# EpidCRN Package Complete Documentation

Complete Reference Guide

September 23, 2025

## Contents

# 1 Package Overview

**EpidCRN** is a Mathematica package for epidemiological models (ME), which uses Chemical Reaction Network Theory (CRNT) methods. It aims to analyze models with unique disease-free boundary fixed point (DFE), possibly multi-strain (ie. with other boundary fixed points besides the DFE), and with unique positive fixed point (endemic).

Our object of study are models with possibly several boundary fixed points (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; in particular, a major concern is the flexibility of operating symbolically whenever this is possible (this is a major concern in ME).

# 2 Package Structure

## 2.1 Modular Organization

The package is split into subpackages:

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

## 2.2 Dependency Chain

`Core` → `CRNT` → `Boundary` → `Bifurcation` → `Siphons`

# 3 Fundamental Functions Reference

## 3.1 Core Functions (EpidCRN‘Core‘)

### 3.1.1 extMat[reactions]

Master function extracting network structure.
**Returns:** {species, $\alpha$, $\beta$, $\gamma$, $R_v$, RHS, deficiency}

- **species**: List of species names as strings
- $\alpha$: Reactant stoichiometric matrix
- $\beta$: Product stoichiometric matrix
- $\gamma$: Net stoichiometric matrix ($\beta - \alpha$)
- $R_v$: Rate vector template
- **RHS**: Right-hand side of ODE system
- **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 → 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:**

- `species`: List of species names as strings
- `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:**

- `mod`: {`RHS, var, par`}
- `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

- **RHS**: Right-hand side vector of the ODE system (from bdAn)
- **var**: List of all variables as symbols (from bdAn)
- **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

- **rationalSols**: List of rational solutions (rules like {`x -> value, y -> value`})
- **scalarEq**:
  - `None` if all solutions are rational
  - Factored polynomial equation if non-rational solutions exist
  - `"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 - \L
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

- Direct algebraic expressions in terms of parameters
- Can be used immediately for analysis
- Represent equilibria with explicit formulas

### 5.7.2 Scalar Equations

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

### 5.7.3 Special Cases

- **"froze"**: Computation exceeded 10-second timeout
- **None**: All solutions are rational (no scalar equation needed)
- **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

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

### 5.9.2 Common Issues

- **Empty results**: Check that `mSi` contains valid siphon names
- **All "froze"**: System may be too complex; try simpler parameter values
- **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

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

### 5.11.1 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 *)
```

## 5.12 Performance Notes

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

### 5.12.1 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:

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

**Sign Correspondence:**

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

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

## 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}

- **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:**

- Automatic variable detection from minimal siphons (mSi)
- Dual scanning modes: Grid mode (fixed resolution) and range mode (adaptive stepping)
- Comprehensive equilibrium classification: DFE, E1, E2, EE-Stable, EE-Unstable, NoSol
- Reproduction number integration with R-curve overlays
- 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**

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

### 8.1 Matrix Conventions

- $\alpha$: reactant stoichiometric matrix (species $\times$ reactions)
- $\beta$: product stoichiometric matrix
- $\gamma = \beta - \alpha$: net stoichiometric matrix
- Laplacian follows CRNT convention (column sums = 0)

### 8.2 Function Compatibility

- Functions expect exact expression matching using === operator
- Most functions return lists, not associations
- Always use timeout protection for `Solve` operations in boundary analysis
- 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

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