Simulate tumor progression for one or more individuals, optionally returning just a sample in time.

Description

Simulate tumor progression including possible restrictions in the order of driver mutations. Optionally add passenger mutations. When used in frequency dependent fitness situation, only fitness effects are allowed. Simulation is done using the BNB algorithm of Mather et al., 2012.

Usage

```
 oncoSimulIndiv(fp, model = "Exp",
                numPassengers = 0, mu = 1e-6, muEF = NULL,
                detectionSize = 1e8, detectionDrivers = 4,
                detectionProb = NA,
                sampleEvery = ifelse(model %in% c("Bozic", "Exp"), 1,
                             0.025),
                initSize = 500, s = 0.1, sh = -1,
                K = initSize/(exp(1) - 1), keepEvery = sampleEvery,
                minDetectDrvCloneSz = "auto",
                extraTime = 0,
                finalTime = 0.25 * 25 * 365, onlyCancer = TRUE,
                keepPhylog = FALSE,
                mutationPropGrowth = ifelse(model == "Bozic",
                                                 FALSE, TRUE),
                max.memory = 2000, max.wall.time = 200,
                max.num.tries = 500,
                errorHitWallTime = TRUE,
                errorHitMaxTries = TRUE,
                verbosity = 0,
                initMutant = NULL,
                AND_DrvProbExit = FALSE,
                fixation = NULL,
                seed = NULL)

oncoSimulPop(Nindiv, fp, model = "Exp", numPassengers = 0, mu = 1e-6,
                muEF = NULL,
                detectionSize = 1e8, detectionDrivers = 4,
                detectionProb = NA,
                sampleEvery = ifelse(model %in% c("Bozic", "Exp"), 1,
                             0.025),
                initSize = 500, s = 0.1, sh = -1,
                K = initSize/(exp(1) - 1), keepEvery = sampleEvery,
                minDetectDrvCloneSz = "auto",
                extraTime = 0,
                finalTime = 0.25 * 25 * 365, onlyCancer = TRUE,
                keepPhylog = FALSE,
                mutationPropGrowth = ifelse(model == "Bozic",
                                                 FALSE, TRUE),
                max.memory = 2000, max.wall.time = 200,
                max.num.tries = 500,
                errorHitWallTime = TRUE,
                errorHitMaxTries = TRUE,
                initMutant = NULL,
                AND_DrvProbExit = FALSE,
                fixation = NULL,
                verbosity = 0,
                mc.cores = detectCores(),
                seed = "auto")


oncoSimulSample(Nindiv,
                fp,
                model = "Exp",
                numPassengers = 0,
                mu = 1e-6,
                muEF = NULL,
                detectionSize = round(runif(Nindiv, 1e5, 1e8)),

                detectionDrivers = {
                                if(inherits(fp, "fitnessEffects")) {
                                    if(length(fp$drv)) {
                                        nd <- (2: round(0.75 * length(fp$drv)))
                                    } else {
                                        nd <- 9e6
                                    }
                                } else {
                                    nd <- (2 : round(0.75 * max(fp)))
                                }
                                if (length(nd) == 1)
                                    nd <- c(nd, nd)
                                sample(nd, Nindiv,
                                       replace = TRUE)
                            },
                detectionProb = NA,
                sampleEvery = ifelse(model %in% c("Bozic", "Exp"), 1,
                    0.025),
                initSize = 500,
                s = 0.1,
                sh = -1,
                K = initSize/(exp(1) - 1),
                minDetectDrvCloneSz = "auto",
                extraTime = 0,
                finalTime = 0.25 * 25 * 365,
                onlyCancer = TRUE, keepPhylog = FALSE,
                mutationPropGrowth = ifelse(model == "Bozic",
                                            FALSE, TRUE),
                max.memory = 2000,
                max.wall.time.total = 600,
                max.num.tries.total = 500 * Nindiv,
                typeSample = "whole",
                thresholdWhole = 0.5,
                initMutant = NULL,
                AND_DrvProbExit = FALSE,
                fixation = NULL,
                verbosity  = 1,
                showProgress = FALSE,
                seed = "auto")
```

Arguments

`Nindiv`	Number of individuals or number of different trajectories to simulate.
`fp`	Either a poset that specifies the order restrictions (see `poset` if you want to use the specification as in v.1. Otherwise, a fitnessEffects object (see `allFitnessEffects`). Always is a fitnessEffects object when you are in a frequency dependent fitness simulation and its presence is mandatory. Of course in this case fp$frequencyDependentFitness must be TRUE. Other arguments below (s, sh, numPassengers) make sense only if you use a poset, as they are included in the fitnessEffects object.
`model`	One of "Bozic", "Exp", "McFarlandLog" (the last one can be abbreviated to "McFL"). The default is "Exp".
`numPassengers`	This has no effect if you use the `allFitnessEffects` specification. This happends always when you are in a simulation that use frequency dependent fitness. If you use the specification of v.1., the number of passenger genes. Note that using v.1 the total number of genes (drivers plus passengers) must be smaller than 64. All driver genes should be included in the poset (even if they depend on no one and no one depends on them), and will be numbered from 1 to the total number of driver genes. Thus, passenger genes will be numbered from (number of driver genes + 1):(number of drivers + number of passengers).
`mu`	Mutation rate. Can be a single value or a named vector. If a single value, all genes will have the same mutation rate. If a named vector, the entries in the vector specify the gene-specific mutation rate. If you pass a vector, it must be named, and it must have entries for all the genes in the fitness specification. Passing a vector is only available when using fitnessEffects objects for fitness specification. See also `mutationPropGrowth`.
`muEF`	Mutator effects. A mutatorEffects object as obtained from `allMutatorEffects`. This specifies how mutations in certain genes change the mutation rate over all the genome. Therefore, this allows you to specify mutator phenotypes: models where mutation of one (or more) gene(s) leads to an increase in the mutation rate. This is only available for version 2 (and above) specifications. All the genes specified in `muEF` MUST be included in `fp`. If you want to have genes that have no direct effect on fitness, but that affect mutation rate, you MUST specify them in `fp`, for instance as `noIntGenes` with an effect of 0. If you use mutator effects you must also use fitnessEffects in `fp`. You are not allowed to use mutator effects object in a frequency depedent fitness simulation.
`detectionSize`	What is the minimal number of cells for cancer to be detected. For `oncoSimulSample` this can be a vector. If set to NA, `detectionSize` plays no role in stopping the simulations.
`detectionDrivers`	The minimal number of drivers (not modules, drivers, whether or not they are from the same module) present in any clone for cancer to be detected. For `oncoSimulSample` this can be a vector. For `oncoSimulSample`, if there are drivers (either because you are using a v.1 object or because you are using a fitnessEffects object with a `drvNames` component —see `allFitnessEffects`—) the default is a vector of drivers from a uniform between 2 and 0.75 the total number of drivers. If there are no drivers (because you are using a fitnessEffects object without a `drvNames`, either because you specified it explicitly or because all of the genes are in the noIntGenes component) the simulations should not stop based on the number of drivers (and, thus, the default is set to 9e6). This is the case when you run the simulation with frequency dependent fitness. If set to NA, `detectionDrivers` plays no role in stopping the simulations.
`detectionProb`	Vector of arguments for the mechanism where probability of detection depends on size. If `NA`, this mechanism is not used. If 'default', the vector will be populated with default values. Otherwise, a named vector with some of the following named elements (see 'Details'): • PDBaseline: Baseline size subtracted to total population size to compute the probability of detection. If not given explicitly, the default is `1.2 * initSize`. • p2: The probability of detection at population size n2. If you specificy p2 you must also specify n2 and you must not specify `cPDetect`. The fault is 0.1. • n2: The population size at which probability of detection is p2. The default is `2 * initSize`. • cPDetect: The change in probability of detection with size. If you specify it, you should not specify either of p2 or n2. See 'Details'. • checkSizePEvery: Time between successive checks for the probability of exiting as a function of population size. If not given explicitly, the default is 20. See 'Details'. If you only provide some of the elements (except for the pair p2, n2, where you must provide both if you provide any), the

	rest will be filled with default values. This option can not be used with v.1 objects.
`sampleEvery`	How often the whole population is sampled. This is not the same as the interval between successive samples that are kept or stored (for that, see keepEvery). For very fast growing clones, you might need to have a small value here to minimize possible numerical problems (such as huge increase in population size between two successive samples that can then lead to problems for random number generators). Likewise, for models with density dependence (such as McF) this value should be very small.
`initSize`	Initial population size.
`K`	Initial population equilibrium size in the McFarland models.
`keepEvery`	Time interval between successive whole population samples that are actually stored. This must be larger or equal to `sampleEvery`. If keepEvery is not a multiple integer of sampleEvery, the interval between successive samples that are stored will be the smallest multiple integer of `sampleEvery` that is larger than or equal to keepEvery. If you want nice plots, set `sampleEvery` and `keepEvery` to small values (say, 5 or 2). Otherwise, you can use a `sampleEvery` of 1 but a `keepEvery` of 15, so that the return objects are not huge and the code runs a lot faster. Setting `keepEvery = NA` means we only keep the very last sample. This is useful if you only care about the final state of the simulation, not its complete history.
`minDetectDrvCloneSz`	A value of 0 or larger than 0 (by default equal to `initSize` in the McFarland model). If larger than 0, when checking if we are done with a simulation, we verify that the sum of the population sizes of all clones that have a number of mutated drivers larger or equal to `detectionDrivers` is larger or equal to this `minDetectDrvCloneSz`. The reason for this parameter is to ensure that, say, a clone with a certain number of drivers that would cause the simulation to end has not just appeared and is present in only one individual that might then immediately go extinct. This can be relevant in secenarios such as the McFarland model. See also `extraTime`.
`extraTime`	A value larger than zero waits those many additional time periods before exiting after having reached the exit condition (population size, number of drivers). The reason for this setting is to prevent the McFL models from always exiting at a time when one clone is increasing its size quickly (see `minDetectDrvCloneSz`). By setting an `extraTime` larger than 0, we can sample at points when we are at the plateau.
`finalTime`	What is the maximum number of time units that the simulation can run. Set to NA to disable this limit.
`onlyCancer`	Return only simulations that reach cancer? If set to TRUE, only simulations that satisfy the `detectionDrivers` or the `detectionSize` requirements or that are "detected" because of the `detectionProb` mechanism will be returned: the simulation will be repeated, within the limits set by `max.num.tries` and `max.wall.time` (and, for oncoSimulSample also `max.num.tries.total` and `max.wall.time.total`), until one which meets the `detectionDrivers` or `detectionSize` or one which is detected stochastically under `detectionProb` is obtained. If `onlyCancer = FALSE` the simulation is returned regardless of final population size or number of drivers in any clone and this includes simulations where the population goes extinct.
`keepPhylog`	If TRUE, keep track of when and from which clone each clone is created. See also `plotClonePhylog`.
`mutationPropGrowth`	If TRUE, make mutation rate proportional to growth rate, so clones that grow faster also mutate faster. Thus, $mutation_rate = mu * birth_rate$. This is a simple way of approximating that mutation happens at cell division (it is not strictly making mutation happen at cell division, since mutation is not strictly coupled with division). Of course, this only makes sense in models where birth rate changes.
`initMutant`	For v.2: a string with the mutations of the initial mutant, if any. This is the same format as for `evalGenotype`. The default (if you pass nothing) is to start the simulation from the wildtype genotype with nothing mutated. For v.1 we no longer accept initMutant: it will be ignored.
`max.num.tries`	Only applies when `onlyCancer = TRUE`. What is the maximum number of times, for an individual simulation, we can repeat the simulation for it to reach cancer? There are certain parameter settings where reaching cancer is extremely unlikely and you might not want to run forever in those cases.
`max.num.tries.total`	Only applies when `onlyCancer = TRUE` and for `oncoSimulSample`. What is the maximum number of times, over all simulations for all individuals in a population sample, that we can repeat the simulations so that cancer is reached for all individuals? The idea is to set a limit on the average minimal probability of reaching cancer for a set of simulations to be accepted.
`max.wall.time`	Maximum wall time for the simulation of one individual (over all `max.num.tries`). If the simulation is not done in this time, it is aborted.
`max.wall.time.total`	Maximum wall time for all the simulations (when using `oncoSimulSample`), in seconds. If the simulation is not completed in this time, it is aborted. To prevent problems from a single individual simulation going wild, this limit is also enforced per simulation (so the run can be aborted directly from C++).
`errorHitMaxTries`	If TRUE (the default) a simulation that reaches the maximum number of repetitions allowed is considered not to have succesfully finished and, thus, an error, and no output from it will be reported. This is often what you want. See `Details`.
`errorHitWallTime`	If TRUE (the default) a simulation that reaches the maximum wall time is considered not to have succesfully finished and, thus, an error, and no output from it will be reported. This is often what you want. See `Details`.
`max.memory`	The largest size (in MB) of the matrix of Populations by Time. If it creating it would use more than this amount of memory, it is not created. This prevents you from accidentally passing parameters that will return an enormous object.
`verbosity`	If 0, run silently. Iincreasing values of verbosity provide progressively more information about intermediate steps, possible numerical notes/warnings from the C++ code, etc. Values less than 0 supress some default notes: use with care.
`typeSample`	"singleCell" (or "single") for single cell sampling, where the probability of sampling a cell (a clone) is directly proportional to its population size. "wholeTumor" (or "whole") for whole tumor sampling (i.e., this is similar to a biopsy being the entire tumor). See `samplePop`.
`thresholdWhole`	In whole tumor sampling, whether a gene is detected as mutated depends on thresholdWhole: a gene is considered

	mutated if it is altered in at least thresholdWhole proportion of the cells in that individual. See `samplePop`.
`mc.cores`	Number of cores to use when simulating more than one individual (i.e., when calling oncoSimulPop).
`showProgress`	If TRUE, provide information, during exection, of the individual done, and the number of attempts and time used.
`AND_DrvProbExit`	If TRUE, cancer will be considered to be reached if both the `detectionProb` mechanism and `detectionDrivers` are satisfied. This is and AND, not an OR condition. Using this option with `fixation` is not allowed (as it does not make much sense).
`fixation`	If non-NULL, a list or a vector, where each element of is a string with a gene or a gene combination or a genotype (see below). Simulations will stop as soon as any of the genes or gene combinations or genotypes are fixed (i.e., reach a minimal frequency). If you pass gene combinations or genotypes, separate genes with commas (not '>'); this means order is not (yet?) supported. This way of specifying gene combinations is the same as the one used for `initMutant` and `evalGenotype`. To differentiate between gene combinations and specific genotypes, genotypes are specified by prepending them with a "_,". For instance, `fixation = c("A", "B, C")` specifies stopping on any genotypes with those gene combinations. In contrast, `fixation = c("_,A", "_,B, C" )` specifies stopping only on gentoypes "A" or "B, C". See the vignette for further examples. In addition to the gene combinations or genotypes themeselves, you can add to the list or vector the named elements `fixation_tolerance`, `min_successive_fixation` and `fixation_min_size`. `fixation_tolerance`: fixation is considered to have happened if the genotype/gene combinations specified as genotypes/gene combinations for fixation have reached a frequency `> 1 - fixation_tolerance`. (The default is 0, so we ask for genotypes/gene combinations with a frequency of 1, which might not be what you want with large mutation rates and complex fitness landscape with genotypes of similar fitness.). `min_successive_fixation`: during how many successive sampling periods the conditions of fixation need to be fulfilled before declaring fixation. These must be successive sampling periods without interruptions (i.e., a single period when the condition is not fulfilled will set the counter to 0). This can help to exclude short, transitional, local maxima that are quickly replaced by other genotypes. (The default is 50, but this is probably too small for “real life” usage). `fixation_min_size`: you might only want to consider fixation to have happened if a minimal size has been reached (this can help weed out local maxima that have fitness that is barely above that of the wild-type genotype). (The default is 0). Using this option with `AND_DrvProbExit` is not allowed (as it does not make much sense). This option is not allowed either with the old v.1 specification.
`s`	Selection coefficient for drivers. Only relevant if using a poset as this is included in the fitnessEffects object. This will eventually be deprecated.
`sh`	Selection coefficient for drivers with restrictions not satisfied. A value of 0 means there are no penalties for a driver appearing in a clone when its restrictions are not satisfied. To specify "sh=Inf" (in Diaz-Uriarte, 2015) use sh = -1. Only relevant if using a poset as this is included in the fitnessEffects object. This will eventually be deprecated.
`seed`	The seed for the C++ PRNG. You can pass a value. If you set it to NULL, then a seed will be generated in R and passed to C++. If you set it to "auto", then if you are using v.1, the behavior is the same as if you set it to NULL (a seed will be generated in R and passed to C++) but if you are using v.2, a random seed will be produced in C++. If you need reproducibility, either pass a value or set it to NULL (setting it to NULL will make the C++ seed reproducible if you use the same seed in R via `set.seed`). However, even using the same value of seed is unlikely to give the exact same results between platforms and compilers. Moreover, note that the defaults for seed are not the same in `oncoSimulIndiv`, `oncoSimulPop` and `oncoSimulSample`. When using oncoSimulPop, if you want reproducibility, you might want to, in addition to setting `seed = NULL`, also do `RNGkind("L'Ecuyer-CMRG")` as we use `mclapply`; look at the vignette of **parallel**.

Details

The basic simulation algorithm implemented is the BNB one of Mather et al., 2012, where I have added modifications to fitness based on the restrictions in the order of mutations.

Full details about the algorithm are provided in Mather et al., 2012. The evolutionary models, including references, and the rest of the parameters are explained in Diaz-Uriarte, 2014, especially in the Supplementary Material. The model called "Bozic" is based on Bozic et al., 2010, and the model called "McFarland" in McFarland et al., 2013.

oncoSimulPop simply calls oncoSimulIndiv multiple times. When run on POSIX systems, it can use multiple cores (via mclapply).

The `summary` methods for these classes return some of the return values (see next) as a one-row (for class oncosimul) or multiple row (for class oncosimulpop) data frame. The `print` methods for these classes simply print the summary.

Changing options `errorHitMaxTries` and `errorHitWallTime` can be useful when conducting many simulations, as in the call to `oncoSimulPop`: setting them to TRUE means nothing is recorded for those simulations where ending conditions are not reached but setting them to FALSE would allow you to record the output; this would potentially result in a mixture where some simulations would not have reached the ending condition, but this might sometimes be what you want. Note, however, that `oncoSimulSample` always has both them to TRUE, as it could not be otherwise.

`GenotypesWDistinctOrderEff` provides the information about order effects that is missing from `Genotypes`. When there are order effects, the `Genotypes` matrix can contain genotypes that are not distinguishable. Suppose there are two genes, the first and the second. In the `Genotype` output you can get two columns where there is a 1 in both genes: those two columns correspond to the two possible orders (first gene mutated first, or first gene mutated after the second). `GenotypesWDistinctOrderEff` disambiguates this. The same is done by `GenotypesLabels`; this is easier to decode for a human (a string of gene labels) but a little bit harder to parse automatically. Note that when you use the default print method for this object, you get, among others, a two-column display with the `GenotypeLabels` information. When order matters, a genotype shown as “x > y _ z” means that a mutation in “x” happened before a mutation in “y”; there is also a mutation in “z” (which could have happened before or after either of “x” or “y”), but “z” is a gene for which order does not matter. When order does not matter, a comma "," separates the identifiers of mutated genes.

Detection of cancer can be a deterministic process, where cancer is always detected (and, thus, simulation ended) when certain conditions are met (`detectionSize`, `detectionDrivers`, `fixation`). Alternatively, it can be stochastic process where probability of detection depends on size. Every so often (see below) we assess population size, and detect cancer or not probabilistically (comparing the probability of detection for that size with a random uniform number). Probability of detection changes with population size according to the function

1 - e^{-cPDetect ((populationsize - PDBaseline)/PDBaseline)}

.

You can pass `cPDetect` manually (you will need to set n2 and p2 to NA). However, it might be more intuitive to specify the pair n2, p2, such that the probability of detection is *p2* for population size *n2* (and from that pair we solve for the value of `cPDetect`). How often do we check? That is controlled by `checkSizePEvery`, the (minimal) time between successive checks (from among the sampling times given by `sampleEvery`: the interval between successive assessments will be the smallest multiple integer of `sampleEvery` that is larger than `checkSizePEvery` —see vignette for details). `checkSizePEvery` has, by default, a different (and much larger) value than `sampleEvery` both to allow to examine the effects of sampling, and to avoid many costly random number generations.

Please note that `detectionProb` is NOT available with version 1 objects.

Value

For `oncoSimulIndiv` a list, of class "oncosimul", with the following components:

`pops.by.time`	A matrix of the population sizes of the clones, with clones in columns and time in row. Not all clones are shown here, only those that were present in at least on of the keepEvery samples.
`NumClones`	Total number of clones in the above matrix. This is not the total number of distinct clones that have appeared over all simulations (which is likely to be larger or much larger).
`TotalPopSize`	Total population size at the end.
`Genotypes`	A matrix of genotypes. For each of the clones in the pops.by.time matrix, its genotype, with a 0 if the gene is not mutated and a 1 if it is mutated.
`MaxNumDrivers`	The largest number of mutated driver genes ever seen in the simulation in any clone.
`MaxDriversLast`	The largest number of mutated drivers in any clone at the end of the simulation.
`NumDriversLargestPop`	The number of mutated driver genes in the clone with largest population size.
`LargestClone`	Population size of the clone with largest number of population size.
`PropLargestPopLast`	Ratio of LargestClone/TotalPopSize
`FinalTime`	The time (in time units) at the end of the simulation.
`NumIter`	The number of iterations of the BNB algorithm.
`HittedWallTime`	TRUE if we reached the limit of max.wall.time. FALSE otherwise.
`TotalPresentDrivers`	The total number of mutated driver genes, whether or not in the same clone. The number of elements in `OccurringDrivers`, below.
`CountByDriver`	A vector of length number of drivers, with the count of the number of clones that have that driver mutated.
`OccurringDrivers`	The actual number of drivers mutated.
`PerSampleStats`	A 5 column matrix with a row for each sampling period. The columns are: total population size, population size of the largest clone, the ratio of the two, the largest number of drivers in any clone, and the number of drivers in the clone with the largest population size.
`other`	A list that contains statistics for an estimate of the simulation error when using the McFarland model as well as other statistics. For the McFarland model, the relevant value is errorMF, which is -99 unless in the McFarland model. For the McFarland model it is the largest difference of successive death rates. The entries named minDMratio and minBMratio are the smallest ratio, over all simulations, of death rate to mutation rate and birth rate to mutation rate, respectively. The BNB algorithm thrives when those are large.

For `oncoSimulPop` a list of length `Nindiv`, and of class `"oncosimulpop"`, where each element of the list is itself a list, of class `oncosimul`, with components as described above.

In v.2, the output is of both class "oncosimul" and "oncosimul2". The oncoSimulIndiv return object differs in

`GenotypesWDistinctOrderEff`	A list of vectors, where each vector corresponds to a genotype in the `Genotypes`, showing (where it matters) the order of mutations. Each vector shows the genotypes, with the numeric codes, showing explicitly the order when it matters. So if you have genes 1, 2, 7 for which order relationships are given, and genes 3, 4, 5, 6 for which other interactions exist, any mutations in 1, 2, 7 are shown first, and in the order they occurred, before showing the rest of the mutations. See details.
`GenotypesLabels`	The genotypes, as character vectors with the original labels provided (i.e., not the integer codes). As before, mutated genes, for those where order matters, come first, and are separated by the rest by a "_". See details.
`OccurringDrivers`	This is the same as in v.1, but we use the labels, not the numeric id codes. Of course, if you entered integers as labels for the genes, you will see numbers (however, as a character string).

Note

Please, note that the meaning of the fitness effects in the McFarland model is not the same as in the original paper; the fitness coefficients are transformed to allow for a simpler fitness function as a product of terms. This differs with respect to v.1. See the vignette for details.

Author(s)

Ramon Diaz-Uriarte

References

Bozic, I., et al., (2010). Accumulation of driver and passenger mutations during tumor progression. *Proceedings of the National Academy of Sciences of the United States of America\/*, **107**, 18545–18550.

Diaz-Uriarte, R. (2015). Identifying restrictions in the order of accumulation of mutations during tumor progression: effects of passengers, evolutionary models, and sampling http://www.biomedcentral.com/1471-2105/16/41/abstract

Gerstung et al., 2011. The Temporal Order of Genetic and Pathway Alterations in Tumorigenesis. *PLoS ONE*, 6.

McFarland, C.~D. et al. (2013). Impact of deleterious passenger mutations on cancer progression. *Proceedings of the National Academy of Sciences of the United States of America\/*, **110**(8), 2910–5.

Mather, W.~H., Hasty, J., and Tsimring, L.~S. (2012). Fast stochastic algorithm for simulating evolutionary population dynamics. *Bioinformatics (Oxford, England)\/*, **28**(9), 1230–1238.

See Also

plot.oncosimul, examplePosets, samplePop, allFitnessEffects

Examples

```
#################################
#####
##### Examples using v.1
#####
#################################


## use poset p701
data(examplePosets)
p701 <- examplePosets[["p701"]]

## Exp Model

b1 <- oncoSimulIndiv(p701)
summary(b1)

plot(b1, addtot = TRUE)

## McFarland; use a small sampleEvery, but also a reasonable
##   keepEvery.
## We also modify mutation rate to values similar to those in the
##   original paper.
## Note that detectionSize will play no role
## finalTime is large, since this is a slower process
## initSize is set to 4000 so the default K is larger and we are likely
## to reach cancer. Alternatively, set K = 2000.

m1 <- oncoSimulIndiv(p701,
                     model = "McFL",
                     mu = 5e-7,
                     initSize = 4000,
                     sampleEvery = 0.025,
                     finalTime = 15000,
                     keepEvery = 10,
                     onlyCancer = FALSE)
plot(m1, addtot = TRUE, log = "")




## Simulating 4 individual trajectories
## (I set mc.cores = 2 to comply with --as-cran checks, but you
##  should either use a reasonable number for your hardware or
##  leave it at its default value).


p1 <- oncoSimulPop(4, p701,
                   keepEvery = 10,
                   mc.cores = 2)
summary(p1)
samplePop(p1)

p2 <- oncoSimulSample(4, p701)


#########################################
######
###### Examples using v.2:
######
#########################################



#### A model similar to the one in McFarland. We use 2070 genes.

set.seed(456)
nd <- 70
np <- 2000
s <- 0.1
sp <- 1e-3
spp <- -sp/(1 + sp)
mcf1 <- allFitnessEffects(noIntGenes = c(rep(s, nd), rep(spp, np)),
                          drv = seq.int(nd))
mcf1s <-  oncoSimulIndiv(mcf1,
                         model = "McFL",
                         mu = 1e-7,
                         detectionSize = 1e8,
                         detectionDrivers = 100,
                         sampleEvery = 0.02,
                         keepEvery = 2,
                         initSize = 2000,
                         finalTime = 1000,
                         onlyCancer = FALSE)
plot(mcf1s, addtot = TRUE, lwdClone = 0.6, log = "")
summary(mcf1s)
plot(mcf1s)


#### Order effects with modules, and 5 genes without interactions
#### with fitness effects from an exponential distribution

oi <- allFitnessEffects(orderEffects =
               c("F > D" = -0.3, "D > F" = 0.4),
               noIntGenes = rexp(5, 10),
                          geneToModule =
                              c("Root" = "Root",
                                "F" = "f1, f2, f3",
                                "D" = "d1, d2") )
oiI1 <- oncoSimulIndiv(oi, model = "Exp")
oiI1$GenotypesLabels
oiI1   ## note the order and separation by "_"


oiP1 <- oncoSimulPop(2, oi,
                     keepEvery = 10,
                     mc.cores = 2)
summary(oiP1)


## Even if order exists, this cannot reflect it;
## G1 to G10 are d1, d2, f1..,f3, and the 5 genes without
## interaction
samplePop(oiP1)


oiS1 <- oncoSimulSample(2, oi)

## The output contains only the summary of the runs AND
## the sample:
oiS1

## And their sizes do differ
object.size(oiS1)
object.size(oiP1)



######## Using a poset for pancreatic cancer from Gerstung et al.
###      (s and sh are made up for the example; only the structure
###       and names come from Gerstung et al.)


pancr <- allFitnessEffects(data.frame(parent = c("Root", rep("KRAS", 4), "SMAD4", "CDNK2A",
                                          "TP53", "TP53", "MLL3"),
                                      child = c("KRAS","SMAD4", "CDNK2A",
                                          "TP53", "MLL3",
                                          rep("PXDN", 3), rep("TGFBR2", 2)),
                                      s = 0.05,
                                      sh = -0.3,
                                      typeDep = "MN"))
plot(pancr)

### Use an exponential growth model

pancr1 <- oncoSimulIndiv(pancr, model = "Exp")
pancr1
summary(pancr1)
plot(pancr1)
pancr1$GenotypesLabels


## Pop and Sample
pancrPop <- oncoSimulPop(4,
                         pancr,
                         keepEvery = 10,
                         mc.cores = 2)
summary(pancrPop)
pancrSPop <- samplePop(pancrPop)
pancrSPop

pancrSamp <- oncoSimulSample(2, pancr)
pancrSamp


## Using gene-specific mutation rates
muv <- c("U" = 1e-3, "z" = 1e-7, "e" = 1e-6, "m" = 1e-5, "D" = 1e-4)
ni <- rep(0.01, 5)
names(ni) <- names(muv)
femuv <- allFitnessEffects(noIntGenes = ni)
oncoSimulIndiv(femuv, mu = muv)
```

```
#########Frequency dependent fitness examples

##A example with cooperation. Presence of WT favours all clones
genofit <- data.frame(A = c(0, 1, 0, 1),
                      B = c(0, 0, 1, 1),
                      Fitness = c("3 + 5*f_",
                                  "3 + 5*(f_ + f_1)",
                                  "3 + 5*(f_ + f_2)",
                                  "5 + 6*(f_ + f_1_2)"))

afe <- allFitnessEffects(genotFitness = genofit,
                         frequencyDependentFitness = TRUE)

osi <- oncoSimulIndiv(afe,
                      model = "McFL",
                      onlyCancer = FALSE,
                      finalTime = 5000,
                      mu = 1e-6,
                      initSize = 5000,
                      keepPhylog = FALSE,
                      seed = NULL,
                      errorHitMaxTries = FALSE,
                      errorHitWallTime = FALSE)
osi

plot(osi, show = "genotypes", type = "line")

osp <- oncoSimulPop(5,
                    afe,
                    model = "McFL",
                    initSize = 5000,
                    mu = 1e-6,
                    keepEvery = 5,
                    mc.cores = 2,
                    finalTime = 5000)

sp <- samplePop(osp)

sp

##A little bit more complex example situation. WT favours clones A and B. A and
##B compete with each other. Presence of A and B favour clone A, B.
genofit <- data.frame(A = c(0, 1, 0, 1),
                      B = c(0, 0, 1, 1),
                      Fitness = c("3 + 5*f_",
                                  "3 + 5*(f_ + f_1 - f_2)",
                                  "3 + 5*(f_ + f_2 - f_1)",
                                  "5 + 6*(f_1 + f_2 + f_1_2)"))

afe <- allFitnessEffects(genotFitness = genofit,
                         frequencyDependentFitness = TRUE)

osi <- oncoSimulIndiv(afe,
                      model = "McFL",
                      onlyCancer = FALSE,
                      finalTime = 5000,
                      mu = 1e-6,
                      initSize = 5000,
                      keepPhylog = FALSE,
                      seed = NULL,
                      errorHitMaxTries = FALSE,
                      errorHitWallTime = FALSE)
osi

plot(osi, show = "genotypes", type = "line")

osp <- oncoSimulPop(5,
                    afe,
                    model = "McFL",
                    initSize = 5000,
                    onlyCancer = FALSE,
                    mu = 1e-6,
                    keepEvery = 5,
                    mc.cores = 2)

summary(osp)

sp <- samplePop(osp)

sp

oss <- oncoSimulSample(5,
                       afe,
                       model = "McFL",
                       initSize = 5000,
                       mu = 1e-6,
                       finalTime = 5000,
                       verbosity = 0)

oss
```

```
## Reinitialize the RNG
set.seed(NULL)
```
