Enumerate reaction networks composed with heteromultimer and single transformations
November 2014

This particular code is to enumerate all possible reaction networks (with certain dimensions, $m \times n$ where n is speciese number m is reaction number) composed of three types of interactions:

$$A + B \rightarrow C$$
$$\mathrm{D} \rightarrow E + F$$
$$G \rightarrow H$$

Those are heterodimerization, disassociation and single transformation. However, we did not include $I \rightarrow 2J$ and $2K \rightarrow L$ those two type of elementary reaction here, we should implement those in near future.

With three types of elementary reactions above, we could construct a set of reaction networks, then we could use DSR graphs and/or bipartie graph to characteristic those networks whether they are multistationary and has closed competition loop (as well as interchangeable competitors).

The main purpose of this document is to explain the procedures that how to construct and enumerate all possible reaction networks when given fixed reaction number and species number.

If a chemical reaction network has m reactions driven by n chemical species, we could have a stoichiometric matrix $N_{m\times n}$ with dimension

But before to go through such checking, we need to preclude situations that clearly not a complex balanced reaction network, by which mean it obeys the following three constraints:

0. Only allow elementary reactions described above, which is the starting point to construct the matrix; (NB: we don't consider birth-death process like $\varnothing \rightarrow X$ and $Y \rightarrow \varnothing$)

a). list all possibly reaction vectors N_i $(i = 1..n)$ and select m of those into matrix (sequence does not matter)

The total number of reaction vectors for $G \rightarrow H$ is $\binom{n}{2}\cdot 2$ (since we need to consider the sides of two species in a reaction);

The total number of reaction vectors for $A + B \rightarrow C$ and $\mathrm{D} \rightarrow E + F$ are both $\binom{n}{3}\cdot 3$ (same here);

We get $r = \binom{n}{2}\cdot 2 + \binom{n}{3}\cdot 3 + \binom{n}{3}\cdot 3$ number of reaction vectors, we store it in matrix $R_{r\times n}$, then we construct all the stoichiometric matrix by choosing m reaction vectors from $R_{r\times n}$ into $N_{m\times n}$, therefore we have total number of $\binom{r}{m}$ matrices to construct. Each constructed matrix $N_{m\times n}$ will go through balanced checking, mass conservation checking to become a valid stoichiometric matrix to go through further bistability check and competition check.

b). (Optional) We could further reduce the number of matrices when constructing them, in the set of constructed matrices there are huge number of matrices are isomorphic, which means any matrix in the set

with column permutation is another matrix in the set. (I did not prove this, I am thinking because I enumberated all possibility in each reaction vector which means no matter how to permutate the columns (species) in a matrix, after permutation the matrix always falls in the same set). Now the set is closure for column (species) permutation, so does the set of $N_{(m-1)\times n}$, then if we construct the set of $N_{m\times n}$ from $N_{(m-1)\times n}$, we just need to add one in three reaction vectors (because with any column permutation we always get an isomorphic graph, the number of nth reaction vector is 3:
$[1, -1, 0, 0, \ldots, 0]$, $[1, 1, -1, 0, \ldots, 0]$, $[1, -1, -1, 0, \ldots, 0]$, the position of 1 and -1 are not important), in this treatment, we can reduce the number of matrices from $\binom{r}{m}$ to $\binom{r}{m-1}\cdot 3$.

I am not sure about if this step is correct

c). Further, we could reduce the number to $\binom{r}{m-2}\cdot 43$ for $n = 5$, or $\binom{r}{m-2}\cdot 46$ for $n \geq 6$. 43 and 46 are numbers of the unique reaction patterns for two reactions between n species. I listed these reaction vectors manually, and implemented it in the code.

1. Mass conservation;

Based on the stoichiometric matrix $N_{m\times n}$, we can construct a vector of mass values $\boldsymbol{m}$, m_i is the mass value of species S_i. Then we have the equation $N\boldsymbol{m} = \mathbf{0}$, because in each reaction the mass of left (reactants) is equal to mass of right (products). We need to make sure $\boldsymbol{m}$ is strictly positive.

a). firstly we check the rank of N (or linearly dependent), if $Rank(N) < m$, then it is linearly dependent, otherwise reject the matrix.

b). then caculate the nullspace basis of N. Then if the ith element in all basis is 0 or negative then m_i is 0 or negative. (This is not clear, may need some prove. When Maple compute the nullspace basis, it always return basis with 1s in e_j which means in the solution space $\sum x_j e_j$, x_j must be strictly positive.)

Actually a) and b) can be combined: if N is linearly independent, NullSpace will be empty. We could just exclude by examine nullspace basis.

Reference: Gevorgyan, A., Poolman, M. G., & Fell, D. A. (2008). Detection of stoichiometric inconsistencies in biomolecular models. Bioinformatics, 24(19), 2245-2251.

2. Complex banlanced: each species has at least one in flow and one out flow (this is very easy to check);

Exclude all matrices with any species (column) has no reactions involved, or all outflow (negative) or all inflow (positive)

Then we need to:

4. check competition: there are at least one species has two -1 and there is another -1 in each of the according reactions;

a). Get the $N\_$ which only have the negative elements in N. In negative matrix, check the RowSum get indices I of -2 and check the ColumnSum get indices J of $Cs_i \leq -2$, if there are two indices $i, h \in I$ and one index $j \in J$ with which $N_{hj} = N_{ij} = -1$, $i = 1..m$, then there is competition (of course the two competitors should be different, $c_1 \neq c_2$).

5. check loop: take indices of competitors, do the network searching, find the loop from one to another and then from the other to this one.

this is fairly easy to understand, I use breadth-first search.

a). first check if there are any species have more than two outflows (negative) then check if there are indeed two reactions with two species interact with another species. Then find the index of competitors.

b). use breadth-first search to search routes between competitors.

c). Note that to complete the competition loop (from bipartite graph), route from one competitor to the other need to avoid the competition reaction.

Cluster matrix into four categories: bistable with closed competition loop, bistable without closed competition loop, monostable with closed competition loop, monostable without closed competition loop.

All the procedure are implemented in the code. Any suggestions and corrections are more than welcome.

Initializations

```
> restart :
> interface(rtablesize = 400) :
> with(ListTools) :
> with(LinearAlgebra) :
> with(VectorCalculus) :
> with(GraphTheory) :
> with(combinat) :
> with(ArrayTools) :
> _Envsignum0 := 0 :
>
```

Functions for multistationality checking (execute before proceeding)

Step 1. Procedures to create the DSR-graph from the stoichiometric matrix

Find the matrix Z from the stoichiometric matrix A:

```
> findZ := proc(A)
    local Z, n, m, i, j :
      n := Dimension(A)[1] :
      m := Dimension(A)[2] :
      Z := Matrix(n, m) :
   for i from 1 to n by 1 do
     for j from 1 to m by 1 do
        if A[i,j] < 0 then Z[i,j] := z[i,j]; end if;      ### what is the z?
```

```
    end do;
  end do:
  return(Z) :
  end proc:
```

```
>
```

Find the DSR graph from labels, A and Z

```
> ##Create signed DSR graph: entries are two matrices and the labels of the nodes
  createDSRgraphsigned := proc(mynodes, A, Z)
    local G, n, m, Adj, varsZ, Zsign, varsA, Asign, X:
    n := Dimension(A)[1] : m := Dimension(A)[2] :
    X := Transpose(Z) :
    varsZ := indets(X) :
    Zsign := subs(seq(varsZ[i] = 1, i = 1 ..numelems(varsZ)), X) :

    Adj := Matrix(n + m, n + m) :
    Adj[[n + 1 ..n + m], [1 ..n]] := Transpose(map(signum, A)) :
    Adj[[1 ..n], [n + 1 ..n + m]] := Transpose(Zsign) :

    G := GraphTheory[Graph](mynodes, Adj, weighted = true) :
    return(G) :
  end proc:
```

Find the DSR graph from labels and A and return the list of edges:

```
> findedgesDSR := proc(mynodes, A)
    local G, Z :
    Z := findZ(A) :
    G := createDSRgraphsigned(mylabels, A, Z) :
    return(Edges(G, weights)) :
  end proc:
```

```
>
```

Step 2. Procedures to test for multistationarity

Procedures

```
> ## compute Mtilde determinant
  computdet := proc(N, X)
   local M, Mt, F, i, bigdet, n1, s1 :
   n1 := Dimension(N)[1] : s1 := Rank(N) :
   M := N.X :
   Mt := M :
  if s1 < n1 then
    F
      := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
     N)))])))) :
   for i from 1 by 1 to Dimension(F)[1] do
      Mt[ArrayTools[SearchArray](F[i])[1]] := F[i] :
   end do:
  end if:
  bigdet := expand(Determinant(Mt)) :
  return(bigdet) :
  end proc:

> ## compute Mtilde determinant
  computdet2 := proc(N, X)
   local M, Mt, F, i, bigdet, n1, s1 :
   n1 := Dimension(N)[1] : s1 := Rank(N) :
   M := N.X :
   Mt := M :
  if s1 < n1 then
    F
      := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
     N)))])))) :
   for i from 1 by 1 to Dimension(F)[1] do
      Mt[ArrayTools[SearchArray](F[i])[1]] := F[i] :
   end do:
  end if:
  bigdet := expand(Determinant(Mt)) :
  return(bigdet, Mt) :
  end proc:
>

> ## injectivity check
  injective := proc(N, X)
   local det, signs, i, l, k :
   det := computdet(N, X) :
   signs := ListTools[MakeUnique](map(sign, [coeffs(det)])) :
   if det ≠ 0 then
      i := (-1)^(numelems(signs) + 1) :
```

```
    else i := 0 : end if:
     return (i, det) :
    end proc:
> ## injectivity check
    injectivePW := proc(N, X)
     local det, signs, i, n, m, X1, l, k :
    n := Dimension(N)[1] : m := Dimension(N)[2] :
     X1 := DiagonalMatrix(Vector([seq(k_i, i = 1 ..m)])).X
         .DiagonalMatrix(Vector([seq(l_i, i = 1 ..n)])) :

     det := computdet(N, X1) :
     signs := ListTools[MakeUnique](map(sign, [coeffs(det)])) :
      if det ≠ 0 then
          i := (-1)^(numelems(signs) + 1) :
    else i := 0 : end if:
     return (i, det) :
    end proc:


> findV := proc(A)
      local V, n, m, i, j :
       n := Dimension(A)[1] :
       m := Dimension(A)[2] :
        V := Matrix(n, m) :
     for i from 1 to n by 1 do
       for j from 1 to m by 1 do
          if A[i, j] < 0 then V[i, j] := -A[i, j]; end if;
       end do;
     end do:
      return (Transpose(V)) :
      end proc:


> isinjective := proc(A)
      local V, i, det :
      V := findV(A) :
      i, det := injectivePW(A, V) :
      if i = 1 then return (1) :
      else
        return (0) :
      end if:
     end proc:



> ## compute Mtilde determinant with F given
    computdetF := proc(N, X, F)
```

```
    local M, Mt, i, bigdet, n1, s1, sp :
    n1 := Dimension(N)[1] : s1 := Rank(N) : sp := Dimension(X)[2] :
    M := N.X :
    Mt := Matrix(sp, sp) :
     Mt[1..Dimension(F)[1]] := F :
     Mt[Dimension(F)[1]+1..sp] := M :

    bigdet := expand(Determinant(Mt)) :
    return(bigdet) :
   end proc:
```

```
> ## compute Mtilde determinant with F given
   computdetF2 := proc(N, X, F)
    local M, i, bigdet, n1, s1, sp, Mt :
    n1 := Dimension(N)[1] : s1 := Rank(N) : sp := Dimension(X)[2] :
    M := N.X :
    Mt := Matrix(sp, sp) :
     Mt[1..Dimension(F)[1]] := F :
     Mt[Dimension(F)[1]+1..sp] := M :

    bigdet := expand(Determinant(Mt)) :
    return(bigdet, Mt) :
   end proc:
```

```
> ## injectivity check with F given
   injectivePWF := proc(N, X, F)
    local det, signs, i, n, m, X1, l, k :
   n := Dimension(X)[2] : m := Dimension(X)[1] :
    X1 := DiagonalMatrix(Vector([seq(k_i, i=1..m)])).X
        .DiagonalMatrix(Vector([seq(l_i, i=1..n)])) :

    det := computdetF(N, X1, F) :
    signs := ListTools[MakeUnique](map(sign, [coeffs(det)])) :
     if det ≠ 0 then
         i := (-1)^(numelems(signs)+1) :
   else i := 0 : end if:
     return(i, det) :
   end proc:
```

```
> ## injectivity check with F given
   injectivePWF2 := proc(N, X, F)
    local det, signs, i, n, m, X1, l, k, Mt :
   n := Dimension(X)[2] : m := Dimension(X)[1] :
    X1 := DiagonalMatrix(Vector([seq(k_i, i=1..m)])).X
        .DiagonalMatrix(Vector([seq(l_i, i=1..n)])) :
```

```
    det, Mt := computdetF2(N, X1, F) :
    signs := ListTools[MakeUnique](map(sign, [coeffs(det)])) :
     if det ≠ 0 then
         i := (-1)^(numelems(signs) + 1) :
    else i := 0 : end if:
     return(i, det, Mt) :
    end proc:


> ## check if after gauss reduction the system becomes injective: check all
    gaussinjallpos := proc(B, V, F)
        local m, myset, n, i, det, bigset, seenset, B2, V2, B1, myset2, control, control2 :
       m := Dimension(B)[2] : n := Dimension(B)[1] :
        myset := {seq(i, i = 1 ..n)} : control := 0 :
        bigset := {seq(i, i = 1 ..m)} :
       seenset := [myset] :

      while myset ≠ FAIL and control = 0 do
        B2 := B[ .., [op(myset), op(bigset minus myset)]] :
        V2 := V[ [op(myset), op(bigset minus myset)],.. ] :
         i, det, B1 := gaussinj(B2, V2, F) :
        if i = 1 then control := 1 : end if:  ## injective found
        if i = 2 then control := 2 : end if:  ## no positive steady states
        myset2 := { } :
        for i from 1 by 1 to Dimension(B1)[1] do
            myset2 := {op(myset2), ArrayTools[SearchArray](B1[i])[1]} :
        end do:
        seenset := [op(seenset), myset2] :
        seenset := [op(seenset), myset] :
        seenset := ListTools[MakeUnique](seenset) :
        while member(myset, seenset) and myset ≠ FAIL do
          myset := nextcomb(myset, m) :
       end do:
      end do:
       control2 := 0 :
      if control = 1 then
          while myset ≠ FAIL and control = 1 do
            B2 := B[ .., [op(myset), op(bigset minus myset)]] :
            B1 := ReducedRowEchelonForm(B2) :
            control2 := gausssamesign(B1) :
           if control2 = 1 then control := 2 : end if:
          seenset := [op(seenset), myset] :
           seenset := ListTools[MakeUnique](seenset) :
          while member(myset, seenset) and myset ≠ FAIL do
            myset := nextcomb(myset, m) :
          end do:
        end do:
```

```
    end if:
    return(control) :
  end proc:
>
```

```
> ## check if after gauss reduction the system becomes injective: check all
  gausssignall := proc(B)
    local m, myset, n, i, det, bigset, seenset, B2, V2, B1, myset2, control, control2 :
    m := Dimension(B)[2] : n := Dimension(B)[1] :
    myset := {seq(i, i = 1..n)} :
    bigset := {seq(i, i = 1..m)} :

   control2 := 0 :
      while myset ≠ FAIL and control2 = 0 do
        B2 := B[.., [op(myset), op(bigset minus myset)]] :
        B1 := ReducedRowEchelonForm(B2) :
        control2 := gausssamesign(B1) :
        myset := nextcomb(myset, m) :
    end do:
   return(control2) :
  end proc:
>
```

```
> ## check if after gauss reduction the system becomes injective
  gaussinj := proc(N, V, F)
    local p, mysigns, control2, myset, B, i, m, n, maxcols, B1, B3, V3, j, maxcols2, count,
      indiceslist, mypos, B4, V4, control, M, Mt, det, signs, sp, det2, l, k :
   m := Dimension(N)[2] : n := Dimension(N)[1] : sp := Dimension(V)[2] :
   B1 := N :
   maxcols := numelems(ArrayTools[SearchArray](B1)) − n :
   B3 := Matrix(Dimension(B1)[1], maxcols) :
   V3 := Matrix(maxcols, sp) :

  count := 1 : indiceslist := [ ] :
  for i from 1 by 1 to Dimension(B1)[1] do
      maxcols2 := ArrayTools[SearchArray](B1[i]) :
      for j from 2 by 1 to numelems(maxcols2) do
        control := 0 :
        mypos := ListTools[Search](maxcols2[j], indiceslist) :
        if mypos ≠ 0 then
          ##print(i, j, mypos) :
           if Equal(V[maxcols2[j]] − V[maxcols2[1]], V3[mypos]) then
                B3[i, mypos] := -B1[i, maxcols2[j]] : control := 1 :
           end if:
        end if:
        if control = 0 then
```

```
            V3[count] := V[maxcols2[j]] - V[maxcols2[1]]:
            indiceslist := [op(indiceslist), maxcols2[j]]:
            B3[i, count] := -B1[i, maxcols2[j]]:
            count := count + 1:
          end if:
        end do:

      end do:
      B4 := SubMatrix(B3, [1..Dimension(B3)[1]], [1..count - 1]):
      V4 := SubMatrix(V3, [1..count - 1], [1..Dimension(V3)[2]]):

      M := B4.DiagonalMatrix(Vector([seq(k_i, i = 1..Dimension(V4)[1])])).V4
        .DiagonalMatrix(Vector([seq(l_i, i = 1..Dimension(V4)[2])])):
      Mt := Matrix(sp, sp):
      Mt[1..Dimension(F)[1]] := F:
      Mt[Dimension(F)[1] + 1..sp] := M:

      ##injectivity test
      det := Determinant(Mt):
      det2 := collect(det, indets(det),'distributed'):
      signs := ListTools[MakeUnique](map(sign, [coeffs(det2)])):
      if det2 ≠ 0 then
        i := (-1)^(numelems(signs) + 1):
      else i := 0: end if:
      return(i, det, B1):
    end proc:

> ## check if after gauss reduction the system becomes injective and return also the
    matrices
    gaussinjV := proc(N, V, F)
      local p, mysigns, control2, myset, B, i, m, n, maxcols, B1, B3, V3, j, maxcols2, count,
        indiceslist, mypos, B4, V4, control, M, Mt, det, sp, signs, det2:
     m := Dimension(N)[2]: n := Dimension(N)[1]: sp := Dimension(V)[2]:
     B1 := N:
     maxcols := numelems(ArrayTools[SearchArray](B1)) - n:
     B3 := Matrix(Dimension(B1)[1], maxcols):
     V3 := Matrix(maxcols, sp):

    count := 1: indiceslist := []:
    for i from 1 by 1 to Dimension(B1)[1] do
      maxcols2 := ArrayTools[SearchArray](B1[i]):
      for j from 2 by 1 to numelems(maxcols2) do
        control := 0:
        mypos := ListTools[Search](maxcols2[j], indiceslist):
        if mypos ≠ 0 then
          ##print(i, j, mypos):
          if Equal(V[maxcols2[j]] - V[maxcols2[1]], V3[mypos]) then
            B3[i, mypos] := -B1[i, maxcols2[j]]: control := 1:
```

```
            end if:
        end if:
        if control = 0 then
            V3[count] := V[maxcols2[j]] - V[maxcols2[1]]:
            indiceslist := [op(indiceslist), maxcols2[j]]:
            B3[i, count] := -B1[i, maxcols2[j]]:
            count := count + 1:
        end if:
    end do:

    end do:
    B4 := SubMatrix(B3, [1..Dimension(B3)[1]], [1..count - 1]):
    V4 := SubMatrix(V3, [1..count - 1], [1..Dimension(V3)[2]]):

    M := B4.DiagonalMatrix(Vector([seq(h_i, i = 1..Dimension(V4)[1])])).V4
      .DiagonalMatrix(Vector([seq(l_i, i = 1..Dimension(V4)[2])])):
    Mt := Matrix(sp, sp):
    Mt[1..Dimension(F)[1]] := F:
    Mt[Dimension(F)[1] + 1..sp] := M:
    ##injectivity test
    det := Determinant(Mt):
    det2 := collect(det, indets(det),'distributed'):
    signs := ListTools[MakeUnique](map(sign, [coeffs(det2)])):
    if det2 ≠ 0 then
        i := (-1)^(numelems(signs) + 1):
    else i := 0: end if:
    return(i, det, B4, V4):
end proc:
```

```
> ## check if after gauss reduction the system becomes injective: check all
gaussinjall := proc(B, V, F, myred)
    local m, myset, i, n, k, det, bigset, seenset, B2, V2, B1, myset2, myset3, control,
      control2:
    m := Dimension(B)[2]: n := Dimension(B)[1]:
    myset := {seq(i, i = 1..n)}: control := 0:
    bigset := {seq(i, i = 1..m)}:
    seenset := [myset]:

    while myset ≠ FAIL and control = 0 do
      B2 := B[.., [op(myset), op(bigset minus myset)]]:
      V2 := V[[op(myset), op(bigset minus myset)],..]:
      B1 := ReducedRowEchelonForm(B2):

      myset2 := {}:
      for k from 1 by 1 to n do
          myset2 := {op(myset2), ArrayTools[SearchArray](B1[k])[1]}:
      end do:
```

```
        myset3 := {op([op(myset), op(bigset minus myset)][[op(myset2)]])} :
        if not member(myset3, seenset) then
            i, det, B1 := gaussinj(B1, V2, F) :
            if i = 1 then control := 1 : end if:  ## injective found
        end if:
        seenset := [op(seenset), myset3] :
        seenset := [op(seenset), myset] :
        seenset := ListTools[MakeUnique](seenset) :

        ##find new subset, filtered by seenset and the already known independent columns
       control2 := 0 :
       while control2 = 0 do
           control2 := 1 :
           myset := nextcomb(myset, m) :
          if myset ≠ FAIL then
             if member(myset, seenset) then control2 := 0 :
             else
              k := 1 :
              while k ≤ numelems(myred) and control2 = 1 do
                 if subset(myred[k], myset) then control2 := 0 : end if:
                 k := k + 1 :
               end do:
            end if:
          end if:
        end do:
     end do:  ##end big do, for myset and control

     return(control) :
    end proc:
>

> ## check if after gauss reduction the system becomes injective: check bistable
    Bistablecheck := proc(B)
       local m, n, j, k, control, control2, myrow, myvec, signvec :
      m := Dimension(B)[2] : n := Dimension(B)[1] :
      j := 1 :
      control2 := 0 :
      control := 2 :
      while control2 = 0 and j ≤ n do
          myrow := convert(ArrayTools[SearchArray](B[j]), list) :
         if numelems(myrow) ≤ 1 then control2 := 1 : control := 3 :
         else
             myvec := convert(B[j][myrow[2..numelems(myrow)]], list) :
             signvec := ListTools[MakeUnique](map(sign, myvec)) :
             if signvec ≠ [-1] then control2 := 1 : control := 0 :
             else
                 k := 2 :
```

```
            while control2 = 0 and k ≤ numelems(myrow) do
                if numelems(ArrayTools[SearchArray](B[ .., myrow[k]])) ≠ 1
    then control2 := 1 : control := 0 : end if:
                k := k + 1 :
            end do:
        end if:
    end if:
    j := j + 1 :
 end do:
return(control) :
end proc:
>
```

```
> ## check if after gauss reduction the system becomes injective: check bistable
Bistablecheck2 := proc(B)
   local m, n, j, k, control, control2, myrow, myvec, signvec, disjsets, nonzerocols,
    totalcard :
  m := Dimension(B)[2] : n := Dimension(B)[1] :
  j := 1 : disjsets := [ ] :
  control2 := 0 :
  control := 2 :
  while control2 = 0 and j ≤ n do
     myrow := convert(ArrayTools[SearchArray](B[j]), list) :
     if numelems(myrow) ≤ 1 then control2 := 1 : control := 3 :
     else
        myvec := convert(B[j][myrow[2 ..numelems(myrow)]], list) :
        signvec := ListTools[MakeUnique](map(sign, myvec)) :
        if signvec ≠ [-1] then control2 := 1 : control := 0 :
        else k := 2 :
            while control2 = 0 and k ≤ numelems(myrow) do
               nonzerocols := ArrayTools[SearchArray](B[ .., myrow[k]]) :
               if numelems(nonzerocols) > 1 then disjsets := [op(disjsets),
    convert(nonzerocols, list)] :
                end if:
               k := k + 1 :
            end do:
        end if:
     end if:
     j := j + 1 :
  end do:
  disjsets := MakeUnique(disjsets) :
  totalcard := 0 :
  for k from 1 by 1 to numelems(disjsets) do  totalcard := totalcard
      + numelems(disjsets[k]) : end do:
  if numelems(MakeUnique(Flatten(disjsets))) ≠ totalcard then control := 0 : end
     if:
```

```
    return(control) :
    end proc:


> ## check if after gauss reduction the system becomes injective: check all
  gaussinjallBi := proc(B, V, F, myred)
      local m, myset, i, n, k, det, bigset, seenset, B2, V2, B1, myset2, myset3, control,
       control2 :
     m := Dimension(B)[2] : n := Dimension(B)[1] :
      myset := {seq(i, i = 1 ..n)} : control := 0 :
      bigset := {seq(i, i = 1 ..m)} :
     seenset := [myset] :

    while myset ≠ FAIL and control = 0 do
      B2 := B[ .., [op(myset), op(bigset minus myset)]] :
      V2 := V[ [op(myset), op(bigset minus myset)],.. ] :
       B1 := ReducedRowEchelonForm(B2) :

       myset2 := { } :
       for k from 1 by 1 to n do
           myset2 := {op(myset2), ArrayTools[SearchArray](B1[k])[1]} :
      end do:
       myset3 := {op([op(myset), op(bigset minus myset)][[op(myset2)]])} :
      if not member(myset3, seenset) then
           i, det, B1 := gaussinj(B1, V2, F) :
           if i = 1 then control := 1 : end if:  ## injective found
           if i = -1 or i = 0 then
                 control := Bistablecheck(B1) :
          end if:
      end if:
      seenset := [op(seenset), myset3] :
      seenset := [op(seenset), myset] :
      seenset := ListTools[MakeUnique](seenset) :

      ## find new subset, filtered by seenset and the already known independent columns
     control2 := 0 :
     while control2 = 0 do
        control2 := 1 :
        myset := nextcomb(myset, m) :
       if myset ≠ FAIL then
          if member(myset, seenset) then control2 := 0 :
          else
           k := 1 :
           while k ≤ numelems(myred) and control2 = 1 do
              if subset(myred[k], myset) then control2 := 0 : end if:
              k := k + 1 :
            end do:
         end if:
```

```
        end if:
      end do:
    end do:  ##end big do, for myset and control

    return(control):
  end proc:
>
```

```
> gaussinjallBi2 := proc(B, V, F, myred)
    local m, myset, i, n, k, det, bigset, seenset, B2, V2, B1, myset2, myset3, control,
     control2, lastseen, myseen:
    m := Dimension(B)[2]: n := Dimension(B)[1]:
    myset := {seq(i, i = 1..n)}: control := 0:
    bigset := {seq(i, i = 1..m)}:
    seenset := []:
    myseen := []:

    while myset ≠ FAIL and control = 0 do
      B2 := B[.., [op(myset), op(bigset minus myset)]]:
      V2 := V[[op(myset), op(bigset minus myset)],..]:
      B1 := ReducedRowEchelonForm(B2):
      lastseen := myset: myseen := [op(myseen), myset]:
      myset2 := {}:
      for k from 1 by 1 to n do
          myset2 := {op(myset2), ArrayTools[SearchArray](B1[k])[1]}:
      end do:
      myset3 := {op([op(myset), op(bigset minus myset)][[op(myset2)]])}:
      if not member(myset3, seenset) then
          i, det, B1 := gaussinj(B1, V2, F):
          if i = 1 then control := 1: end if:  ## injective found
          if i = -1 or i = 0 then
                control := Bistablecheck2(B1):
          end if:
      end if:
      seenset := [op(seenset), myset3]:
      seenset := [op(seenset), myset]:
      seenset := ListTools[MakeUnique](seenset):

      ##find new subset, filtered by seenset and the already known independent columns
      control2 := 0:
      while control2 = 0 do
        control2 := 1:
        myset := nextcomb(myset, m):
        if myset ≠ FAIL then
          if member(myset, seenset) then control2 := 0:
          else
           k := 1:
```

```
            while k ≤ numelems(myred) and control2 = 1 do
              if subset(myred[k], myset) then control2 := 0 : end if:
              k := k + 1 :
            end do:
          end if:
        end if:
      end do:
    end do:  ##end big do, for myset and control

    return(control, myset, B1, lastseen, myseen) :
  end proc:
```

```
> isinjectiveextended := proc(A)
    local V, M, F, i, n1, s1, B, toexclude, myred, control, myset, B1, lastseen, myseen :
    myred := { } :
    V := findV(A) :
    n1 := Dimension(A)[1] : s1 := Rank(A) :
    toexclude := [ ] :
    if s1 < n1 then
      F
      := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
      A)))])))) :
      for i from 1 by 1 to Dimension(F)[1] do
        toexclude := [op(toexclude), ArrayTools[SearchArray](F[i])[1]] :
      end do:
    else
      F := [ ] :
    end if:
    B := SubMatrix(A, [op({seq(i, i = 1 .. n1)} minus {op(toexclude)})], [1
      ..Dimension(A)[2]]) :
    if Dimension(B)[1] < Dimension(B)[2] then
      control, myset, B1, lastseen, myseen := gaussinjallBi2(B, V, F, myred) :
      return(control) :
    else
      return(0) :
    end if:
  end proc:
```

```
>
```

Step 3. Finding the positive feedback loops for multistationary networks

Auxiliary procedures

```
> addlist := proc(mylist, myaddlist)
    local i, newlist :
    newlist := [op(mylist), op(myaddlist)] :
    return newlist :
  end proc:
```

This procedure computes the polynomial $p_{A,Z}$ in the main text. The input are the matrices A and Z (in the function denoted N and X).

```
> ## compute Mtilde determinant
  computdetS := proc(N, X)
    local M, Mt, F, i, bigdet, n1, s1 :
    n1 := Dimension(N)[1] : s1 := Rank(N) :
    M := N.Transpose(X) :
    Mt := M :
    if s1 < n1 then
      F
        := ReducedRowEchelonForm(Transpose(Matrix([op(NullSpace(Transpose(
      N)))])))) :
    for i from 1 by 1 to Dimension(F)[1] do
        Mt[ArrayTools[SearchArray](F[i])[1]] := F[i] :
    end do:
  end if:
  bigdet := expand(Determinant(Mt)) :
  return(bigdet) :
  end proc:
```

This function returns the list of monomials that have the wrong sign. The input are the determinant and the wrong sign.

```
> ##Given a determinant and a wrong sign, return the list of wrong monomials
  badterms := proc(deter, mysign)
      local vars, coeflist, monomlist, coeflistsign, wterms, i :
      vars := indets(deter) :
      coeflist := [coeffs(deter, vars, 't')] :
      monomlist := [t] :
      coeflistsign := map(sign, coeflist) :
      wterms := [ ] :
      for i from 1 by 1 to numelems(coeflistsign) do
          if mysign = coeflistsign[i] then wterms := [op(wterms), monomlist[i]] : end
       if:
      end do:
```

```
    return(wterms) :

  end proc:
```

Given a monomial on the entries of a matrix Amatrix, this function finds a matrix from Amatrix such that the variables in the monomial become 1 and the rest are zero.

```
> ##Find submatrix of a matrix corresponding to a monomial
findmatrix := proc(wmonom, Amatrix)
    local vars, wvarscomp, Anew, i, Anew1 :
    vars := indets(wmonom) :
    wvarscomp := indets(Amatrix) minus vars :
    Anew := subs(seq(wvarscomp[i] = 0, i = 1 ..numelems(wvarscomp)), Amatrix) :
    Anew1 := subs(seq(vars[i] = 1, i = 1 ..numelems(vars)), Anew) :
    return(Anew1) :
  end proc:
```

Find the two submatrices of A and Z corresponding to the monomial, and make A symbolic by introducing a new variable x.

```
> ## Find the two matrices A,Z corresponding to a monomial
twomatrices := proc(wmonom, A, Z)
    local vars, wvarscomp, Zembed, i, row, col, Aembed, Aembedx, nZ, X :
    X := Transpose(Z) :
    vars := indets(wmonom) :
    wvarscomp := indets(X) minus vars :
    Zembed := subs(seq(wvarscomp[i] = 0, i = 1 ..numelems(wvarscomp)), X) :
    row, col := ArrayTools[SearchArray](Zembed) :
    Zembed := Zembed[convert(row, list), convert(col, list)] :
    nZ := numelems(vars) :
    Aembed := A[convert(col, list), convert(row, list)] :
    Aembed := map(signum, Aembed) :
    row, col := ArrayTools[SearchArray](Aembed) :
    Aembedx := Matrix(nZ, nZ) :
    for i from 1 by 1 to numelems(col) do
      Aembedx[row[i], col[i]] := Aembed[row[i], col[i]]·x_i :
    end do:
    return(Aembedx, Zembed) :
  end proc:
```

Extract the indices of the species and the reaction in the given monomial in the variables of Z

```
> ## Extract indices of species and reaction in the monomial from Z
extractsr := proc(wmonom, Z)
    local vars, wvarscomp, Zembed, i, row, col, X :
    X := Transpose(Z) :
    vars := indets(wmonom) :
    wvarscomp := indets(X) minus vars :
```

```
    ## indeterminates not in the monomial
    Zembed := subs(seq(wvarscomp[i] = 0, i = 1 ..numelems(wvarscomp)), X) :
    ##set the entries of the indeterminates not in the monomial to zero
    row, col := ArrayTools[SearchArray](Zembed) :
    ##find the nonzero entries of the resulting matrix.
    return(row, col) :  ##return the species and reaction indices
  end proc:
```

```
> ##Create DSR graph: entries are two matrices and the labels of the nodes
createDSRgraph := proc(mynodes, A, Z)
  local G, n, m, Adj, varsZ, Zsign, varsA, Asign, X:
  n := Dimension(A)[1] : m := Dimension(A)[2] :
  X := Transpose(Z) :
  varsZ := indets(X) :
  Zsign := subs(seq(varsZ[i] = 1, i = 1 ..numelems(varsZ)), X) :
  varsA := indets(A) :
  Asign := subs(seq(varsA[i] = 1, i = 1 ..numelems(varsA)), A) :

  Adj := Matrix(n + m, n + m) :
  Adj[[n + 1 ..n + m], [1 ..n]] := Transpose(map(signum, Asign)) :
  Adj[[1 ..n], [n + 1 ..n + m]] := Transpose(Zsign) :

  G := GraphTheory[Graph](mynodes, Adj, weighted = true) :
  return(G) :
end proc:
>
```

This function selects the subgraphs that give rise to the monomials with the wrong sign.

```
> ## Select the subgraphs that correspond to the wrong terms of A and Z
graphlist := proc(mydet, A, Z)
 local srlist, row, col, Gsub, s, wsign, wrongterms, k, wcurrent, Aembedx, Zembed,
     detZ, detAx, wsignA, wrongtermsA, wcurrentA, j, Aembedx1, mynodes :
Gsub := [ ]:
 srlist := [ ]:
 s := Rank(A) :
wsign := (-1)^(s + 1) :                    ## find wrong sign
wrongterms := badterms(mydet, wsign) :
     ## select the monomials with the wrong sign
 for k from 1 by 1 to numelems(wrongterms) do
     ## for each such monomial, find the associated subgraph
     wcurrent := wrongterms[k] :
     row, col := extractsr(wcurrent, Z) :
     ## find the indices of the species and the reactions in the monomial
     mynodes := [seq(S[col[i]], i = 1 ..numelems(col)), seq(R[row[i]], i = 1
      ..numelems(row))] :
```

```
    Aembedx, Zembed := twomatrices(wcurrent, A, Z) :
    ## the returned Zembed is giving half of the edges of the subgraphs
    detZ := subs(seq(indets(Zembed)[i] = 1, i = 1 ..numelems(indets(Zembed))),
     Determinant(Zembed)) :
    detAx := expand(Determinant(Aembedx)) :
    wsignA := wsign·detZ :
    wrongtermsA := badterms(detAx, wsignA) :
    ## select the monomials with the wrong sign of the subsystem
    for j from 1 by 1 to numelems(wrongtermsA) do
      wcurrentA := wrongtermsA[j] :
      Aembedx1 := findmatrix(wcurrentA, Aembedx) :
    ## find the other half of the edges of the subgraphs
      Gsub := [op(Gsub), createDSRgraph(mynodes, Aembedx1, Zembed)] :
    end do:
  end do:
  return(Gsub) :  ##return the list of graphs
 end proc:
```

Given a list of edges that form a loop, the function returns the edges ordered such that connected they form the loop.

```
> ## Order the edges to have a loop
 orderedge := proc(myedges)
   local ordereddedges, endpoint, total, control, k :
  ordereddedges := [myedges[1]] :
  endpoint := myedges[1][1][2] :
  total := numelems(myedges) :
 while numelems(ordereddedges) < total do
    control := 0 : k := 2 :
    while control = 0 do
      if endpoint = myedges[k][1][1] then
          ordereddedges := [op(ordereddedges), myedges[k]] :
          control := 1 :
         endpoint := myedges[k][1][2] :

      end if:
      k := k + 1 :
   end do:
 end do:
  return(ordereddedges) :
 end proc:
```

Find the sequence of signs of the loop

```
> ##Extract the  sequence  of signs of a loop
 extractsign := proc(ordereddedges)
   local graphsign, i :
   graphsign := [ ] :
```

```
    for i from 1 by 1 to numelems(orderededges) do
      graphsign := [op(graphsign), orderededges[i][2]]:
    end do:
    return(graphsign):
  end proc:
```

Given a list of graphs, we find the positive feedback loops that they contain and return the sign pattern of each positive feedback loop as well (as those given in Table 1 in the main text).

```
> ##Find the positive feedback loops in the list of graphs
  positivefeed := proc(Gsub)
    local selected, j, mygraph, Gsubcomp, k, mycomp, newgraph, wedges, myprod, i,
      signcycle:
    selected := []:
    signcycle := []:
    Gsubcomp := []:
    for j from 1 by 1 to numelems(Gsub) do
      mygraph := Gsub[j]:
      Gsubcomp := ConnectedComponents(mygraph):
      for k from 1 by 1 to numelems(Gsubcomp) do
        mycomp := Gsubcomp[k]:
        newgraph := InducedSubgraph(mygraph, mycomp):
        wedges := Edges(newgraph, weights):
        myprod := mul(wedges[i][2], i = 1..numelems(wedges)):
        if myprod = 1 then          ##if the loop is positive, select it
          selected := [op(selected), [op(wedges)]]:
        end if:
      end do:
    end do:
    selected := ListTools[MakeUnique](selected):
    for k from 1 by 1 to numelems(selected) do
      selected[k] := orderedge(selected[k]):
      signcycle := [op(signcycle), extractsign(selected[k])]:
    end do:
    return(selected, signcycle):
  end proc:
```

The main procedure to find the positive loop is the following:

```
> ##main program: find the positive loops
  findloops := proc(A, Z)
    local Gsub, selected, signcycle, mydet:
```

```
    mydet := computdetS(A, Z) :  ## find the polynomial p_{A, Z}
    Gsub := graphlist(mydet, A, Z) :
      ## find the list of subgraphs corresponding to the wrong signs
    selected, signcycle := positivefeed(Gsub) :
      ## find the positive feedback loops and their sign pattern
    return(selected) :
  end proc:
```

The second main procedure of the method is the function that draws the selected positive feedback loops. It requires a list with the names of the nodes (see the examples below)

```
> ## draw the positive feedback loops
drawloops := proc(selected, speciesord)
  local loops, i, vertices, speciesdic, selected2 :
 loops := [ ]:
 speciesdic := {seq(S_i = speciesord[i], i = 1 ..numelems(speciesord))} :
 selected2 := subs(speciesdic, selected) :
for i from 1 by 1 to numelems(selected2) do
     vertices := ListTools[MakeUnique]([seq(op(selected2[i][j][1]), j = 1
     ..numelems(selected2[i]))]) :
     loops := [op(loops), Digraph(vertices, {op(selected2[i])})] :
 end do:
DrawGraph(loops, style = circle);

 end proc:
```

Functions for constructing stoichiomatric matrix and examine the existence of competition and closed loop.

1. To enumerate stoichiomatric vectors (reaction patterns)

List all reaction types based on the number of species. $R_n = \binom{n}{2}\cdot 2 + \binom{n}{3}\cdot 6$

```
> listRs := proc(n)
      local R, r, se, i, j, k, sign, l :
      r := binomial(n, 2)·2 + binomial(n, 3)·6 :
      R := Matrix(r, n) :
      # Now construct the reaction pattern matrix
```

```
    i := 1 :

    # here we first consider single transformation
    for se from -1 to 1 by 2 do
      for j from 1 to n by 1 do
        for k from j + 1 to n by 1 do
          R[i, j] := se :
          R[i, k] := -se :
          i := i + 1 :
        end do:
      end do:
    end do:

    # now consider heterodimerization and disassociation
    for sign from -1 to 1 by 2 do
      for j from 1 to n by 1 do
        for k from j + 1 to n by 1 do
          for l from k + 1 to n by 1 do
            R[i, j] := sign :
            R[i, k] := -sign :
            R[i, l] := -sign :
            i := i + 1 :
            R[i, j] := sign :
            R[i, k] := sign :
            R[i, l] := -sign :
            i := i + 1 :
            R[i, j] := sign :
            R[i, k] := -sign :
            R[i, l] := sign :
            i := i + 1 :
          end do:
        end do:
      end do:
    end do:

    # in this case we don't consider homodimerization and disassociation

    # Here transpose the R
    # R:=Transpose(R) :
     # no need to ranspose, need to assign rows to untransposed A's rows
    return(R) :
  end proc:
```

Here is a long function to enumerate first two reaction pattern (will reduce large mount of symmetric reactions).

```
> listR2 := proc(n)
    local R2, r2, R3, r3, R4, r4 :
    if n ≥ 4 then
      r2 := 29·2 :
      R2 := Matrix(r2, 4) :

      # in this case we don't consider homodimerization and disassociation

      ## now we construct the reaction pattern with n = 4
      R2[1] := ⟨1,-1⟩ : R2[2] := ⟨0, 0, 1,-1⟩ :
      R2[3] := ⟨1,-1⟩ : R2[4] := ⟨0, 1,-1⟩ :
      R2[5] := ⟨1,-1⟩ : R2[6] := ⟨1, 0,-1⟩ :
      R2[7] := ⟨1,-1⟩ : R2[8] := ⟨0,-1, 1⟩ :
      R2[9] := ⟨1,-1⟩ : R2[10] := ⟨-1, 0, 1⟩ :
      R2[11] := ⟨1,-1⟩ : R2[12] := ⟨-1, 1⟩ :
      R2[13] := ⟨1,-1⟩ : R2[14] := ⟨0, 1,-1,-1⟩ :
      R2[15] := ⟨1,-1⟩ : R2[16] := ⟨1, 0,-1,-1⟩ :
      R2[17] := ⟨1,-1⟩ : R2[18] := ⟨0,-1, 1,-1⟩ :
      R2[19] := ⟨1,-1⟩ : R2[20] := ⟨-1, 0, 1,-1⟩ :
      R2[21] := ⟨1,-1⟩ : R2[22] := ⟨0, 1, 1,-1⟩ :
      R2[23] := ⟨1,-1⟩ : R2[24] := ⟨1, 0, 1,-1⟩ :
      R2[25] := ⟨1,-1⟩ : R2[26] := ⟨0,-1, 1, 1⟩ :
      R2[27] := ⟨1,-1⟩ : R2[28] := ⟨-1, 0, 1, 1⟩ :
      R2[29] := ⟨1,-1⟩ : R2[30] := ⟨-1,-1, 1⟩ :
      R2[31] := ⟨1,-1⟩ : R2[32] := ⟨1, 1,-1⟩ :
      R2[33] := ⟨1,-1,-1⟩ : R2[34] := ⟨0, 1,-1,-1⟩ :
      R2[35] := ⟨1,-1,-1⟩ : R2[36] := ⟨1, 0,-1,-1⟩ :
      R2[37] := ⟨1,-1,-1⟩ : R2[38] := ⟨0,-1,-1, 1⟩ :
      R2[39] := ⟨1,-1,-1⟩ : R2[40] := ⟨-1,-1, 0, 1⟩ :
      R2[41] := ⟨1,-1,-1⟩ : R2[42] := ⟨0, 1, 1,-1⟩ :
      R2[43] := ⟨1,-1,-1⟩ : R2[44] := ⟨1, 1, 0,-1⟩ :
      R2[45] := ⟨1,-1,-1⟩ : R2[46] := ⟨0, 1,-1, 1⟩ :
      R2[47] := ⟨1,-1,-1⟩ : R2[48] := ⟨-1, 1, 0, 1⟩ :
      R2[49] := ⟨1,-1,-1⟩ : R2[50] := ⟨-1, 1, 1⟩ :
      R2[51] := ⟨1, 1,-1⟩ : R2[52] := ⟨0, 1, 1,-1⟩ :
      R2[53] := ⟨1, 1,-1⟩ : R2[54] := ⟨1, 1, 0,-1⟩ :
      R2[55] := ⟨1, 1,-1⟩ : R2[56] := ⟨0, 1,-1, 1⟩ :
      R2[57] := ⟨1, 1,-1⟩ : R2[58] := ⟨1,-1, 0, 1⟩ :


      if n ≥ 5 then
        ## now we add the reaction pattern with n = 5

        r3 := 43·2 :
        R3 := Matrix(r3, 5) :
        R3[1..r2] := R2[ ] :

        R3[59] := ⟨1,-1⟩ : R3[60] := ⟨0, 0, 1,-1,-1⟩ :
```

```
        R3[61] := ⟨1,-1⟩ : R3[62] := ⟨0, 0, 1, 1,-1⟩ :
        R3[63] := ⟨1,-1,-1⟩ : R3[64] := ⟨0, 0, 1,-1,-1⟩ :
        R3[65] := ⟨1,-1,-1⟩ : R3[66] := ⟨1, 0, 0,-1,-1⟩ :
        R3[67] := ⟨1,-1,-1⟩ : R3[68] := ⟨0, 0,-1, 1,-1⟩ :
        R3[69] := ⟨1,-1,-1⟩ : R3[70] := ⟨-1, 0, 0, 1,-1⟩ :
        R3[71] := ⟨1,-1,-1⟩ : R3[72] := ⟨0, 0, 1, 1,-1⟩ :
        R3[73] := ⟨1,-1,-1⟩ : R3[74] := ⟨1, 0, 0, 1,-1⟩ :
        R3[75] := ⟨1,-1,-1⟩ : R3[76] := ⟨0, 0,-1, 1, 1⟩ :
        R3[77] := ⟨1,-1,-1⟩ : R3[78] := ⟨-1, 0, 0, 1, 1⟩ :
        R3[79] := ⟨1, 1,-1⟩ : R3[80] := ⟨0, 0, 1, 1,-1⟩ :
        R3[81] := ⟨1, 1,-1⟩ : R3[82] := ⟨1, 0, 0, 1,-1⟩ :
        R3[83] := ⟨1, 1,-1⟩ : R3[84] := ⟨0, 0,-1, 1, 1⟩ :
        R3[85] := ⟨1, 1,-1⟩ : R3[86] := ⟨-1, 0, 0, 1, 1⟩ :

        if n ≥ 6 then
          ## now we add the reaction pattern with n = 6

          r4 := 46·2 :
          R4 := Matrix(r4, n) :
          R4[1..r3] := R3[ ] :

          R4[87] := ⟨1,-1,-1⟩ : R4[88] := ⟨0, 0, 0, 1,-1,-1⟩ :
          R4[89] := ⟨1,-1,-1⟩ : R4[90] := ⟨0, 0, 0, 1, 1,-1⟩ :
          R4[91] := ⟨1, 1,-1⟩ : R4[92] := ⟨0, 0, 0, 1, 1,-1⟩ :

          return(R4) :
        else
          return(R3) :
        end if:
      else
        return(R2) :
      end if:
    else
      error "ERROR: n is smaller than 4"
    end if:

  end proc:
```

2. Now we can construct stoichiomatric matrix based on the reaction patterns, and examine their properties.

Here, we construct the stoichiomatric matrices.

The total number of stoichiomatric matrices is $\binom{R_n}{m}$, which is still a huge number. But currently there seems no other better options.

We only consider when $m \leq 6$.

The function(s) to examine existence of competition and loops a stoichiomatric matrix.

This function is used to check the existence of competition in the system

```
> existcompetition := proc(A)
    local i, j, m, n, count, check, An, Rs, Cs, Checks :
    An := (A - |A|)/2 :
    Rs := AddAlongDimension(A, 2) :
    Cs := AddAlongDimension(An, 1) :
    m := Dimension(A)[1] : n := Dimension(A)[2] :

    count := 0 :
    for j from 1 to n by 1 do
      if Cs[j] ≤ -2 then
        check := 0 :
        for i from 1 to m by 1 do
          if Rs[i] = -1 and A[i, j] ≤ -1 then
            check := check + 1 :
          end if:
        end do:

        if check ≥ 2 then
          count := count + 1 :
          return(count) :
        end if:
      end if:
    end do:
    return(count) :
  end proc:
```

This function is to check the existence of closed positive feedback loop with competition. It returns
a number if 0 then no competition, if 1 then only competition no loop, if 2 then with competition loop no switching (back), if 3 then with competition loop and switching (back).

```
> existcompetitionloop := proc(A)
    local i, j, m, n, count, check, An, Rs, Cs, Checks, exist, k, l, loops, switches, comps, p,
     q, v, r, s :
    An := (A - |A|)/2 :
    Rs := AddAlongDimension(A, 2) :
    Cs := AddAlongDimension(An, 1) :
    m := Dimension(A)[1] : n := Dimension(A)[2] :

    count := 0 : exist := 0 : loops := 0 : switches := 0 :
    #print(Rs);
```

```
#print(Cs);
for j from 1 to n by 1 do
   if Cs[j] ≤ -2 then
      check := 0 :
      Checks := Array( ) :
      k := 0 :
      for i from 1 to m by 1 do
         if Rs[i] = -1 and A[i,j] ≤ -1 then
            k := k + 1 :
            check := check + 1 :
            Checks(k) := i :
         end if:
      end do:
      #print(check);
      if check ≥ 2 then
         count := count + 1 :
         #print(count);
         comps := Size(Checks, 2) :
         v := Vector(5) :
         v[3] := j :
         for p from 1 to comps by 1 do
            v[1] := Checks(p) :
            if A[Checks[p], j] = -1 then
               for r from 1 to n by 1 do
                  if A[Checks[p], r] = -1 and r ≠ j then
                     v[4] := r :
                  end if:
               end do:
               if v[4] = 0 then
                  error "Can not find the other reactant" :
               end if:
            elif A[Checks[p], j] = -2 then
               v[4] := j :
            else
               error "The reactant is neither -1 nor -2" :
            end if:
            for q from p + 1 to comps by 1 do
               v[2] := Checks(q) :
               if A[Checks[q], j] = -1 then
                  for r from 1 to n by 1 do
                     if A[Checks[q], r] = -1 and r ≠ j then
                        v[5] := r :
                     end if:
                  end do:
                  if v[5] = 0 then
                     error "Can not find the other reactant for second reaction" :
                  end if:
               elif A[Checks[q], j] = -2 then
```

```
                    v[5] := j :
                  else
                    error "The reactant is neither -1 nor -2 in second reaction" :
                  end if:
                  #print(v);
                  if v[4] ≠ v[5] then
                    l := 0 :
                    l := checkloop(A, v) :
                    if l = 1 then
                      loops := loops + 1 :
                      #print(loops);
                    elif l = 2 then
                      switches := switches + 1 :
                      exist := 3 :
                      return(exist) :
                    end if:
                  end if:
                end do:
              end do:
            end if:
          end if:
        end do:

        if count ≥ 1 then
          if loops ≥ 1 then
            exist := 2 :
          else
            exist := 1 :
          end if:
        end if:
        return(exist) :
      end proc:
>
```

This function is used to check the competition loop (return 1) and switches (return 2), if no exist any of those return 0.

```
> checkloop := proc(A, v)
    local exist, loop, switch, Q, i, j, k, m, n, visited :
    m := Dimension(A)[1] : n := Dimension(A)[2] :
    loop := 0 :
    switch := 0 :
    exist := 0 :
    Q := queue[new]( ) :
    visited := Vector(n) :
    queue[enqueue](Q, v[4]) :
    while not queue[empty](Q) do
      j := queue[dequeue](Q) :
```

```
    for i from 1 to m do
      if i ≠ v[1] and i ≠ v[2] then
        if A[i,j] ≤ -1 then
          for k from 1 to n do
            if A[i, k] ≥ 1 then
              if k = v[5] then
                loop := loop + 1 :
                break:
              elif k ≠ v[4] then
                if visited[k] = 0 then
                  queue[enqueue](Q, k) :
                  visited[k] := 1 :
                end if:
              end if:
            end if:
          end do:
          if loop ≥ 1 then
            break:
          end if:
        end if:
      end if:
    end do:
    if loop ≥ 1 then
      queue[clear](Q) :
      break:
    end if:
  end do:

  queue[clear](Q) :
  visited := Vector(n) :
  if loop ≥ 1 then
    exist := 1 :
    for j from 1 to n do
      if j = v[4] then
        switch := switch + 1 :
        exist := 2 :
        return (exist) :
      elif A[v[2], j] ≥ 1 and visited[j] = 0 then
        queue[enqueue](Q, j) :
        visited[j] := 1 :
      end if:
    end do:

    while not queue[empty](Q) do
      j := queue[dequeue](Q) :
      for i from 1 to m do
        if i ≠ v[1] and i ≠ v[2] then
          if A[i, j] ≤ -1 then
```

```
                for k from 1 to n do
                    if A[i, k] ≥ 1 then
                        if k = v[4] then
                            switch := switch + 1 :
                            exist := 2 :
                            return (exist) :
                        elif k ≠ v[5] and visited[k] = 0 then
                            queue[enqueue](Q, k) :
                            visited[k] := 1 :
                        end if:
                    end if:
                end do:
            end if:
        end if:
    end do:
end if:

loop := 0 :
queue[clear](Q) :
queue[enqueue](Q, v[5]) :
visited := Vector(n) :
while not queue[empty](Q) do
    j := queue[dequeue](Q) :
    for i from 1 to m do
        if i ≠ v[2] and i ≠ v[1] then
            if A[i, j] ≤ -1 then
                for k from 1 to n do
                    if A[i, k] ≥ 1 then
                        if k = v[4] then
                            loop := loop + 1 :
                            break:
                        elif k ≠ v[5] and visited[k] = 0 then
                            queue[enqueue](Q, k) :
                            visited[k] := 1 :
                        end if:
                    end if:
                end do:
                if loop ≥ 1 then
                    break:
                end if:
            end if:
        end if:
    end do:
    if loop ≥ 1 then
        queue[clear](Q) :
        break:
    end if:
```

```
    end do:

    queue[clear](Q) :
    visited := Vector(n) :
    if loop ≥ 1 then
      exist := 1 :
      for j from 1 to n do
        if j = v[5] then
          switch := switch + 1 :
          exist := 2 :
          return(exist) :
        elif A[v[1], j] ≥ 1 and visited[j] = 0 then
          queue[enqueue](Q, j) :
          visited[j] := 1 :
        end if:
      end do:

      while not queue[empty](Q) do
        j := queue[dequeue](Q) :
        for i from 1 to m do
          if i ≠ v[1] and i ≠ v[2] then
            if A[i, j] ≤ -1 then
              for k from 1 to n do
                if A[i, k] ≥ 1 then
                  if k = v[5] then
                    switch := switch + 1 :
                    exist := 2 :
                    return(exist) :
                  elif k ≠ v[4] and visited[k] = 0 then
                    queue[enqueue](Q, k) :
                    visited[k] := 1 :
                  end if:
                end if:
              end do:
            end if:
          end if:
        end do:
      end do:
    end if:

    return(exist) :
  end proc:
```

The function to check if the stoichiomatric matrix is mass conserved.

First check the passed matrix $A_{m\times n}$ (must be in a consistent form)

```
> ismassconserved_original := proc(A)
    local R, N, NS, m, n, absAdd, Add, x, y, z, e, i, nsAdd, nsAbsAdd, a, b, c :
```

```
    m := 0 :
    n := Dimension(A)[2] :
    absAdd := AddAlongDimension(|A|, 1) :
    z := Search(0, absAdd) :
    if z = 0 then
      Add := AddAlongDimension(A, 1) :
      x := Search(0, VectorAdd(absAdd, Add, 1, -1)) :
      if x = 0 then
        y := Search(0, VectorAdd(absAdd, Add, 1, 1)) :
        if y = 0 then
          N := NullSpace(A) :
          e := numelems(N) :
          if e > 0 then
            NS := Matrix(e, n) :
            for i from 1 to e by 1 do
              NS[i] := N[i] :
            end do:
            nsAbsAdd := AddAlongDimension(|NS|, 1) :
            a := Search(0, nsAbsAdd) :
            if a = 0 then
              nsAdd := AddAlongDimension(NS, 1) :
              b := Search(0, VectorAdd(nsAbsAdd, nsAdd, 1, 1)) :
              if b = 0 then
                m := 1 :
              end if:
            end if:
          end if:
        end if:
      end if:
    end if:
    return(m) :
  end proc:
```

The function to check if the stoichiomatric matrix is mass conserved.

First check the passed matrix $A_{m\times n}$ (must be in a consistent form)

```
> ismassconserved := proc(A)
    local R, N, NS, m, n, absAdd, Add, x, y, z, e, i, nsAdd, nsAbsAdd, a, b, c :
    m := 0 :
    n := Dimension(A)[2] :
    absAdd := AddAlongDimension(|A|, 1) :
    z := Search(0, absAdd) :
    if z = 0 then
      Add := AddAlongDimension(A, 1) :
      x := Search(0, VectorAdd(absAdd, Add, 1, -1)) :
      if x = 0 then
        y := Search(0, VectorAdd(absAdd, Add, 1, 1)) :
        if y = 0 then
          N := NullSpace(A) :
```

```
            e := numelems(N) :
            if e > 0 then
               NS := Matrix(e, n) :
               for i from 1 to e by 1 do
                  NS[i] := N[i] :
               end do:
               nsAbsAdd := AddAlongDimension(|NS|, 1) :
               a := Search(0, nsAbsAdd) :
               if a = 0 then
                  nsAdd := AddAlongDimension(NS, 1) :
                  b := Search(0, VectorAdd(nsAbsAdd, nsAdd, 1, 1)) :
                  if b = 0 then
                     m := 1 :
                     # here implement sufficient check of mass conservation

                  end if:
               end if:
            end if:
         end if:
      end if:
      return(m) :
   end proc:
```

To get the indecies of vectors in a Matrix

```
> indicesInMatrix := proc(B, A)
      ## return the indices of vectors from B in Matrix A
      local n, m, p, q, Ind, i, j :
      p := Dimension(A)[1] :
      q := Dimension(A)[2] :
      m := Dimension(B)[1] :
      n := Dimension(B)[2] :

      if n ≠ q then
         error "ERROR: The two matrices have the different column number."
      end if:

      Ind := Vector(m) :
      for i from 1 to m by 1 do
         for j from 1 to p by 1 do
            if IsEqual(B[i], A[j]) then
               Ind[i] := j :
               break:
            end if:
         end do:
         if Ind[i] = 0 then
```

```
            error "ERROR: no index found for ith vector from B in A"
          end if:
        end do:

        return(Ind) :
    end proc:
```

Construct and examine the properties of all stoichiomatric matrices.

```
> constrM := proc(n, m)
    local R, tA, A, iA, Z, r, g, h, i, j, k, l, total, right, mc, V, R2, r2, injective, injective0,
     injective1, injectiveEx, fileName, matrixData, interV, comp, injectiveEx0,
     injectiveEx1, injectiveEx2, injectiveEx3, s, selected, pfloops, unique, pfintersect,
     pfcount, f, myset, Ind :
    r := binomial(n, 2)·2 + binomial(n, 3)·6 :

    A := Matrix(m, n) :

    R := listRs(n) :
    R2 := listR2(n) :
    Ind := indicesInMatrix(R2, R) :

    if n = 4  then  r2 := 29·2 : end if:
    if n = 5  then  r2 := 43·2 : end if:
    if n ≥ 6  then  r2 := 46·2 : end if:
    if n < 4 then error "ERROR: n is smaller than 4" end if:

    total := r2/2 · binomial(r, m − 2) :

    # here we use some variable to count how many reactions are correct.
    right := 0 : injective0 := 0 : injective1 := 0 :
    #injectiveEx0:= 0:injectiveEx1:= 0:injectiveEx2:= 0: injectiveEx3:= 0:
    for l from 1 to r2 − 1 by 2 do

      myset := {seq(i, i = 1 ..m − 2)} :

      A[1] := R2[l] :
      A[2] := R2[l + 1] :
      while myset ≠ FAIL do
        if Ind[l] in myset or Ind[l + 1] in myset then
          myset := nextcomb(myset, r) :
        else
          A[3 ..m] := R[[op(myset)]] :
          mc := ismassconserved(A) :
```

```
        if mc = 1 then
          right := right + 1 :

          # check the existence of compeition and competition loop.
          comp := existcompetitionloop(A) :

          tA := Transpose(A) :

          # check the existence of intersecting positive feedback loops
          Z := findZ(tA) : s := Rank(tA) : selected := findloops(tA, Z) :
          pfloops := numelems(selected) :
          pfintersect := 0 :
          if pfloops ≥ 2 then
            pfcount := 0 :
            for f from 1 to numelems(selected) by 1 do
              pfcount := pfcount + numelems(selected[f]) :
            end do:
            unique := [ ]:
            for f from 1 to numelems(selected) by 1 do
              unique := [op(unique), op(selected[f]) ] :
            end do:
            if pfcount > numelems(MakeUnique(unique)) then
              pfintersect := 1 :
            end if:
          end if:

          iA := Transpose(A) :
          # simple injectivity check
          injective := isinjective(iA) :
          if injective = 0 then
            injective0 := injective0 + 1 :
            injectiveEx := isinjectiveextended(iA) :
            if injectiveEx = 1 or injectiveEx = 3 then
              if comp = 3 then
                if pfintersect = 1 then
                  fileName
 := sprintf(
"%1dspecies/nonmultistationary/competitionloop_intersectingloops/injectiveEx%1d_\
%d.csv", n, injectiveEx, right) :
                  ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                else
                  fileName
 := sprintf(
"%1dspecies/nonmultistationary/competitionloop_nointersectingloops/injectiveEx%1\
d_%d.csv", n, injectiveEx, right) :
                  ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
```

```
                end if:
            else
                if pfintersect = 1 then
                    fileName
 := sprintf(
"%1dspecies/nonmultistationary/nocompetitionloop_intersectingloops/injectiveEx%1\
d_%d.csv", n, injectiveEx, right) :
                    ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                else
                    fileName
 := sprintf(
"%1dspecies/nonmultistationary/nocompetitionloop_nointersectingloops/injectiveEx\
%1d_%d.csv", n, injectiveEx, right) :
                    ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                end if:
            end if:
        elif injectiveEx = 0 or injectiveEx = 2 then
            if comp = 3 then
                if pfintersect = 1 then
                    fileName
 := sprintf(
"%1dspecies/multistationary/competitionloop_intersectingloops/injectiveEx%1d_%d.
csv", n, injectiveEx, right) :
                    ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                else
                    fileName
 := sprintf(
"%1dspecies/multistationary/competitionloop_nointersectingloops/injectiveEx%1d_%\
d.csv", n, injectiveEx, right) :
                    ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                end if:
            else
                if pfintersect = 1 then
                    fileName
 := sprintf(
"%1dspecies/multistationary/nocompetitionloop_intersectingloops/injectiveEx%1d_%\
d.csv", n, injectiveEx, right) :
                    ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                else
                    fileName
 := sprintf(
"%1dspecies/multistationary/nocompetitionloop_nointersectingloops/injectiveEx%1d\
_%d.csv", n, injectiveEx, right) :
```

```
                    ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                  end if:
                end if:
              else
                error "ERROR: injectivity extended of A is not any of 0 to 3."
              end if:
            elif injective = 1 then
              injective1 := injective1 + 1 :
              if comp = 3 then
                if pfintersect = 1 then
                  fileName
 := sprintf(
"%1dspecies/nonmultistationary/competitionloop_intersectingloops/injective%1d_%d\
.csv", n, injective, right) :
                  ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                else
                  fileName
 := sprintf(
"%1dspecies/nonmultistationary/competitionloop_nointersectingloops/injective%1d_\
%d.csv", n, injective, right) :
                  ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                end if:
              else
                if pfintersect = 1 then
                  fileName
 := sprintf(
"%1dspecies/nonmultistationary/nocompetitionloop_intersectingloops/injective%1d_\
%d.csv", n, injective, right) :
                  ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                else
                  fileName
 := sprintf(
"%1dspecies/nonmultistationary/nocompetitionloop_nointersectingloops/injective%1\
d_%d.csv", n, injective, right) :
                  ExportMatrix(fileName, iA, target = csv, format = rectangular,
mode = ascii) :
                end if:
              end if:
            else
              error "ERROR: the injectivity of iA is neither 0 nor 1."
            end if:
          end if:
          myset := nextcomb(myset, r) :
        end if:
```

```
        end do:
    end do:

    V := [injective0, injective1, right, total, r, r2] :
    return(V) :
end proc:
>
```

```
> V := constrM(5, 5) # just count right matrices ~ 2360s (40 mins) (9.29s to 2359.41s)
                    V := [0, 0, 0, 0, 0, 0, 9229, 3532880, 80, 86]                    (1)
>
> V := constrM(5, 5) # also count injective extended matrices and export bistable matrices ~ 2716s
                                                  #(45 mins) (4621.73 s to 7337.76 s)
                    V := [65, 578, 87, 3977, 4707, 4522, 9229, 3532880, 80, 86]       (2)
> V := constrM(5, 5)
    # both check injectivity extended and competition loop ~2763.17s (46 mins)  (15s to 2778.17s)
                    V := [65, 578, 87, 3977, 4707, 4522, 9229, 3532880, 80, 86]       (3)
> V := constrM(5, 5) # check competition loop and intersecting loops ~ (36s to )
                    V := [4707, 4522, 9229, 3532880, 80, 86]                          (4)
> V := constrM(5, 5) # new algorithm to exclude potentially duplicated reactions in matrix
                    V := [4593, 4340, 8933, 3532880, 80, 86]                          (5)
```

▶ Testing