```
lapK[RN_, rates_] := Module[{complexes, edges, laplacian},
  {complexes, edges} = getComE[RN];
  laplacian = IkFHJ[complexes, edges, rates];
  laplacian
];
```

Input: Reaction network and rate constants
Returns: Laplacian matrix (n_complexes × n_complexes)

4 Boundary Analysis Functions

4.1 NGM[mod, inf]

Next Generation Matrix analysis.
Input:

1. `mod`: `{RHS, var, par}`
2. `inf`: List of infected compartment indices

Returns: `{Jx, Jy, ..., K, ...}` where `K = ngm[[4]]` is the transmission matrix

4.2 Current Session Work: Enhanced Boundary Analysis Modules

4.2.1 bdAn[RN, rts] - Simplified Analysis Module

Core analysis without boundary fixed point computation.

```
bdAn[RN_, rts_] := Module[{
    spe, al, be, gam, Rv, RHS, def, var, par, cp, cv, ct,
    mSi, mSiIndices, inf, mod, K, eig, R0A, cDFE, RDFE, eq0, var0, E0,
    Jx, Jy, eigenSystem, eigenvals, eigenvecs, nonzeroIndices,
    relevantEigenvals, strainAssociation, sortedPairs, mSiNGM,
    ngm
   },

   {spe, al, be, gam, Rv, RHS, def} = extMat[RN];
   var = ToExpression[spe];
   RHS = gam . rts;
   par = Par[RHS, var];
   cp = Thread[par > 0];
   cv = Thread[var >= 0];
   ct = Join[cp, cv];

   (* Direct assignment - mSi contains variable names as strings *)
   mSi = minSiph[spe, asoRea[RN]];
   mSiIndices = Map[Flatten[Position[spe, #] & /@ #] &, mSi]; (* Indices for NGM *)
   inf = Union[Flatten[mSiIndices]];

   (* Compute DFE *)
   cDFE = Flatten[Thread[ToExpression[#] -> 0] & /@ mSi];
   RDFE = RHS /. cDFE;
   eq0 = Thread[RDFE == 0];
   var0 = Complement[var, var[[inf]]];
   E0 = Join[Solve[eq0, var0] // Flatten, Thread[var[[inf]] -> 0]];

   (* Compute NGM *)
   mod = {RHS, var, par};
   ngm = NGM[mod, inf];
   Jx = ngm[[1]] // FullSimplify;
   Jy = ngm[[5]] // FullSimplify;
   K = ngm[[4]] // FullSimplify;

   (* Get eigenvalues and organize by strain *)
   eigenSystem = Eigensystem[K];
   eigenvals = eigenSystem[[1]];
   eigenvecs = eigenSystem[[2]];
   nonzeroIndices = {};
   Do[If[eigenvals[[i]] =!= 0, AppendTo[nonzeroIndices, i]],
      {i, Length[eigenvals]}];

   If[Length[nonzeroIndices] > 0,
    relevantEigenvals = eigenvals[[nonzeroIndices]];
    mSiNGM = Table[Flatten[Table[Position[inf, mSiIndices[[i]][[j]]][[1,1]],
      {j, Length[mSiIndices[[i]]]}]], {i, Length[mSiIndices]}];
    strainAssociation = Table[Module[{strain1Nonzeros, strain2Nonzeros, evec},
       evec = eigenvecs[[nonzeroIndices[[i]]]];
       strain1Nonzeros = Count[evec[[mSiNGM[[1]]]], Except[0]];
       strain2Nonzeros = Count[evec[[mSiNGM[[2]]]], Except[0]];
       If[strain1Nonzeros > strain2Nonzeros, 1,
        If[strain2Nonzeros > strain1Nonzeros, 2, i]]],
       {i, Length[relevantEigenvals]}];
    sortedPairs = Sort[Transpose[{strainAssociation, relevantEigenvals}]];
    R0A = sortedPairs[[All, 2]];,
    R0A = {};];

   {RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A}
]
```

Returns: {`RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A`}

5 bdFp Module Usage Guide

5.1 Purpose

`bdFp` computes boundary fixed points on siphon facets for epidemic models, separating rational solutions from algebraic ones and providing scalar elimination equations for non-rational cases.

5.2 Syntax

```
bdfpT = bdFp[RHS, var, mSi]
```

5.3 Parameters

5.3.1 Required Inputs

1. **RHS**: Right-hand side vector of the ODE system (from `bdAn`)
2. **var**: List of all variables as symbols (from `bdAn`)
3. **mSi**: Minimal siphons as variable names (strings) (from `bdAn`)

5.4 Output Format

Returns a list of pairs, one for each siphon facet:

```
{
  {rationalSols1, scalarEq1},
  {rationalSols2, scalarEq2},
  ...
}
```

5.4.1 Output Components

1. **rationalSols**: List of rational solutions (rules like {`x -> value, y -> value`})
2. **scalarEq**:
   (a) `None` if all solutions are rational
   (b) Factored polynomial equation if non-rational solutions exist
   (c) `"froze"` if computation timed out

5.5 Example Usage

5.5.1 Basic Usage

```
(* Get core analysis first *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A} = bdAn[RN, rts];

(* Compute boundary fixed points *)
bdfpT = bdFp[RHS, var, mSi];

(* Extract results for each facet *)
facet1Results = bdfpT[[1]];  (* {rationalSols, scalarEq} for first siphon *)
facet2Results = bdfpT[[2]];  (* {rationalSols, scalarEq} for second siphon *)
```

5.5.2 Accessing Results

```
(* Get rational solutions for facet 1 *)
rationalSols1 = bdfpT[[1]][[1]];

(* Get scalar equation for facet 1 (if any) *)
scalarEq1 = bdfpT[[1]][[2]];

(* Check if facet has only rational solutions *)
If[bdfpT[[2]][[2]] === None,
  Print["Facet␣2␣has␣only␣rational␣solutions"],
  Print["Facet␣2␣has␣algebraic␣solutions␣satisfying:␣", bdfpT[[2]][[2]]]
];
```

5.6 Example Output

5.6.1 Console Output

```
bdFp finds the fixed points on siphon facets are
fps on siphon facet {i1} are {{i_2->0, s->\Lambda/(\mu+\rho), v_1->(\Lambda\rho)/(\mu(\mu+
and 2 more fps satisfying the scalar equation i_2^2(\beta_v\gamma_2 + \mu)(\beta_2 i_2 - \
fps on siphon facet {i2} are {{rational solution 1}, {rational solution 2}}
```

5.6.2 Return Value

```
{
  {
    {{i_2->0, s->\Lambda/(\mu+\rho), v_1->(\Lambda\rho)/(\mu(\mu+\rho))}},
    i_2^2(\beta_v\gamma_2 + \mu)(\beta_2 i_2 - \Lambda + \mu) == 0
  },
  {
    {{solution1}, {solution2}},
    None
  }
}
```

5.7 Interpretation Guide

5.7.1 Rational Solutions

1. Direct algebraic expressions in terms of parameters
2. Can be used immediately for analysis
3. Represent equilibria with explicit formulas

5.7.2 Scalar Equations

1. Factored polynomial constraints for non-rational solutions
2. Polynomial degree indicates number of solutions
3. Factors reveal mathematical structure
4. Solutions involve radicals, roots, or complex expressions

5.7.3 Special Cases

1. **”froze”**: Computation exceeded 10-second timeout
2. **None**: All solutions are rational (no scalar equation needed)
3. **Empty rational list {}**: All solutions are non-rational

5.8 Common Workflow

```
(* 1. Load model and get core analysis *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A} = bdAn[RN, rts];

(* 2. Compute boundary fixed points *)
bdfpT = bdFp[RHS, var, mSi];

(* 3. Extract and analyze results *)
Do[
  {rationalSols, scalarEq} = bdfpT[[i]];
  Print["Siphon␣", i, ":"];
  Print["␣␣Rational␣solutions:␣", Length[rationalSols]];
  If[scalarEq =!= None,
    Print["␣␣Non-rational␣solutions␣satisfy:␣", scalarEq];
  ];
, {i, Length[bdfpT]}];

(* 4. Use rational solutions for further analysis *)
rationalEquilibria = Flatten[bdfpT[[All, 1]], 1];
```

5.9 Error Handling

5.9.1 Timeout Protection

1. 10-second timeout for main `Solve` operation
2. 3-second timeout for elimination step
3. Returns `"froze"` for timed-out computations

5.9.2 Common Issues

1. **Empty results**: Check that `mSi` contains valid siphon names
2. **All ”froze”**: System may be too complex; try simpler parameter values
3. **No scalar equation**: Normal if all solutions are rational

5.10 Integration with bdAn/bd2

5.10.1 With bdAn (Core Analysis)

```
coreResults = bdAn[RN, rts];
{RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A} = coreResults;
bdfpT = bdFp[RHS, var, mSi];
```

5.10.2 With bd2 (Complete Analysis)

```
fullResults = bd2[RN, rts];
{RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A, EA, bdfpT} = fullResults;
(* bdfpT already computed by bd2 *)
```

5.11 Performance Notes

1. Runtime scales with system complexity and siphon count
2. Elimination step may be slow for high-dimensional systems
3. Factorization improves readability but adds computational cost
4. Consider using `bdAn` alone for preliminary analysis if speed is critical

5.11.1 bd2[RN, rts] - Complete Analysis Module

Complete analysis combining bdAn + bdFp.

```
bd2[RN_, rts_] := Module[{
    coreResults, RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A,
    bdfpT, EA
   },

   (* Get core analysis from bdAn *)
   coreResults = bdAn[RN, rts];
   {RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A} = coreResults;

   (* Compute boundary fixed points using bdFp with timeout protection *)
   bdfpT = bdFp[RHS, var, mSi];

   (* Create EA structure for backward compatibility *)
   EA = Table[
     Module[{siphonVars, remainingVars},
       siphonVars = ToExpression[mSi[[j]]];
       remainingVars = Complement[var, siphonVars];
       {Thread[RHS /. Thread[siphonVars -> 0] == 0], remainingVars}
     ], {j, Length[mSi]}];

   (* Return complete results *)
   {RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A, EA, bdfpT}
]
```

Returns: {`RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A, EA, bdfpT`}

6 Invasion Graph Theory Implementation

6.1 Theoretical Foundation

Invasion graphs provide a framework for understanding global dynamics of multi-strain epidemic systems by mapping which strains can invade established equilibria. This connects the Lotka-Volterra invasion graph theory of Almaraz et al. with epidemiological invasion numbers.

6.2 Key Definitions

6.2.1 Admissible Communities

For epidemic models, an admissible community corresponds to a biologically meaningful endemic state where certain strains are present.

6.2.2 Invasion Rates

For community I with equilibrium $\mathbf{u}^*$, the invasion rate of strain $j \notin I$ is:

$$r_j(I) = b_j + \sum_{k \in I} a_{jk} u_k^*$$

6.2.3 Invasion Numbers (Epidemic Context)

For Rahman-type models:

$$R_j^{(i)} = \mathcal{R}_j \times s_i$$

where $\mathcal{R}_j = \beta_j / \nu_j$ and s_i is susceptible level at equilibrium i.

6.3 Equivalence Theorem

Key Result: LV invasion rates and epidemic invasion numbers are equivalent:

$$r_j^{\text{LV}}(I) = \nu_j (R_j^{(i)} - 1)$$

Sign Correspondence:

$$r_j^{\text{LV}}(I) > 0 \Leftrightarrow R_j^{(i)} > 1 \quad \text{(successful invasion)} \tag{1}$$
$$r_j^{\text{LV}}(I) < 0 \Leftrightarrow R_j^{(i)} < 1 \quad \text{(failed invasion)} \tag{2}$$

6.4 Implementation Algorithm

```
invasionGraph[RHS_, var_, par_, mSi_] := Module[{
    communities, invasionRates, edges
   },

   (* 1. Identify all admissible communities *)
   communities = findAdmissibleCommunities[RHS, var, mSi];

   (* 2. For each community, compute invasion rates *)
   invasionRates = Table[
     computeInvasionRates[RHS, var, communities[[i]]],
     {i, Length[communities]}];

   (* 3. Construct edges based on invasion success *)
   edges = {};
   Do[
     invasiveSets = Position[invasionRates[[i]], _?Positive];
     Do[AppendTo[edges, communities[[i]] -> communities[[j]]],
        {j, invasiveSets}];,
     {i, Length[communities]}];

   {communities, invasionRates, edges}
]
```

6.5 Rahman Model Implementation

```
rahmanInvasionGraph[E1t_, E2t_, R0A_, E0_] := Module[{
    R01, R02, R12, R21, S1, S2, invasionMatrix
   },

   (* Extract basic reproduction numbers *)
   {R01, R02} = R0A /. E0;

   (* Extract susceptible levels at single-strain equilibria *)
   S1 = (* Extract S level from E1t *);
   S2 = (* Extract S level from E2t *);

   (* Compute invasion numbers *)
   R12 = R0A[[1]] /. E2t; (* Strain 1 invading strain 2 *)
   R21 = R0A[[2]] /. E1t; (* Strain 2 invading strain 1 *)

   (* Construct invasion graph based on thresholds *)
   invasionMatrix = {
     {"DFE", If[R01 > 1, "E1", ""], If[R02 > 1, "E2", ""]},
     {"E1", "", If[R12 > 1, "Coexistence", ""]},
     {"E2", If[R21 > 1, "Coexistence", ""], ""}
   };

   {R01, R02, R12, R21, invasionMatrix}
]
```

7 Bifurcation Analysis Functions

7.1 fpHopf[RHS, var, par, p0val]

Fixed point finder with Hopf analysis.

Returns: `{posSols, complexEigs, angle, eigs}`

1. **angle**: `ArcTan[Re/Im]*180/Pi` (negative = stable focus, positive = Hopf)

7.2 Parameter Space Scanning Functions

7.2.1 scan Function

The `scan` function represents the most sophisticated parameter scanning approach:

```
{finalPlot, noSolPoints, results} = scan[RHS, var, par, persRule, plotInd,
  mSi, gridRes, steadyTol, stabTol, chopTol, R01, R02, R12, R21]
```

Key Features:

1. Automatic variable detection from minimal siphons (`mSi`)
2. Dual scanning modes: Grid mode (fixed resolution) and range mode (adaptive stepping)
3. Comprehensive equilibrium classification: DFE, E1, E2, EE-Stable, EE-Unstable, NoSol
4. Reproduction number integration with R-curve overlays
5. Robust numerical methods with timeouts and error handling

7.2.2 scanPar Function

Simpler, more direct scanning approach:

```
{finalPlot, errors, results} = scanPar[RHS, var, par, p0val, plotInd,
  gridRes, plot, hTol, delta, wRan, hRan, R01, R02, R21, R12]
```

8 Critical Usage Notes

Critical Variable Handling Rules

1. **mSi contains variable names as strings**, not indices
2. Species names should be strings (”s”, ”i1”, ”i2”)
3. Use `ToExpression[mSi[[j]]]` to convert to symbols when needed
4. NEVER convert `mSi` to indices in `bdAn` or related functions

8.1 Matrix Conventions

1. α: reactant stoichiometric matrix (species $\times$ reactions)
2. β: product stoichiometric matrix
3. $\gamma = \beta - \alpha$: net stoichiometric matrix
4. Laplacian follows CRNT convention (column sums = 0)

8.2 Function Compatibility

1. Functions expect exact expression matching using === operator
2. Most functions return lists, not associations
3. Always use timeout protection for `Solve` operations in boundary analysis
4. Check for rational solutions using `FreeQ[sol, Sqrt | Power[_, Except[_Integer]] | Root | _Complex]`

9 Complete Analysis Workflow

```
(* 1. Load package *)
<< EpidCRN`;

(* 2. Define model *)
RN = {0 -> "s", "s" + "i1" -> 2*"i1", "s" + "i2" -> 2*"i2",
      "s" -> 0, "i1" -> 0, "i2" -> 0};
rts = {La, be1*s*i1/(1+a1*i1), be2*s*i2/(1+a2*i2),
       mu*s, (mu+ga1)*i1, (mu+ga2)*i2};

(* 3. Core analysis (fast, no boundary points) *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A} = bdAn[RN, rts];

(* 4. Complete analysis (with boundary points and timeout protection) *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A, EA, bdfpT} = bd2[RN, rts];
```

```
(* 5. Extract reproduction numbers *)
{R01, R02} = R0A /. E0;

(* 6. Parameter space analysis *)
{plot, errors, results} = scanPar[RHS, var, par, p0val, plotInd,
  30, Automatic, 0.01, 1/20, 0.5, 0.5, R01, R02, R21, R12];

(* 7. Bifurcation analysis *)
{bestAngle, bestValues, finalP0Val} =
  optHopf[RHS, var, par, coP, {3,4}, 120, "NelderMead", 4, 4, 500];

(* 8. Invasion graph construction *)
{communities, invasionRates, edges} = invasionGraph[RHS, var, par, mSi];
```

10 File Structure

```
EpidCRN/
    EpidCRN.wl         (main package loader)
    Core.wl            (EpidCRN`Core` context)
    CRNT.wl            (EpidCRN`CRNT` context)
    Boundary.wl        (EpidCRN`Boundary` context)
    Bifurcation.wl     (EpidCRN`Bifurcation` context)
    Siphons.wl         (EpidCRN`Siphons` context)
    Utils.wl           (EpidCRN`Utils` context)
    Visualization.wl   (EpidCRN`Visualization` context)
```

Critical: File names must match subcontext names exactly for `Get[]` to work automatically.

11 Notes for Future Development

1. This documentation preserves all fundamental functions with complete code to prevent package modification errors
2. The `mSi = mS` assignment in `bdAn` is critical - do NOT convert to indices
3. All timeout protections and rational solution filtering have been thoroughly tested
4. The invasion graph framework provides a unified approach to understanding multi-strain dynamics
5. Function compatibility and variable handling rules must be strictly followed

12 Antigenic Variation Models

12.1 Trypanosoma Model (TrypN.nb)

12.1.1 typAnt[RN, rts] - Trypanosoma Antigenic Variation Analysis

Purpose: Analyzes two-strain Trypanosoma brucei models with antigenic variation through Variant Surface Glycoprotein (VSG) switching.

Model Structure:

- **s**: Susceptible host population
- **i1, i2**: Infected with VSG variant 1 or 2
- **r1, r2**: Recovered from variant 1 or 2

- Includes recruitment, infection, recovery, death, and waning immunity

Usage:

```
<< EpidCRN`;
(* Define reaction network and rates as in TrypN.nb *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A} = typAnt[RN, rts];
```

Returns:

1. **RHS**: Right-hand side of ODE system
2. **var**: Variable list {s, i1, i2, r1, r2}
3. **par**: Parameter list
4. **cp**: Parameter constraints (all positive)
5. **mSi**: Minimal siphons (infection compartments)
6. **Jx, Jy**: Jacobian blocks
7. **E0**: Disease-free equilibrium
8. **K**: Next generation matrix
9. **R0A**: Basic reproduction numbers for each variant

Key Features:

- Strain-specific reproduction numbers R_{01}, R_{02}
- Automatic detection of minimal siphons
- NGM analysis for multi-strain dynamics
- Foundation for invasion graph analysis

12.2 Recker Model (ReckerN.nb)

12.2.1 recAnt[RN, rts] - Recker Antigenic Variation Analysis

Purpose: Analyzes multi-strain pathogen models with antigenic variation following Recker et al. framework for malaria and other pathogens with antigenic diversity.

Model Structure:

- **s**: Susceptible population
- **i1, i2, i3**: Infected with antigenic variant 1, 2, or 3
- **r1, r2, r3**: Recovered with variant-specific immunity
- Cross-immunity between variants (implicit in structure)
- Waning immunity allowing reinfection

Usage:

```
<< EpidCRN`;
(* Define 3-strain reaction network and rates as in ReckerN.nb *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A, EA, bdfpT} = recAnt[RN, rts];
```

Returns: Same as `bd2` output (complete boundary analysis):

1. **RHS**: ODE system right-hand side
2. **var**: Variables {s, i1, i2, i3, r1, r2, r3}
3. **par**: Parameters
4. **cp**: Parameter constraints
5. **mSi**: Minimal siphons (infection states)
6. **Jx, Jy**: Jacobian blocks
7. **E0**: Disease-free equilibrium
8. **ngm**: Full NGM structure
9. **R0A**: Basic reproduction numbers for variants
10. **EA**: Endemic equilibria structure
11. **bdfpT**: Boundary fixed points with rational solutions and scalar equations

Key Features:

- Three antigenic variants with full CRNT analysis
- Boundary fixed points on siphon facets
- Rational and algebraic solution separation
- Supports invasion number computation for multi-strain coexistence
- Compatible with parameter scanning and bifurcation analysis

12.3 Comparison: typAnt vs recAnt

Feature	typAnt	recAnt
Application	Trypanosoma (VSG)	General antigenic variation
Number of strains	2	3 (extensible)
Function call	Direct implementation	Wrapper for bd2
Boundary points	No (uses bdAn logic)	Yes (full bd2 output)
Computation time	Faster	Slower (includes bdFp)
Use case	Quick R0 analysis	Complete equilibrium analysis

12.4 Antigenic Variation Analysis Workflow

```
(* 1. Load package *)
<< EpidCRN`;

(* 2. Define model - use TrypN.nb or ReckerN.nb as template *)
RN = {...};  (* reaction network *)
rts = {...}; (* rate expressions *)

(* 3. For quick analysis: use typAnt *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, K, R0A} = typAnt[RN, rts];
Print["R0␣values:␣", R0A /. E0];

(* 4. For complete analysis: use recAnt *)
{RHS, var, par, cp, mSi, Jx, Jy, E0, ngm, R0A, EA, bdfpT} = recAnt[RN, rts];
Print["Boundary␣equilibria:␣", bdfpT];

(* 5. Invasion numbers for coexistence analysis *)
(* Extract boundary equilibria and compute invasion conditions *)
```

12.5 Implementation Notes

1. Both functions follow EpidCRN naming convention: short abbreviations returning lists
2. `typAnt` = ”Trypanosoma Antigenic variation”
3. `recAnt` = ”Recker Antigenic variation”
4. All outputs are lists (not associations) for consistency
5. Functions are fully documented in respective notebook files
6. Compatible with existing EpidCRN workflow (NGM, scanning, bifurcation)

13 Reaction Network Formats and Converters

The EpidCRN package uses three formats for reaction networks:

13.1 Format 1: Symbolic Format

Used by: Internal computations
Format: Reactions use symbolic variables

```
RN = {0 -> s, i+s -> 2*i, s -> 0, i -> 0}
```

Characteristics:

- Species are Mathematica symbols (s, i, r, etc.)
- Sum of species: i+s (symbolic Plus)
- Suitable for direct mathematical operations
- Can be converted using symbToStr

13.2 Format 2: String-Plus Format

Used by: bdAn, NGM, most package functions
Format: String species with Plus for sums

```
RN = {"0" -> "s", "i" + "s" -> "2i", "s" -> "0"}
```

Characteristics:

- Species are strings (”s”, ”i”, ”r”, etc.)
- Sum of species: ”i” + ”s” (Plus of strings)
- Required format for bdAn, NGM, and analysis functions
- Standard format throughout most of the package

13.3 Format 3: Concatenated String Format

Used by: Display/printing only (not functional)
Format: Single concatenated strings

```
RN = {0 -> s, s+i -> 2i, s -> 0}
```

Characteristics:

- Species sums as single strings: "s+i" (not "s"+"i")
- Not accepted by bdAn or other package functions
- Only for display purposes
- Must be converted to Format 2 for use

13.4 Format Converters

13.4.1 symbToStr - Convert Symbolic to String Format

Purpose: Converts symbolic RN (from ODE2RN) to string format for use with bdAn, NGM
Usage:

```
(* Get symbolic RN from ODE2RN *)
{RN, rts, spe, alp, bet, gam} = ODE2RN[RHS, var];

(* Convert to string format for bdAn/NGM *)
RNstr = symbToStr /@ RN;

(* Now can use with bdAn *)
{RHSback, speBack, Rv} = bdAn[RNstr, rts];
```

13.4.2 strToSymb - Convert String to Symbolic Format

Purpose: Converts string-based RN to symbolic format
Usage:

```
(* String format RN *)
RNstr = {"0" -> "s", "s" -> "0"};

(* Convert to symbolic *)
RNsym = strToSymb[RNstr];
```

13.5 Workflow: ODE2RN to NGM

To use ODE2RN output with NGM and other package functions:

```
(* Step 1: Convert ODE to RN *)
RHS = {La - be*i*s - mu*s, be*i*s - (ga + mu)*i};
var = {s, i};
{RN, rts, spe, alp, bet, gam} = ODE2RN[RHS, var];
(* RN is already in string format, ready to use *)

(* Step 2: Use with bdAn to get model structure *)
{RHSback, speBack, Rv} = bdAn[RN, rts];
mod = {RHSback, var, parameters};

(* Step 3: Use with NGM *)
```

```
infVars = {2};  (* Index of infection variable *)
ngmResult = NGM[mod, infVars];
```

Note: ODE2RN now outputs RN in string format directly, so no conversion is needed.

14 ODE to Reaction Network Conversion

14.1 ODE2RN - Convert ODE System to Reaction Network

Purpose: Converts an ODE system to a chemical reaction network representation by identifying monomials and reconstructing the stoichiometric structure.

14.1.1 Algorithm

Given an ODE system $\frac{dx_i}{dt} = f_i(x)$, the algorithm uses helper functions `allT` (extract all terms) and `isN` (detect negative sign):

1. **Extract all terms:** Use `allT[RHS[[i]]]` to expand each equation and extract all terms (monomials with coefficients)
2. **Separate by sign:** Use `isN[term]` to classify terms:
   - Sources: positive terms (where `isN[term]=False`)
   - Sinks: negative terms (where `isN[term]=True`)
3. **Build rates (rts):** Join sources and negated sinks, remove duplicates: `rts = DeleteDuplicates[Join[sources, -sinks]]`
4. **Build gam (γ):** For each equation and each rate, check if rate appears:
   - If rate appears as negative term: $\gamma_{ij} = -1$
   - If rate appears as positive term: $\gamma_{ij} = +1$
   - If rate does not appear: $\gamma_{ij} = 0$
5. **Build alp (α):** Reactant matrix from exponents: α_{ij} = Exponent[rts[[j]], var[[i]]]
6. **Build bet (β):** Product matrix: $\beta = \gamma + \alpha$
7. **Build RN:** Construct reaction network from α (left side) and β (right side)

Helper functions:

- `allT[expr]`: Expands polynomial and returns list of all terms
- `isN[term]`: Returns True if term has negative sign (coefficient -1 or negative number)

14.1.2 Mathematical Foundation

For an ODE system:

$$\frac{dx}{dt} = \sum_j \gamma_j \cdot r_j(x)$$

where γ_j is the j-th column of the stoichiometric matrix and r_j is the j-th reaction rate.
Sign convention:

- Positive-only terms $\rightarrow$ production ($0 \rightarrow$ species)
- Negative-containing terms $\rightarrow$ consumption (reactants)
- Constants (sources) $\rightarrow$ reactions from empty set

14.1.3 Key Assumption

Positive-only monomials: Monomials appearing only with positive sign are assumed to appear in exactly one reaction. This simplifies rate identification.

14.1.4 Usage

```
(* Input: ODE system as list of expressions *)
RHS = {La - mu*s - be*s*i, be*s*i - (mu+ga)*i};
var = {s, i};

(* Convert to reaction network *)
{RN, rts, spe, alp, bet, gam} = ODE2RN[RHS, var];

(* Output: *)
(* RN: reaction network in standard format *)
(* rts: rate vector *)
(* spe: species list *)
(* alp: alpha matrix (reactants) *)
(* bet: beta matrix (products) *)
(* gam: gamma matrix (net) *)
```

14.1.5 Example

Input ODE:

$$\frac{ds}{dt} = \Lambda - \mu s - \beta si \tag{3}$$

$$\frac{di}{dt} = \beta si - (\mu + \gamma)i \tag{4}$$

Algorithm steps:

1. **Extract terms:** allT gives $\{\Lambda, -\mu s, -\beta si\}$ for eq1, $\{\beta si, -\gamma i, -\mu i\}$ for eq2
2. **Separate by sign:** sources = $\{\Lambda, \beta si\}$, sinks = $\{-\mu s, -\beta si, -\gamma i, -\mu i\}$
3. **Rates:** rts = $\{\Lambda, \beta si, \mu s, \gamma i, \mu i\}$ (after removing duplicates)
4. **Gamma:**
$$\gamma = \begin{pmatrix} 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 \end{pmatrix}$$
5. **Alpha (exponents):**
$$\alpha = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$
6. **Beta = gamma + alpha:**
$$\beta = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{pmatrix}$$

Output RN:

```
RN = {0 -> s, i+s -> 2*i, s -> 0, i -> 0, i -> 0};
rts = {La, be*i*s, mu*s, ga*i, i*mu};
```

14.1.6 Implementation Notes

1. Function returns lists (not associations) following EpidCRN convention
2. Species names are lowercase strings in RN output (via ToLowerCase)
3. Rate vector corresponds to reaction order in RN
4. Stoichiometric matrices follow standard CRNT conventions
5. Verification: $\frac{dx}{dt} = \gamma \cdot rts$ reproduces original ODE
6. Matrix names: alp, bet, gam (3 letters for matrices)
7. Helper functions allT and isN use Expand and detect negative coefficients via pattern matching
8. Gamma matrix uses -1 for consumption (negative terms) and +1 for production (positive terms)
9. Alpha matrix extracts exponents directly using Exponent function

14.1.7 Location

Implemented in `Core.wl` (lines 308-459) with helper functions:

- allT (line 312): Extract all terms from expanded polynomial
- isN (line 318): Detect negative sign in term
- ODE2RN (line 344): Main conversion function

Inverse operation to `extMat`.

15 Development Plan for EpidCRN Package Enhancement

Based on comprehensive analysis of the package architecture and current capabilities, we propose a focused three-phase improvement strategy targeting performance, completeness, and usability.

15.1 Phase 1: Performance Optimization (Core & Boundary subpackages)

Objective: Address computational bottlenecks in symbolic boundary analysis.

Rationale: Current implementation experiences timeouts (10s for Solve, 3s for elimination) and slow performance for high-dimensional systems, as documented in Section 5.11-5.12.

Improvements:

1. **Caching Strategy**: Implement memoization for repeated `minSiph` and `NGM` computations
   - Cache siphon analysis results indexed by reaction network structure
   - Store NGM eigenvalues/eigenvectors for parametric families
2. **Symbolic Simplification Pipeline**: Add preprocessing layer in `bdFp`
   - Early detection of rational vs. algebraic cases before calling Solve
   - Gröbner basis preprocessing for polynomial systems
   - Automatic variable elimination ordering optimization
3. **Numerical-Symbolic Hybrid**: For complex systems exceeding timeout
   - Numerical homotopy continuation as fallback for `"froze"` cases
   - Symbolic-numeric validation workflow

Success Metrics: Reduce average computation time by 40%, eliminate 80% of timeout occurrences.

15.2 Phase 2: Complete Invasion Graph Framework (Siphons & new InvasionGraph subpackage)

Objective: Implement and validate the invasion graph theory outlined in Section 6.

Rationale: Section 6 presents theoretical foundation but lacks complete implementation (`findAdmissibleCommunities` and `computeInvasionRates` are referenced but not defined). This framework unifies multi-strain dynamics analysis.

Improvements:

1. **Core Implementation**:
   - Implement `findAdmissibleCommunities[RHS, var, mSi]` using siphon decomposition
   - Implement `computeInvasionRates[RHS, var, community, bdfpT]` connecting to boundary fixed points
   - Complete `rahmanInvasionGraph` with automatic susceptible level extraction
2. **Visualization Integration**:
   - Create `plotInvasionGraph[communities, edges]` in Visualization subpackage
   - Add R0-annotated directed graphs showing invasion thresholds
3. **Validation Suite**:
   - Test against classical models (2-strain SIS, competing strains, vaccine models)
   - Cross-validate with Lotka-Volterra equivalence theorem (Section 6.3)

Success Metrics: Complete API with 5+ validated examples, integrate with existing `bd2` workflow.

15.3 Phase 3: Enhanced User Experience (Utils & Documentation)

Objective: Improve package accessibility and error diagnostics.

Rationale: Critical usage notes (Section 8) suggest common pitfalls; timeout handling returns cryptic `"froze"` messages.

Improvements:

1. **Input Validation Layer**:
   - Add `validateNetwork[RN, rts]` checking reaction-rate consistency
   - Validate siphon structure before expensive computations
   - Type checking for string vs. symbol usage (addresses critical note in Section 8)
2. **Diagnostic Tools**:
   - `estimateComplexity[RN]` predicting computation time before analysis
   - `explainTimeout[bdfpT]` suggesting model simplifications when timeouts occur
   - Progress indicators for long-running `scan` and `scanPar` operations
3. **Interactive Examples**:
   - Create `Examples/` directory with notebook tutorials
   - Document complete workflows for common model types (SIS, SIR, SEIR variants)
   - Add quickstart guide referencing Section 9 workflow

Success Metrics: Reduce user-reported errors by 50%, achieve 90% successful first-run rate for documented examples.

15.4 Implementation Priority

We recommend sequential execution: Phase 1 $\rightarrow$ Phase 2 $\rightarrow$ Phase 3, as performance improvements enable broader testing in Phase 2, and both inform better user guidance in Phase 3. Estimated timeline: 6-8 weeks total (2.5 weeks per phase).