Feedback loop and self-loop detection in an ODE

Contents

1 Introduction

A biological system can be mathematically represented as a system of ordinary differential equations (ODEs). The Jacobian matrix of a system of ODEs is the matrix of the partial derivatives. The Jacobian matrix can be represented as a graph where each species is a node and an entry unequal to zero in the matrix is an edge in the graph. With directed path detection in the graph it is possible to determine if a species is affecting itself either directly (self-loop) or via other species (loop). The loop detection package can calculate self-loops and loops and shows if these are positive or negative. The output is a data frame which can easily be filtered for different parameters (e.g. loop length, loops containing a certain edge). The packages rootSolve, deSolve and igraph have to be installed to use loop detection.

1.1 Calculate Jacobian matrix

The following code shows how to calculate the Jacobian matrix of a user-defined ODE with the package *rootSolve*. This ODE has to be in the same format as required by the *deSolve* package. With the Jacobian it is possible to use the loop detection package to calculate self-loops and loops for the ODE system.

1.2 Read in the ODE

A text file with the ODE (here: a chain model with negative feedback, example given in "folder") will be read in with the function *source*. The parameters and states will be handed over as vectors. Subsequently it is possible to calculate the Jacobian matrix with the function *jacobian.full* from the *rootSolve* package.

```
source("NEGm4_func.txt")
param <- c(klin1=1,klin2=3,klin3=0.1,klin4=1,klin5=1,klin6=2,klin7=1,klin8=3,knonlin1=1,knonlin2=1)
state <- c(x1=1,x2=0.1,x3=1,x4=1)
jacobian <- jacobian.full(y=state, func=NEGm4, parms = param)
```

2 Find all loops

To find all loops for the given Jacobian matrix the function *get_all_loops* creates a directed graph from the matrix with the function *graph_from_adjacency_matrix* from the *igraph* package. To find all paths in the graph, the function iterates over all nodes and uses *all_simple_path* from *igraph*.

```
result <- get_all_loops(jacobian=jacobian)
result
```

```
##            loop length sign
## 1          1, 1      1   -1
## 2          2, 2      1   -1
## 3          3, 3      1   -1
## 4          4, 4      1   -1
## 5    3, 4, 2, 3      3   -1
## 6 3, 4, 1, 2, 3      4    1
```

The output is a data frame with three columns: loop, length and sign. So each row in the data frame shows the loop, the respective length and sign (1=positive loop, -1=negative loop). In the example ODE we found six loops. For all four nodes we found negative self loops. The fifth one is a path with the nodes and edges 3-> 4-> 2-> 3, which is negative and has length three. The last one is a path with the nodes and edges 3-> 4-> 1-> 2-> 3, which is positive and has length four.

3 Handle huge amount of loops

In the function *get_all_loops* it is also possible to restrict the amount of returned loops in case a system has so many loops that the function cannot handle the amount anymore or the function would take too much time. For this, the user can use the optional argument *num_loops*. In every iteration the function will check if the number of loops is still below the value in *num_loops*. The default value of *num_loops* is one million.

```
result <- get_all_loops(jacobian=jacobian, num_loops=100)
```

There will be a warning if the number of loops found is greater than the value of *num_loops*. In this case there will only be a part of all loops stored in *result*. It is possible that the function reports more loops than specified in *num_loops*. This is, because the current iteration will be finished even if the number of loops is above the given value. The warning will report how many species, starting from the first row and column of the Jacobian, were analysed until the function interrupted.

4 Helpful functions to get an overview about the results

Within the data frame it is easy to sort (e.g. for length). The following example shows how to sort for length in descending order.

```
result[order(-result$length),]
```

```
##            loop length sign
## 6 3, 4, 1, 2, 3      4    1
## 5    3, 4, 2, 3      3   -1
## 1          1, 1      1   -1
## 2          2, 2      1   -1
## 3          3, 3      1   -1
## 4          4, 4      1   -1
```

With the *table* function it is also possible to get an overview on how many negative and positive loops or how many loops with a certain length were found. Here another system (dyk92) with 189 loops is used to illustrate the different possible function.

```
## for sign
table(result$sign)
```

```
##  -1   1
##  75 114
```

```
## for length
table(result$length)
```

```
##  1  2  3  4  5  6  7  8  9
##  9 13  3 18 12 50 30 36 18
```

In this model we have 75 negative and 114 positive loops and most of the loops (50) have a length of six.

5 Find loops with a certain edge

The function *get_edges* can find certain edges in a list of loops. The following example shows how to filter for loops with an edge from node 1 to node 8.

```
edges_isin <- get_edges(loop_list=result$loop, from=1, to=8)
result[edges_isin,]
```

```
##                             loop length sign
##    44        1, 8, 5, 7, 3, 6, 1      6    1
##    45  1, 8, 5, 7, 3, 2, 4, 6, 1      8    1
##    47        1, 8, 5, 2, 4, 6, 1      6    1
##    51        1, 8, 5, 2, 3, 6, 1      6    1
##    62              1, 8, 4, 6, 1      4    1
##    63  1, 8, 4, 2, 5, 7, 3, 6, 1      8    1
##    68        1, 8, 4, 2, 3, 6, 1      6    1
##    110 8, 9, 7, 5, 2, 4, 6, 1, 8      8    1
##    114 8, 9, 7, 5, 2, 3, 6, 1, 8      8    1
##    120       8, 9, 7, 3, 6, 1, 8      6    1
##    133 8, 9, 7, 3, 2, 4, 6, 1, 8      8    1
##    152       7, 9, 6, 1, 8, 5, 7      6    1
##    153 7, 9, 6, 1, 8, 5, 2, 3, 7      8    1
##    154 7, 9, 6, 1, 8, 4, 2, 5, 7      8    1
##    155 7, 9, 6, 1, 8, 4, 2, 3, 7      8    1
##    156             8, 9, 6, 1, 8      4    1
```

The function *get_edges* returns a Boolean vector with TRUE if the edge is present in the loop and FALSE if not. In the example we can find 16 loops with an edge from node 1 to node 8.

6 Evaluate a system over time

With the function *ode* from the *deSolve* package, we can evaluate a system over time and then have a look at different timesteps to determine the loops at a particular timestep. Here we use again the system dYK92 and assume that we have the states and parameters already in our storage.

```
## generate a timeframe
times <- seq(0, 1000, by=1)
## calculate the solution for the different timesteps
sol <- ode(y=state, times=times, func=dYK92, parms=param)
head(sol)
```

```
##      time        x1         x2        x3        x4         x5        x6
## [1,]    0 0.1768300 0.04797300 0.1845100 0.0993930 0.06082600 0.3822800
## [2,]    1 0.1767378 0.04733262 0.1806270 0.1427959 0.05664778 0.3521990
## [3,]    2 0.1648230 0.05106629 0.1948228 0.1480367 0.05536406 0.3467079
## [4,]    3 0.1594947 0.05305210 0.2024325 0.1485769 0.05479298 0.3443250
## [5,]    4 0.1573974 0.05388657 0.2056359 0.1485834 0.05455027 0.3433554
## [6,]    5 0.1565927 0.05421487 0.2068969 0.1485588 0.05445324 0.3429766
##              x7        x8         x9
## [1,] 0.03223700 0.1260000 0.06677900
## [2,] 0.03003271 0.1244000 0.06596298
## [3,] 0.02935244 0.1141186 0.06052918
## [4,] 0.02904985 0.1096162 0.05815243
## [5,] 0.02892125 0.1078466 0.05721857
## [6,] 0.02886984 0.1071675 0.05686023
```

```
## plot the concentration over time for each species
plot(sol, xlab="time", ylab="-")

## assuming we want to have a look at the loops in the last timestep (here 1000)
timestep <- 1000

## define the states (we start here with out[timestep,2], because the first column is the time column)
state <- c(sol[timestep,2:10])

## now we want to use the function get_all_loops to get the loop list at timestep 1000
jacobian_t1000 <- jacobian.full(y=state, func=dYK92, parms=param)
result_t1000 <- get_all_loops(jacobian_t1000)
```

7 Compare lists of loops

If we want to compare two lists of loops, e.g. the list of loops which were calculated with the given states (here: result$loops) and the list of loops which were calculated with the states at timestep 1000 (here: result_t1000$loops) we can use the function *%in%*. To guarantee correct results, it is necessary to sort each loop in each list such that each loop starts with the smallest node index. To do so, we can apply the function *sort_loops* to each of the two lists and then use the function *%in%*.

```
## example of one loop before applying the sort function
result$loop[20]
```

```
## [[1]]
## [1] 4 6 1 4
```

```
## apply the sort function
sorted_result <- sort_loops(loop_list=result$loop)
sorted_result_t1000 <- sort_loops(loop_list=result_t1000$loop)
```

```
## example of the same loop after applying the sort function
sorted_result[20]
```

```
## [[1]]
## [1] 1 4 6 1
```

```
## compare both lists
compare_vec <- sorted_result %in% sorted_result_t1000
head(compare_vec)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

The result is a Boolean vector where the entry is TRUE if we could find the loop of the first list result$loop anywhere in the second list result_t1000$loop and FALSE otherwise. If the lists have not the same length, we have to do this also vice versa. We can also use the function *all* to get one TRUE if all entries are present in both lists or FALSE if not.

```
all(sorted_result %in% sorted_result_t1000)
```

```
## [1] TRUE
```