1 Inputs and Outputs for popModel()

1.1 The input arguments to popModel()

‘`numSpecies`’ (integer) is the number of different species to be modelled.

‘`numStages`’ is a vector containing the number of life stages in each species, e.g. ‘`numStages=c(2,1)`’ if there are two species - one with two stages and another with one.

‘`numStrains`’ is a vector containing the number of strains in each species, e.g. ‘`numStrains=c(5,10)`’ if there are two species - one with five strains and another with ten.

‘`timeVec`’ is a vector describing the output times for the solution e.g. ‘`seq(0,10,0.5)`’.

‘`speciesNames`’ This is a vector of the strings containing the species’ names e.g. ‘`speciesNames=c(‘host’,‘egg parasitoid’,‘larval parasitoid’)`’ These can also be used in the user-defined rate functions to index the variables in the input argument ‘x’ (see Section 1.1.1) and will be used to name the columns of the output matrix from **popModel()**.

‘`stageNames`’ This is a list containing vectors of the stage names (strings) for each species. E.g. for 2 species with one stage and three stages respectively this could have the form
‘`stageNames=list(‘adults’,c(‘eggs’,‘juvs’,‘adults’))`’.

‘`rateFunctions`’ is a list containing the user-defined rate functions – see Section 1.1.1 for details.

‘`ICs`’ is a list of matrices containing the initial conditions for every stage and strain of each species. Due to the model formulation these are very restrictive and must be zero for all stages apart from the reproductive stage (usually the last stage). Thus it is recommended that the initial conditions are defined through immigration rates in the **immigrationFunc** (see section 1.1.1). However, the ICs can be set as follows: Each species has a matrix with the number of columns equal to the number of strains in that species and the number of rows equal to the number of stages in that species. E.g. for 2 species, the first with 2 strains and 3 stages, the second with 1 strain and 1 stage, then for zero starting conditions: ‘`ICs=list(matrix(0,ncol=2,nrow=3),matrix(0,ncol=1,nrow=1))`’.

‘`timeDependLoss`’ (optional; default is rep(TRUE,numSpecies)) is a vector containing TRUE/FALSE for each species. It is TRUE if the per capita loss rate varies with time for any stage of the species (e.g. TRUE for density dependent death and/or density dependent emigration) and FALSE otherwise. E.g. ‘`timeDependLoss=c(TRUE, FALSE)`’ if there is no emigration and the first species has time dependent per capita death rates but the second does not.

‘`timeDependDuration`’ (optional; default is rep(FALSE,numSpecies)) is a vector containing TRUE/FALSE for each species. It is TRUE if the stage duration varies with time for any stage of the species and FALSE otherwise. E.g. ‘`timeDependDuration=c(TRUE, FALSE)`’ if the first species has any time dependent stage durations but the second does not.

‘`solverOptions`’ (optional; default is ‘`list(DDEsolver=‘PBS’, tol=1e-7, hbsize=1e3, method=‘lsoda’, atol=1e-7, dt=0.1))`’ is a list of instructions for the DDE solver, containing: ‘DDEsolver’, ‘tol’, ’hbsize’, ‘method’ and ‘atol’. ‘DDEsolver’ must be either ‘deSolve’ or ‘PBS’ (these are the R packages used to solve the DDEs). The ‘tol’ option sets the relative tolerances and ‘hbsize’ sets the size of the history buffer. The remaining two items, ‘method’ and ‘atol’ set the numerical integration scheme and the absolute tolerance if DDEsolver=‘deSolve’ (PBS does not have these options).

‘`checkForNegs`’ (optional; default is TRUE) is TRUE if you would like to check your solution for negative values using the function **checkSolution()**. Note **checkSolution()** can also be called separately using ‘`checkSolution(output,numSpecies,numStages,numStrains,ntol)`’.

‘`ntol`’ (optional; default is 0.01) is the tolerance on the magnitude of the negative values detected by **checkSolution()** . For example if ntol=0.01 then a warning is triggered if a stage or strain of any species falls below less than minus 1% of its maximum value at any time. If ntol is zero then any values below zero will trigger a warning even if their magnitude is insignificant (eg 10^{-30}).

‘`plotFigs`’ (optional; defalt is TRUE) is TRUE if you would like a basic plot of the solution. The function that is called to do this is called **genericPlot()** and can also be called separately using
‘`genericPlot(modelOutput, numSpecies, numStages, varNames, speciesNames, stageNames, saveFig, figType, figName)`’
To create more sophisticated plots the user is recommended to use the results in the matrix generated from **popModel()** with their own plotting scripts.

‘`saveFig`’(optional; default is FALSE.) To save the figure generated by **popModel()** make this TRUE.

‘`figType`’(optional; default is ‘eps’.) Format for the saved figure from **popModel()**. This can be ‘eps’, ‘pdf’, ‘png’ or ‘tiff’.

‘`figName`’(optional; default name is ‘stagePopFig’ and it will be saved in your working directory). A string containing the filepath for where the figure file should be saved.

‘`sumOverStrains`’ (optional; default is TRUE). If there is more than one strain in any species then the results for each strain are given in the model output. If you are only interested in the results for the species as a whole then change this to FALSE to simplify the model output.

‘`plotStrainsFig`’ (optional; default is TRUE (if max(numStrains)>1). This will produce plots for each individual strain.

‘`saveStrainsFig`’ (optional; default is FALSE). Change to TRUE to save the plot.

‘`strainsFigType`’ (optional; default is ‘eps’). Format for the saved figure. This can be ‘eps’, ‘pdf’, ‘png’ or ‘tiff’.

‘`strainsFigName`’ (optional; default is ‘strainFig’ plus the species name). A string containing the filepath describing where the strain plots should be saved.

1.1.1 The Rate Functions (user-defined)

The user must define a list containing all the functions named below. These must have the input arguments specified below in order for `stagePop` to run (however, these arguments do not necessarily need to be used within the function). The output from each of them must be a single value which equals the rate for the stage, species and time given in the input arguments. The input arguments to these functions are all single values apart from ‘`x`’ which is a vector of all the state variables at the input time. This is included to allow the user to specify density-dependent rates and can be indexed using the names specified in the **popModel()** arguments ‘`speciesNames`’ and ‘`stageNames`’ using x$speciesName[‘stage’,strain]. For more details see Appendix 3 and/or use the R help function for each of the functions below.

reproFunc(x,time,species,strain) The output from this is a value for the rate at which new organisms enter the first stage through reproduction of the species in the input argument at the given time. The units are: organisms time unit^{-1}. Note, unlike the other rate functions, this does not have the ‘stage’ input argument (this is because new organisms produced by reproduction are only allowed to enter the first stage).

deathFunc(stage,x,time,species,strain) The output from this is the per capita death rate of organisms in the stage and species given in the input arguments at the given time. The units are: time unit^{-1}.

durationFunc(stage,x,time,species,strain) The output from this is the length of the stage duration (in time units) for the stage and species given in the input arguments. Note that if the stage durations vary in time then this function will only be used to compute the initial values of the stage durations; for future values the user must define **develFunc**.

develFunc(stage,x,time,species,strain) The output from this is the rate of development rate of the stage and species given in the input arguments at the given time. The units are: time unit^{-1}. Note that if the stage duration (and hence development rate) is constant in time then there is no need to define this function e.g. if timeDependDuration is FALSE for all species. The output from **develFunc** must always be strictly greater than zero.

immigrationFunc(stage,x,time,species,strain) The output from this is the rate of immigration (individuals per unit time) into the stage specified in the input arguments for the input species at the given time. Since the initial conditions for `stagePop` are quite restrictive, the user will generally start the simulation by specifying immigration into a given stage over a short interval at the start of the simulation. Note that if these rates are set very high the DDE solvers may fail so the user should endeavour to enter realistic (or at least low) values.

emigrationFunc(stage,x,time,species,strain) The output from this is the per capita emigration rate of individuals in the stage and species given in the input arguments. The units are: time unit^{-1}.

1.2 Ouput from popModel()

The output from **popModel()** is a matrix which contains the values of the state variables, the probabilities of survival, the durations of each stage (if time-varying) and the rates of change of each state variable at the times specified in the input time vector (‘`timeVec`’). Each column of the output matrix is named according to the **popModel()** input option ‘`variableNames`’:

‘`time`’ is the time point t at which the solution is output (specified by input argument ‘`timeVec`’).

‘`speciesName[i].StageNames[[i]][j]`’ is the density (or number) of stage j of species i at time t. For multiple strains, for strain k in species i, stage j, this is ‘`speciesName[i].StageNames[[i]][j].strain[k]`’

‘`prob.speciesName[i].StageNames[[i]][j]`’ is the probability that stage j of species i will survive the length of the stage duration at time t. Note this is only present if species i has time dependent per capita death rates or has time dependent stage durations.

‘`dur.speciesName[i].StageNames[[i]][j]`’ is the duration of stage j for species i at time t. Note this is only present if the stage durations for species i are time dependent.

‘`dot.speciesName[i].StageNames[[i]][j]`’ is the rate of change of the density (or number) of stage j of species i at time t.

2 Assumptions made in `stagePop`

In `stagePop` the following assumptions are hard-coded

- The birth (via reproduction) of new individuals is always into the first life stage.
- The simulation time always begins at zero.
- For all simulation times less than zero:
  - there is no reproduction or immigration,
  - the number or density of organisms in each stage is equal to those at time zero,
  - the death rate is equal to that at time zero,
  - the development function is equal to that at time zero.

3 Checking the solution from `stagePop` is accurate

As with any numerical integration the results from `stagePop` are subject to error. Ideally, the user should check the results against an analytical solution (e.g. at equilibrium conditions). However, since this is frequently not possible, the simplest way to check for inaccuracies is to use the **checkSolution()** function (this is called if the **popModel()** input argument '`checkForNegs`' is TRUE (default)). This will find negative values in your solution within a limit specified by '`ntol`'. If negative values do occur then the user can try reducing the size of the tolerances in 'solverOptions' to improve the numerical accuracy. For further confirmation of the solution this can be repeated with both DDE solvers (i.e. deSolve and PBS). The user should be aware that the size of the tolerances needed may differ vastly between modelling projects and DDE solvers. As the tolerance size decreases the run time for `stagePop` will increase, thus if CPU time is important the user is recommended to find the largest tolerance size required for an accurate solution. To do this we recommend repeating the simulation with increasingly smaller tolerances until the solution no longer changes.

4 Trouble Shooting

If error messages or warnings appear in the console window that mention the integration step size this generally means that the DDE solver can not solve the problem without making the step size too small to compute (generally because the problem is stiff). The default solver method for deSolve is '`lsoda`' which is designed to deal with stiff and non-stiff problems but the user can also try different methods via the '`method`' option. If time lags are long then errors may occur saying the lag history is not long enough (or that the lag for a variable is too long). To deal with this the history buffer size of the solver can be changed in using '`hbsize`'. If both the DDE solvers (deSolve and PBS) fail to complete the integration it is likely that the user has incorrectly specified the problem or has generated extremely high rates in the user-defined rate functions. In this case it is recommended that the rate functions be carefully checked for internal consistency and/or the problem be non-dimensionalised or simply more appropriately scaled. If an error says 'The number of derivatives returned by func() must equal the length of the initial conditions vector' then the vector of initial conditions is incorrect (e.g. perhaps an entry is missing).