OncoSimulR: forward genetic simulation in asexual populations with arbitrary epistatic interactions and a focus on modeling tumor progression.

Ramon Diaz-Uriarte, Sergio Sanchez-Carrillo, Juan Antonio M:

Department of Biochemistry, School of Medicine, Universidad Autónoma de Madr

r.diaz@uam.es, rdiaz0:

2024-07-25. OncoSimulR version 4.7.1. Revision: 3044304

Contents

1 Introduction

OncoSimulR is an individual- or clone-based forward-time genetic simulator for biallelic markers (wildtype vs. mutated) in asexually reproducing populations without spatial structure (perfect mixing). Its design emphasizes flexible specification of fitness and mutator effects.

OncoSimulR was originally developed to simulate tumor progression with emphasis on allowing users to set restrictions in the accumulation of mutations as specified, for example, by Oncogenetic Trees (OT: Desper et al., 1999; Szabo & Boucher, 2008) or Conjunctive Bayesian Networks (CBN: Beerenwinkel, Eriksson, et al., 2007; Gerstung et al., 2009; Gerstung, Eriksson, et al., 2011), with the possibility of adding passenger mutations to the simulations and allowing for several types of sampling.

Since then, OncoSimulR has been vastly extended to allow you to specify other types of restrictions in the accumulation of genes, such as the XOR models of Korsunsky et al. (2014) or the "semimonotone" model of Farahani & Lagergren (2013). Moreover, different fitness effects related to the order in which mutations appear can also be incorporated, involving arbitrary numbers of genes. This is *very* different from "restrictions in the order of accumulation of mutations". With order effects, described in a recent cancer paper by Ortmann and collaborators (Ortmann et al., 2015), the effect of having both mutations "A" and "B" differs depending on whether "A" appeared before or after "B" (the actual case involves genes JAK2 and TET2).

More generally, OncoSimulR now also allows you to specify arbitrary epistatic interactions between arbitrary collections of genes and to model, for example, synthetic mortality or synthetic viability (again, involving an arbitrary number of genes, some of which might also depend on other genes, or show order effects with other genes). Moreover, it is possible to specify the above interactions in terms of modules, not genes. This idea is discussed in, for example, Raphael & Vandin (2015) and Gerstung, Eriksson, et al. (2011): the restrictions encoded in, say, CBNs or OT can be considered to apply not to genes, but to modules, where each module is a set of genes (and the intersection between modules is the empty set) that performs a specific biological function. Modules, then, play the role of a "union operation" over the set of genes in a module. In addition, arbitrary numbers of genes without interactions (and with fitness effects coming from any distribution you might want) are also possible.

You can also directly specify the mapping between genotypes and fitness and, thus, you can simulate on fitness landscapes of arbitrary complexity.

It is now (released initially in this repo as the freq-dep-fitness branch on February 2019) also possible to simulate scenarios with frequency-dependent fitness, where the fitness of one or more genotypes depends on the relative or absolute frequencies of other genotypes, as in game theory and adaptive dynamics. This makes it possible to model predation and parasitism, cooperation and mutualism, and commensalism. It also allows to model therapeutic interventions (where fitness changes at specified time points or as a function of the total populations size or as a function of arbitrary user-defined variables); in particular, it is possible to emulate adaptive therapy (Hansen & Read (2020b); Hansen & Read (2020a)).

Simulations can start from arbitrary initial population compositions and it is also possible to simulate multiple species. Thus, simulations that involve both ecological and evolutionary processes are possible.

Mutator/antimutator genes, genes that alter the mutation rate of other genes (Gerrish et al., 2007; Tomlinson et al., 1996), can also be simulated with OncoSimulR and specified with most of the mechanisms above (you can have, for instance, interactions between mutator genes). And, regardless of the presence or not of other mutator/antimutator genes, different genes can have different mutation rates.

Simulations can be stopped as a function of total population size, number of mutated driver genes, or number of time periods. Simulations can also be stopped with a stochastic detection mechanism where the probability of detecting a tumor increases with total population size. Simulations return the number of cells of every genotype/clone at each of the sampling periods and we can take samples from the former with single-cell or whole- tumor resolution, adding noise if we want. If we ask for them, simulations also store and return the genealogical relationships of all clones generated during the simulation.

The models so far implemented are all continuous time models, which are simulated using the BNB algorithm of Mather et al. (2012). The core of the code is implemented in C++, providing for fast execution. To help with simulation studies, code to simulate random graphs of the kind often seen in CBNs, OTs, etc, is also available. Finally, OncoSimulR also allows for the generation of random fitness landscapes and the representation of fitness landscapes and provides statistics of evolutionary predictability.

Funding

1.1 Key features of OncoSimulR

As mentioned above, OncoSimulR is now a very general package for forward genetic simulation, with applicability well beyond tumor progression. This is a summary of some of its key features:

- You can specify arbitrary interactions between genes, with arbitrary fitness effects, with explicit support for:

  - Restrictions in the accumulations of mutations, as specified by Oncogenetic Trees (OTs), Conjunctive Bayesian Networks (CBNs), semimonotone progression networks, and XOR relationships.

  - Epistatic interactions including, but not limited to, synthetic viability and synthetic lethality.

  - Order effects.

- You can add passenger mutations.

- You can add mutator/antimutator effects.

- Fitness and mutation rates can be gene-specific.

- You can add arbitrary numbers of non-interacting genes with arbitrary fitness effects.

- you can allow for deviations from the OT, CBN, semimonotone, and XOR models, specifying a penalty for such deviations (the s_h parameter).

- You can conduct multiple simulations, and sample from them with different temporal schemes and using both whole tumor or single cell sampling.

- You can stop the simulations using a flexible combination of conditions: final time, number of drivers, population size, fixation of certain genotypes, and a stochastic stopping mechanism that depends on population size.

- Right now, three different models are available, two that lead to exponential growth, one of them loosely based on Bozic et al. (2010), and another that leads to logistic-like growth, based on McFarland et al. (2013).

- You can use large numbers of genes (e.g., see an example of 50000 in section 6.5.3).

- Simulations are generally very fast: I use C++ to implement the BNB algorithm (see sections 18.5 and 18.6 for more detailed comments on the usage of this algorithm).

- You can obtain the true sequence of events and the phylogenetic relationships between clones (see section 18.1 for the details of what we mean by "clone").

- You can generate random fitness landscapes (under the House of Cards, Rough Mount Fuji, or additive models, or combinations of the former and under the NK model) and use those landscapes as input to the simulation functions.

- You can plot fitness landscapes.

- You can obtain statistics of evolutionary predictability from the simulations.

- You can now also use simulations with frequency-dependent fitness: fitness (birth rate) is not fixed for a genotype, but can be a function of the frequecies of the clones (see section 10). We can therefore use OncoSimulR to examine, via simulations, results from game theory and adaptive dynamics and study complex scenarios that are not amenable to analytical solutions. More generally, we can model predation and parasitism, cooperation and mutualism, and commensalism.

- It is possible to start the simulation with arbitrary initial composition (section 6.7) and to simulate multiple species (section 6.8). You can thus run simulations that involve both ecological and evolutionary processes involving inter-species relationships plus genetic restrictions in evolution.

- It is possible to simulate many different therapeutic interventions. Section 15 shows examples of interventions where certain genotypes change fitness (because of chemotherapy) at specified times. More generally, since fitness (birth rates) can be made a function of total populations sizes and/or frequencies (see section 10), many different arbitrary intervention schemes can be simulated. Possible models are, of course, not limited to cancer chemotherapy, but could include antibiotic treatment of bacteria, antiviral therapy, etc.

The table below, modified from the table at the Genetics Simulation Resources (GSR) page, provides a summary of the key features of OncoSimulR. (An explanation of the meaning of terms specific to the GSR table is available from https://popmodels. cancercontrol.cancer.gov/gsr/search/ or from the Genetics Simulation Resources table itself, by moving the mouse over each term).

Table 1: Key features of OncoSimulR. Modified from the original table from https://popmodels.ca ncercontrol.cancer.gov/gsr/packages/oncosimulr /#detailed .

Attribute Category	Attribute
Target	
Type of Simulated Data	Haploid DNA Sequence
Variations	Biallelic Marker, Genotype or Sequencing Error
Simulation Method	Forward-time
Type of Dynamical Model	Continuous time
Entities Tracked	Clones (see 18.2)
Input	Program specific (R data frames and matrices specifying genotypes' fitness, gene effects, and starting genotype)
Output	
Data Type	Genotype or Sequence, Individual Relationship (complete parent-child relationships between clones), Demographic (populations sizes of all clones at sampling times), Diversity Measures (LOD, POM, diversity of genotypes), Fitness
Sample Type	Random or Independent, Longitudinal, Other (proportional to population size)
Evolutionary Features	
Mating Scheme	Asexual Reproduction
Demographic	
Population Size Changes	Exponential (two models), Logistic (McFarland et al., 2013)
Fitness Components	
Birth Rate	Individually Determined from Genotype (models "Exp", "McFL", "McFLD"). Frequency-Dependently Determined from Genotype (models "Exp", "McFL", "McFLD")
Death Rate	Individually Determined from Genotype (model "Bozic"), Influenced by Environment —population size (models "McFL" and "McFLD")
Natural Selection	
Determinant	Single and Multi-locus, Fitness of Offspring, Environmental Factors (population size, genotype frequencies)
Models	Directional Selection, Multi-locus models, Epistasis, Random Fitness Effects, Frequency-Dependent
Mutation Models	Two-allele Mutation Model (wildtype, mutant), without back mutation

Attribute Category	Attribute
Events Allowed	Varying Genetic Features: change of individual mutation rates (mutator/antimutator genes)
Spatial Structure	No Spatial Structure (perfectly mixed and no migration)

Further details about the original motivation for wanting to simulate data this way in the context of tumor progression can be found in Diaz-Uriarte (2015), where additional comments about model parameters and caveats are discussed.

Are there similar programs? The Java program by Reiter et al. (2013), TTP, offers somewhat similar functionality to the previous version of OncoSimulR, but it is restricted to at most four drivers (whereas v.1 of OncoSimulR allowed for up to 64), you cannot use arbitrary CBNs or OTs (or XORs or semimonotone graphs) to specify restrictions, there is no allowance for passengers, and a single type of model (a discrete time Galton-Watson process) is implemented. The current functionality of OncoSimulR goes well beyond the the previous version (and, thus, also the TPT of Reiter et al. (2013)). We now allow you to specify all types of fitness effects in other general forward genetic simulators such as FFPopSim (Zanini & Neher, 2012), and some that, to our knowledge (e.g., order effects) are not available from any genetics simulator. In addition, the "Lego system" to flexibly combine different fitness specifications is also unique; by "Lego system" I mean that we can combine different pieces and blocks, similarly to what we do with Lego bricks. (I find this an intuitive and very graphical analogy, which I have copied from Hothorn et al. (2006) and Hothorn et al. (2008)). In a nutshell, salient features of OncoSimulR compared to other simulators are the unparalleled flexibility to specify fitness and mutator effects, with modules and order effects as particularly unique, and the options for sampling and stopping the simulations, particularly convenient in cancer evolution models. Also unique in this type of software is the addition of functions for simulating fitness landscapes and assessing evolutionary predictability.

1.2 What kinds of questions is OncoSimulR suited for?

OncoSimulR can be used to address questions that span from the effect of mutator genes in cancer to the interplay between fitness landscapes and mutation rates. The main types of questions that OncoSimulR can help address involve combinations of:

- Simulating asexual evolution (the `oncoSimul*` functions) where:
  - Fitness is:
    * A function of specific epistatic effects between genes
    * A function of order effects
    * A function of epistatic effects specified using DAGs/posets where these DAGs/posets:
      · Are user-specified
      · Generated randomly (`simOGraph`)

* Any mapping between genotypes and fitness where this mapping is:
  · User-specified
  · Generated randomly from families of random fitness landscapes (`rfitness`)
* A function of the frequency of other genotypes (i.e., frequency-dependent fitness), such as in adaptive dynamics (see section 10 for more details). This also allows you to model competition, cooperation and mutualism, parasitism and predation, and commensalism between clones.
  – Mutation rates can:
    * Vary between genes
    * Be affected by other genes

- Examining times to evolutionarily or biomedically relevant events (fixation of genotypes, reaching a minimal size, acquiring a minimal number of driver genes, etc —specified with the stopping conditions to the `oncoSimul*` functions).

- Using different sampling schemes (`samplePop`) that are related to:

  – Assessing genotypes from single-cell vs. whole tumor (or whole population) with the `typeSample` argument
  – Genotyping error (`propError` argument)
  – Timing of samples (`timeSample` argument)
  – . . . and assessing the consequences of those on the observed genotypes and their diversity (`sampledGenotypes`) and any other inferences that depend on the observational process.
  – (OncoSimulR returns the abundances of all genotypes at each of the sampling points, so you are not restricted by what the `samplePop` function provides.)

- Tracking the genealogical relationships of clones (`plotClonePhylog`) and assessing evolutionary predictability (`LOD`, `POM`).

Some specific questions that you can address with the help of OncoSimulR are discussed in section 1.3.

––––––––––––––––––––

A quick overview of the main functions and their relationships is shown in Figure 1, where we use italics for the type/class of R object and courier font for the name of the functions.
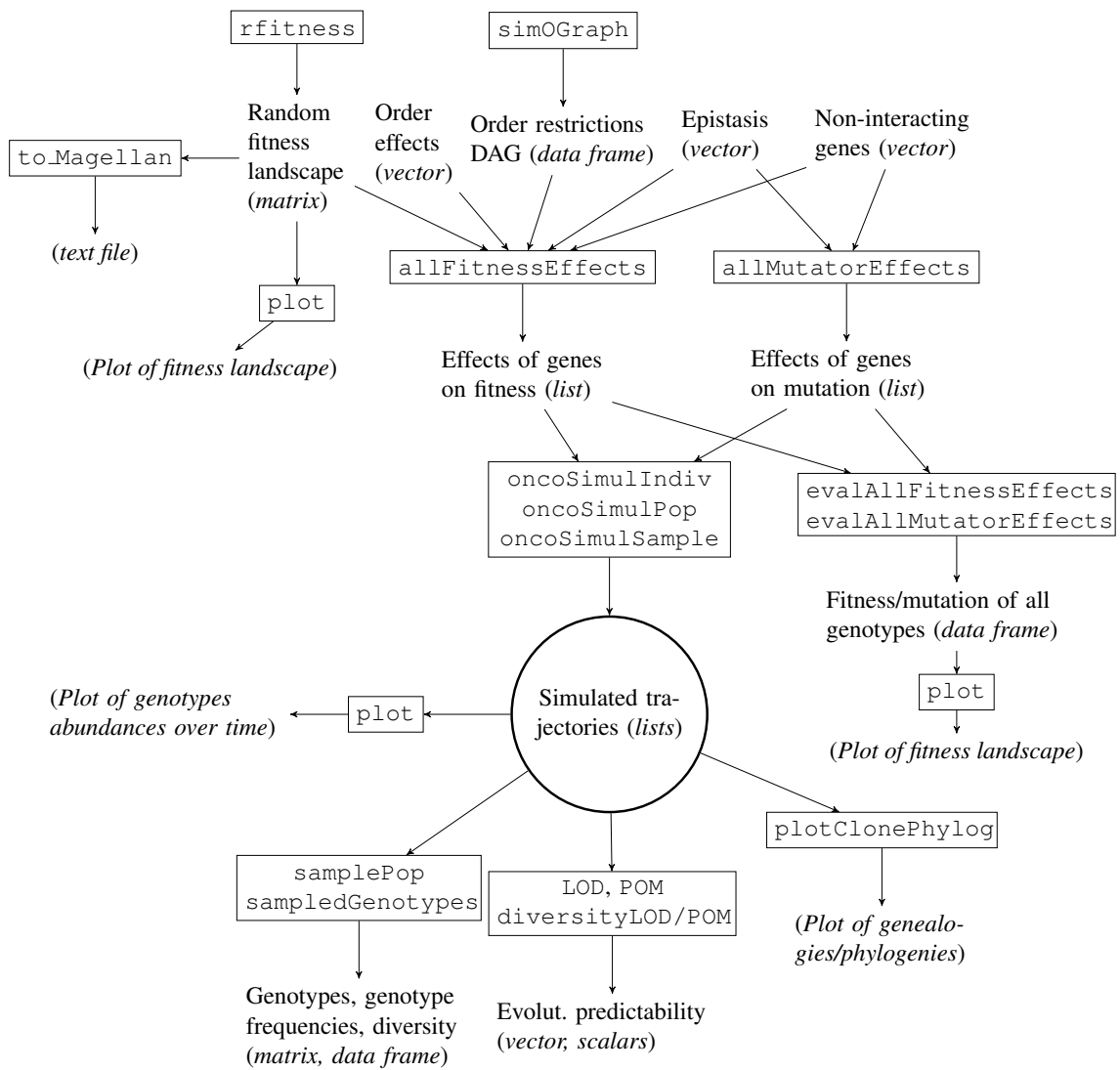
Figure 1: Relationships between the main functions in OncoSimulR.

1.3 Examples of questions that can be addressed with OncoSimulR

Most of the examples in the rest of this vignette, starting with those in 1.7, focus on the mechanics. Here, we will illustrate some problems in cancer genomics and evolutionary genetics where OncoSimulR could be of help. This section does not try to provide an answer to any of these questions (those would be full papers by themselves). Instead, this section simply tries to illustrate some kinds of questions where you can use OncoSimulR; of course, the possible uses of OncoSimulR are only limited by your ingenuity. Here, I will only use short snippets of working code as we are limited by time of execution; for real work you would want to use many more scenarios and many more simulations, you would use appropriate statistical methods to compare the output of runs, etc, etc, etc.

```
## Load the package
library(OncoSimulR)
```

1.3.1 Recovering restrictions in the order of accumulation of mutations

This is a question that was addressed, for instance, in Diaz-Uriarte (2015): do methods that try to infer restrictions in the order of accumulation of mutations (Gerstung et al., 2009; Ramazzotti et al., 2015; e.g., Szabo & Boucher, 2008) work well under different evolutionary models and with different sampling schemes?

A possible way to examine that question would involve:

- generating random DAGs that encode restrictions;
- simulating cancer evolution using those DAGs;
- sampling the data and adding different levels of noise to the sampled data;
- running the inferential method;
- comparing the inferred DAG with the original, true, one.

```
## For reproducibility
set.seed(2)
RNGkind("L'Ecuyer-CMRG")

## Simulate a DAG
g1 <- simOGraph(4, out = "rT")

## Simulate 10 evolutionary trajectories
s1 <- oncoSimulPop(10, allFitnessEffects(g1, drvNames = 1:4),
                   onlyCancer = TRUE,
                   mc.cores = 2, ## adapt to your hardware
                   seed = NULL) ## for reproducibility of vignette

## Sample those data uniformly, and add noise
d1 <- samplePop(s1, timeSample = "unif", propError = 0.1)
##
##   Subjects by Genes matrix of 10 subjects and 4 genes.
```

```
## You would now run the appropriate inferential method and
## compare observed and true. For example

## require(Oncotree)
## fit1 <- oncotree.fit(d1)

## Now, you'd compare fitted and original. This is well beyond
## the scope of this document (and OncoSimulR itself).
```

1.3.2 Sign epistasis and probability of crossing fitness valleys

This question, and the question in the next section (1.3.3), encompass a wide range of issues that have been addressed in evolutionary genetics studies and which include from detailed analysis of simple models with a few uphill paths and valleys as in Weissman et al. (2009) or Ochs & Desai (2015), to questions that refer to larger, more complex fitness landscapes as in Szendro, Franke, et al. (2013) or Franke et al. (2011) or Krug (2019) (see below).

Using as an example Ochs & Desai (2015) (we will see this example again in section 5.3, where we cover different ways of specifying fitness), we could specify the fitness landscape and run simulations until fixation (with argument `fixation` to `oncoSimulPop` —see more details in section 6.3.3 and 6.3.4, again with this example). We would then examine the proportion of genotypes fixed under different scenarios. And we can extend this example by adding mutator genes:

```
## For reproducibility
set.seed(2)
RNGkind("L'Ecuyer-CMRG")

## Specify fitness effects.

## Numeric values arbitrary, but set the intermediate genotype en
## route to ui as mildly deleterious so there is a valley.

## As in Ochs and Desai, the ui and uv genotypes
## can never appear.

u <- 0.2; i <- -0.02; vi <- 0.6; ui <- uv <- -Inf

od <- allFitnessEffects(
    epistasis = c("u" = u,   "u:i" = ui,
                  "u:v" = uv, "i" = i,
                  "v:-i" = -Inf, "v:i" = vi))

## For the sake of extending this example, also turn i into a
```

```
## mutator gene

odm <- allMutatorEffects(noIntGenes = c("i" = 50))

## How do mutation and fitness look like for each genotype?
evalAllGenotypesFitAndMut(od, odm, addwt = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
##    Genotype Birth MutatorFactor
## 1        WT 1.000             1
## 2         i 0.980            50
## 3         u 1.200             1
## 4         v 0.000             1
## 5      i, u 0.000            50
## 6      i, v 1.568            50
## 7      u, v 0.000             1
## 8   i, u, v 0.000            50
```

Ochs and Desai explicitly say "Each simulated population was evolved until either the uphill genotype or valley-crossing genotype fixed." So we will use `fixation`.

```
## Set a small initSize, as o.w. unlikely to pass the valley
initS <- 10
## The number of replicates is tiny, for the sake of speed
## of creation of the vignette. Even fewer in Windows, since we run on a single
## core

if(.Platform$OS.type == "windows") {
    nruns <- 4
} else {
    nruns <- 10
}

od_sim <- oncoSimulPop(nruns, od, muEF = odm,
                       fixation = c("u", "i, v"), initSize = initS,
                       model = "McFL",
                       mu = 1e-4, detectionDrivers = NA,
                       finalTime = NA,
                       detectionSize = NA, detectionProb = NA,
                       onlyCancer = TRUE,
                       mc.cores = 2, ## adapt to your hardware
                       seed = NULL) ## for reproducibility
## What is the frequency of each final genotype?
sampledGenotypes(samplePop(od_sim))
##
##  Subjects by Genes matrix of 10 subjects and 3 genes.
##    Genotype Freq
```

```
## 1      i, v     4
## 2         u     6
##
##  Shannon's diversity (entropy) of sampled genotypes: 0.673
```

1.3.3 Predictability of evolution in complex fitness landscapes

Focusing now on predictability in more general fitness landscapes, we would run simulations under random fitness landscapes with varied ruggedness, and would then examine the evolutionary predictability of the trajectories with measures such as "Lines of Descent" and "Path of the Maximum" (Szendro, Franke, et al., 2013) and the diversity of the sampled genotypes under different sampling regimes (see details in section 16). (See also related comments in section 1.5).

```
## For reproducibility
set.seed(7)
RNGkind("L'Ecuyer-CMRG")


## Repeat the following loop for different combinations of whatever
## interests you, such as number of genes, or distribution of the
## c and sd (which affect how rugged the landscape is), or
## reference genotype, or evolutionary model, or stopping criterion,
## or sampling procedure, or ...

##  Generate a random fitness landscape, from the Rough Mount
##  Fuji model, with g genes, and c ("slope" constant) and
##  reference chosen randomly (reference is random by default and
##  thus not specified below). Require a minimal number of
##  accessible genotypes

g <- 6
c <- runif(1, 1/5, 5)
rl <- rfitness(g, c = c, min_accessible_genotypes = g)

## Plot it if you want; commented here as it takes long for a
## vignette

## plot(rl)

## Obtain landscape measures from MAGELLAN. Export to MAGELLAN and
## call your own copy of MAGELLAN's binary

## to_Magellan(rl, file = "rl1.txt") ## (Commented out here to avoid writing file

## or use the binary copy provided with OncoSimulR
```

```
## see also below.

Magellan_stats(rl) ## (Commented out here to avoid writing files)
##       ngeno      npeaks      nsinks       gamma      gamma.         r.s
##      64.000       2.000       1.000       0.769       0.854       0.372
##     nchains      nsteps        nori       depth        magn        sign
##       1.000       5.000       4.000       2.000       0.863       0.129
##       rsign        f.1.        X.2.       f.3..      mode_f      outD_m
##       0.008       0.916       0.010       0.074       1.000       0.346
##      outD_v     steps_m     reach_m       fitG_m       opt_i mProbOpt_0
##       1.587       3.052      13.597      32.452      60.000       0.128
##     opt_i.1 mProbOpt_1
##      63.000       0.872
```

```
## Simulate evolution in that landscape many times (here just 10)
simulrl <- oncoSimulPop(10, allFitnessEffects(genotFitness = rl),
                        keepPhylog = TRUE, keepEvery = 1,
                        onlyCancer = TRUE,
                        initSize = 4000,
                        seed = NULL, ## for reproducibility
                        mc.cores = 2) ## adapt to your hardware

## Obtain measures of evolutionary predictability
diversityLOD(LOD(simulrl))
## [1] 1.418
```

```
diversityPOM(POM(simulrl))
## [1] 1.418
```

```
sampledGenotypes(samplePop(simulrl, typeSample = "whole"))
##
##  Subjects by Genes matrix of 10 subjects and 6 genes.
##   Genotype Freq
## 1        A    1
## 2        B    1
## 3     D, F    1
## 4        E    4
## 5        F    3
##
##  Shannon's diversity (entropy) of sampled genotypes: 1.418
```

1.3.4   Mutator and antimutator genes

The effects of mutator and antimutator genes have been examined both in cancer genetics (Nowak, 2006; Tomlinson et al., 1996) and in evolutionary genetics (Gerrish et al., 2007), and are related to wider issues such as Muller's ratchet and the evolution of sex. There are, thus, a large range of questions related to mutator and antimutator

genes.

One question addressed in Tomlinson et al. (1996) concerns under what circumstances mutator genes are likely to play a role in cancer progression. For instance, Tomlinson et al. (1996) find that an increased mutation rate is more likely to matter if the number of required mutations in driver genes needed to reach cancer is large and if the mutator effect is large.

We might want to ask, then, how long it takes before to reach cancer under different scenarios. Time to reach cancer is stored in the component `FinalTime` of the output. We would specify different numbers and effects of mutator genes (argument `muEF`). We would also change the criteria for reaching cancer and in our case we can easily do that by specifying different numbers in `detectionDrivers`. Of course, we would also want to examine the effects of varying numbers of mutators, drivers, and possibly fitness consequences of mutators. Below we assume mutators are neutral and we assume there are no additional genes with deleterious mutations, but this need not be so, of course (Gerrish et al., 2007; McFarland et al., 2014; see also Tomlinson et al., 1996).

Let us run an example. For the sake of simplicity, we assume no epistatic interactions.

```r
sd <- 0.1 ## fitness effect of drivers
sm <- 0 ## fitness effect of mutator
nd <- 20 ## number of drivers
nm <- 5  ## number of mutators
mut <- 10 ## mutator effect


fitnessGenesVector <- c(rep(sd, nd), rep(sm, nm))
names(fitnessGenesVector) <- 1:(nd + nm)
mutatorGenesVector <- rep(mut, nm)
names(mutatorGenesVector) <- (nd + 1):(nd + nm)


ft <- allFitnessEffects(noIntGenes = fitnessGenesVector,
                        drvNames = 1:nd)
mt <- allMutatorEffects(noIntGenes = mutatorGenesVector)
```

Now, simulate using the fitness and mutator specification. We fix the number of drivers to cancer, and we stop when those numbers of drivers are reached. Since we only care about the time it takes to reach cancer, not the actual trajectories, we set `keepEvery = NA`:

```r
## For reproducibility
set.seed(2)
RNGkind("L'Ecuyer-CMRG")



ddr <- 4
st <- oncoSimulPop(4, ft, muEF = mt,
                   detectionDrivers = ddr,
```

```
                  finalTime = NA,
                  detectionSize = NA,
                  detectionProb = NA,
                  onlyCancer = TRUE,
                  keepEvery = NA,
                  mc.cores = 2, ## adapt to your hardware
                  seed = NULL) ## for reproducibility

## How long did it take to reach cancer?
unlist(lapply(st, function(x) x$FinalTime))
## [1]  370  141 1793  282
```

(Incidentally, notice that it is easy to get OncoSimulR to throw an exception if you accidentally specify a huge mutation rate when all mutator genes are mutated: see section 18.7.)

1.3.5 Epistatic interactions between drivers and passengers in cancer and the consequences of order effects

1.3.5.1 Epistatic interactions between drivers and passengers   Bauer et al. (2014) have examined the effects of epistatic relationships between drivers and passengers in cancer initiation. We could use their model as a starting point, and examine how likely cancer is to develop under different variations of their model and different evolutionary scenarios (e.g., initial sample size, mutation rates, evolutionary model, etc).

There are several ways to specify their model, as we discuss in section 5.1. We will use one based on DAGs here:

```
K <- 4
sp <- 1e-5
sdp <- 0.015
sdplus <- 0.05
sdminus <- 0.1


cnt <- (1 + sdplus)/(1 + sdminus)
prod_cnt <- cnt - 1
bauer <- data.frame(parent = c("Root", rep("D", K)),
                    child = c("D", paste0("s", 1:K)),
                    s = c(prod_cnt, rep(sdp, K)),
                    sh = c(0, rep(sp, K)),
                    typeDep = "MN")
fbauer <- allFitnessEffects(bauer)
(b1 <- evalAllGenotypes(fbauer, order = FALSE, addwt = TRUE))
##             Genotype  Birth
## 1                 WT 1.0000
## 2                  D 0.9545
```

```
## 3                     s1 1.0000
## 4                     s2 1.0000
## 5                     s3 1.0000
## 6                     s4 1.0000
## 7                 D, s1 0.9689
## 8                 D, s2 0.9689
## 9                 D, s3 0.9689
## 10                D, s4 0.9689
## 11             s1, s2 1.0000
## 12             s1, s3 1.0000
## 13             s1, s4 1.0000
## 14             s2, s3 1.0000
## 15             s2, s4 1.0000
## 16             s3, s4 1.0000
## 17         D, s1, s2 0.9834
## 18         D, s1, s3 0.9834
## 19         D, s1, s4 0.9834
## 20         D, s2, s3 0.9834
## 21         D, s2, s4 0.9834
## 22         D, s3, s4 0.9834
## 23         s1, s2, s3 1.0000
## 24         s1, s2, s4 1.0000
## 25         s1, s3, s4 1.0000
## 26         s2, s3, s4 1.0000
## 27     D, s1, s2, s3 0.9981
## 28     D, s1, s2, s4 0.9981
## 29     D, s1, s3, s4 0.9981
## 30     D, s2, s3, s4 0.9981
## 31     s1, s2, s3, s4 1.0000
## 32 D, s1, s2, s3, s4 1.0131

## How does the fitness landscape look like?
plot(b1, use_ggrepel = TRUE) ## avoid overlapping labels
## Warning: ggrepel: 12 unlabeled data points (too many overlaps).
## Consider increasing max.overlaps
```
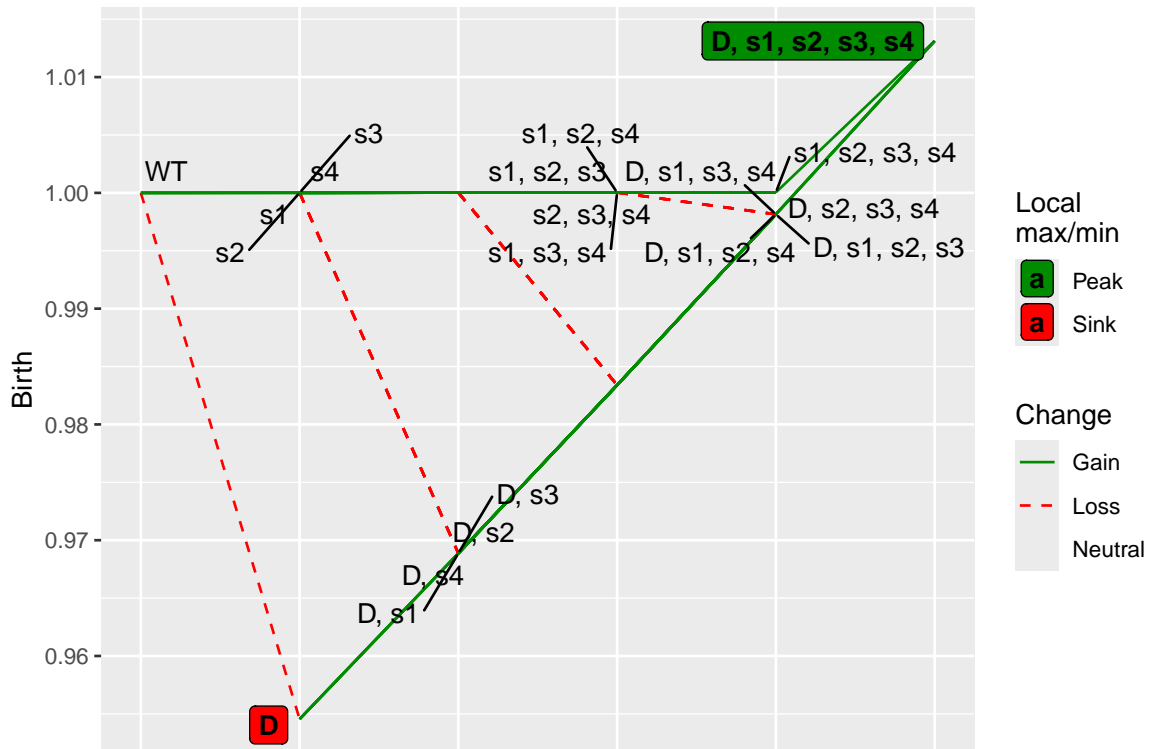
Now run simulations and examine how frequently the runs end up with population sizes larger than a pre-specified threshold; for instance, below we look at increasing population size 4x in the default maximum number of 2281 time periods (for real, you would of course increase the number of total populations, the range of initial population sizes, model, mutation rate, required population size or number of drivers, etc):

```
## For reproducibility
set.seed(2)
RNGkind("L'Ecuyer-CMRG")

totalpops <- 5
initSize <- 100
sb1 <- oncoSimulPop(totalpops, fbauer, model = "Exp",
                    initSize = initSize,
                    onlyCancer = FALSE,
                    mc.cores = 2, ## adapt to your hardware
                    seed = NULL) ## for reproducibility

## What proportion of the simulations reach 4x initSize?
sum(summary(sb1)[, "TotalPopSize"] > (4 * initSize))/totalpops
## [1] 0.2
```

Alternatively, to examine how long it takes to reach cancer for a pre-specified size, you could look at the value of `FinalTime` as we did above (section 1.3.4) after running simulations with `onlyCancer = TRUE` and `detectionSize` set to some reasonable value:

```
totalpops <- 5
initSize <- 100
sb2 <- oncoSimulPop(totalpops, fbauer, model = "Exp",
                    initSize = initSize,
                    onlyCancer = TRUE,
                    detectionSize = 10 * initSize,
                    mc.cores = 2, ## adapt to your hardware
                    seed = NULL) ## for reproducibility


## How long did it take to reach cancer?
unlist(lapply(sb2, function(x) x$FinalTime))
## [1] 416 354 339 445 215
```

1.3.5.2 Consequences of order effects for cancer initiation Instead of focusing on different models for epistatic interactions, you might want to examine the consequences of order effects (Ortmann et al., 2015). You would proceed as above, but using models that differ by, say, the presence or absence of order effects. Details on their specification are provided in section 3.6. Here is one particular model (you would, of course, want to compare this to models without order effects or with other magnitudes and types of order effects):

```
## Order effects involving three genes.

## Genotype "D, M" has different fitness effects
## depending on whether M or D mutated first.
## Ditto for genotype "F, D, M".

## Meaning of specification: X > Y means
## that X is mutated before Y.


o3 <- allFitnessEffects(orderEffects = c(
                        "F > D > M" = -0.3,
                        "D > F > M" = 0.4,
                        "D > M > F" = 0.2,
                        "D > M"     = 0.1,
                        "M > D"     = 0.5))

## With the above specification, let's double check
## the fitness of the possible genotypes

(oeag <- evalAllGenotypes(o3, addwt = TRUE, order = TRUE))
##     Genotype Birth
## 1         WT  1.00
## 2          D  1.00
## 3          F  1.00
```

```
## 4        M  1.00
## 5     D > F  1.00
## 6     D > M  1.10
## 7     F > D  1.00
## 8     F > M  1.00
## 9     M > D  1.50
## 10    M > F  1.00
## 11 D > F > M  1.54
## 12 D > M > F  1.32
## 13 F > D > M  0.77
## 14 F > M > D  1.50
## 15 M > D > F  1.50
## 16 M > F > D  1.50
```

Now, run simulations and examine how frequently the runs do not end up in extinction. As above, for real, you would of course increase the number of total populations, the range of initial population sizes, mutation rate, etc:

```
## For reproducibility
set.seed(2)
RNGkind("L'Ecuyer-CMRG")


totalpops <- 5
soe1 <- oncoSimulPop(totalpops, o3, model = "Exp",
                     initSize = 500,
                     onlyCancer = FALSE,
                     mc.cores = 2, ## adapt to your hardware
                     seed = NULL) ## for reproducibility

## What proportion of the simulations do not end up extinct?
sum(summary(soe1)[, "TotalPopSize"] > 0)/totalpops
## [1] 0.4
```

As we just said, alternatively, to examine how long it takes to reach cancer you could run simulations with `onlyCancer = TRUE` and look at the value of `FinalTime` as we did above (section 1.3.4).

1.3.6 Simulating evolution with frequency-dependent fitness

The new frequency-dependent fitness funcionality allows users to run simulations in a different way, defining fitness (birth rates) as functions of clone's frequencies. We can thus model frequency-dependent selection, as well as predation and parasitism, cooperation and mutualism, and commensalism. See section 10 for further details and examples.

1.4 Trade-offs and what is OncoSimulR not well suited for

OncoSimulR is designed for complex fitness specifications and selection scenarios and uses forward-time simulations; the types of questions where OncoSimulR can be of help are discussed in sections 1.2 and 1.3 and running time and space consumption of OncoSimulR are addressed in section 2. You should be aware that **coalescent simulations**, sometimes also called backward-time simulations, are much more efficient for simulating neutral data as well as some special selection scenarios (Carvajal-Rodriguez, 2010; Hoban et al., 2011; Yuan et al., 2012).

In addition, since OncoSimulR allows you to specify fitness with arbitrary epistatic and order effects, as well as mutator effects, you need to learn the syntax of how to specify those effects and you might be paying a performance penalty if your scenario does not require this complexity. For instance, in the model of Beerenwinkel, Antal, et al. (2007), the fitness of a genotype depends only on the total number of drivers mutated, but not on which drivers are mutated (and, thus, not on the epistatic interactions nor the order of accumulation of the drivers). This means that the syntax for specifying that model could probably be a lot simpler (e.g., specify s per driver).

But it also means that code written for just that case could probably run much faster. First, because fitness evaluation is easier. Second, and possibly much more important, because what we need to keep track of leads to much simpler and economic structures: we do not need to keep track of clones (where two cells are regarded as different clones if they differ anywhere in their genotype), but only of clone types or clone classes as defined by the number of mutated drivers, and keeping track of clones can be expensive —see sections 2 and 18.2.

So for those cases where you do not need the full flexibility of OncoSimulR, special purpose software might be easier to use and faster to run. Of course, for some types of problems this special purpose software might not be available, though.

1.5 Random fitness landscapes, clonal competition, predictability, and the strong selection weak mutation (SSWM) regime

Many studies about evolutionary predictability (among other topics) focus on the strong selection, weak mutation regime, SSWM (Gillespie, 1984; Orr, 2002) (see overview in Krug (2019)). In this regime, mutations are rare (much smaller than the mutation rate times the population size) and selection is strong (much larger than 1/population size), so that the population consists of a single clone most of the time, and evolution proceeds by complete, successive clonal expansions of advantageous mutations.

We can easily simulate variations around these scenarios with OncoSimulR, moving away from the SSWM by increasing the population size, or changing the size of the fitness differences.

The examples below, not run for the sake of speed, play with population size and

fitness differences. To make sure we use a similar fitness landscape, we use the same simulated fitness landscape, scaled differently, so that the differences in fitness between mutants are increased or decreased while keeping their ranking identical (and, thus, having the same set of accessible and inaccessible genotypes and paths over the landscape).

If you run the code, you will see that as we increase population size we move further away from the SSWM: the population is no longer composed of a single clone most of the time.

Before running the examples, and to show the effects quantitatively, we define a simple wrapper to compute a few statistics.

```r
## oncoSimul object  -> measures of clonal interference
##    they are not averaged over time. One value for sampled time
clonal_interf_per_time <- function(x) {
    x <- x$pops.by.time
    y <- x[, -1, drop = FALSE]
    shannon <- apply(y, 1, OncoSimulR:::shannonI)
    tot <- rowSums(y)
    half_tot <- tot * 0.5
    five_p_tot <- tot * 0.05
    freq_most_freq <- apply(y/tot, 1, max)
    single_more_half <- rowSums(y > half_tot)
    ## whether more than 1 clone with more than 5% pop.
    how_many_gt_5p <- rowSums(y > five_p_tot)
    several_gt_5p <- (how_many_gt_5p > 1)
    return(cbind(shannon, ## Diversity of clones
                freq_most_freq, ## Frequency of the most freq. clone
                single_more_half, ## Any clone with a frequency > 50%?
                several_gt_5p, ## Are there more than 1 clones with
                                ## frequency > 5%?
                how_many_gt_5p ## How many clones are there with
                                ## frequency > 5%
                ))
}
```

```r
set.seed(1)
r7b <- rfitness(7, scale = c(1.2, 0, 1))

## Large pop sizes: clonal interference
(sr7b <- oncoSimulIndiv(allFitnessEffects(genotFitness = r7b),
                    model = "McFL",
                    mu = 1e-6,
                    onlyCancer = FALSE,
                    finalTime = 400,
                    initSize = 1e7,
                    keepEvery = 4,
```

```
                    detectionSize = 1e10))

plot(sr7b, show = "genotypes")

colMeans(clonal_interf_per_time(sr7b))

## Small pop sizes: a single clone most of the time
(sr7c <- oncoSimulIndiv(allFitnessEffects(genotFitness = r7b),
                    model = "McFL",
                    mu = 1e-6,
                    onlyCancer = FALSE,
                    finalTime = 60000,
                    initSize = 1e3,
                    keepEvery = 4,
                    detectionSize = 1e10))

plot(sr7c, show = "genotypes")

colMeans(clonal_interf_per_time(sr7c))



## Even smaller fitness differences, but large pop. sizes
set.seed(1); r7b2 <- rfitness(7, scale = c(1.05, 0, 1))

(sr7b2 <- oncoSimulIndiv(allFitnessEffects(genotFitness = r7b2),
                    model = "McFL",
                    mu = 1e-6,
                    onlyCancer = FALSE,
                    finalTime = 3500,
                    initSize = 1e7,
                    keepEvery = 4,
                    detectionSize = 1e10))
sr7b2
plot(sr7b2, show = "genotypes")
colMeans(clonal_interf_per_time(sr7b2))


## Increase pop size further
(sr7b3 <- oncoSimulIndiv(allFitnessEffects(genotFitness = r7b2),
                    model = "McFL",
                    mu = 1e-6,
                    onlyCancer = FALSE,
                    finalTime = 1500,
                    initSize = 1e8,
                    keepEvery = 4,
```

```
                            detectionSize = 1e10))
sr7b3
plot(sr7b3, show = "genotypes")
colMeans(clonal_interf_per_time(sr7b3))
```

1.6 Steps for using OncoSimulR

Using this package will often involve the following steps:

1. Specify fitness effects: sections 3 and 5.

2. Simulate cancer progression: section 6. You can simulate for a single individual or subject or for a set of subjects. You will need to:

   - Decide on a model. This basically amounts to choosing a model with exponential growth ("Exp" or "Bozic") or a model with carrying capacity ("McFL"). If exponential growth, you can choose whether the the effects of mutations operate on the death rate ("Bozic") or the birth rate ("Exp")[1].

   - Specify other parameters of the simulation. In particular, decide when to stop the simulation, mutation rates, etc.

   Of course, at least for initial playing around, you can use the defaults.

3. Sample from the simulated data and do something with those simulated data (e.g., fit an OT model to them, examine diversity or time until cancer, etc). Most of what you do with the data, however, is outside the scope of this package and this vignette.

Before anything else, let us load the package in case it was not yet loaded. We also explicitly load *graph* and *igraph* for the vignette to work (you do not need that for your usual interactive work). And I set the default color for vertices in igraph.

```
library(OncoSimulR)
library(graph)
library(igraph)
igraph_options(vertex.color = "SkyBlue2")
```

To be explicit, what version are we running?

```
packageVersion("OncoSimulR")
## [1] '4.7.1'
```

1.7 Two quick examples of fitness specifications

Following 1.6 we will run two very minimal examples. First a model with a few genes and **epistasis**:

```
## 1. Fitness effects: here we specify an
##    epistatic model with modules.
sa <- 0.1
sb <- -0.2
sab <- 0.25
```

---

[1]It is of course possible to do this with the carrying capacity (or gompertz-like) models, but there probably is little reason to do it. McFarland et al. (2013) discuss this has little effect on their results, for example. In addition, decreasing the death rate will more easily lead to numerical problems as shown in section 3.11.2.

```
sac <- -0.1
sbc <- 0.25
sv2 <- allFitnessEffects(epistasis = c("-A : B" = sb,
                                        "A : -B" = sa,
                                        "A : C" = sac,
                                        "A:B" = sab,
                                        "-A:B:C" = sbc),
                         geneToModule = c(
                             "A" = "a1, a2",
                             "B" = "b",
                             "C" = "c"),
                         drvNames = c("a1", "a2", "b", "c"))
evalAllGenotypes(sv2, addwt = TRUE)
##          Genotype Birth
## 1              WT 1.000
## 2              a1 1.100
## 3              a2 1.100
## 4               b 0.800
## 5               c 1.000
## 6          a1, a2 1.100
## 7           a1, b 1.250
## 8           a1, c 0.990
## 9           a2, b 1.250
## 10          a2, c 0.990
## 11           b, c 1.000
## 12      a1, a2, b 1.250
## 13      a1, a2, c 0.990
## 14       a1, b, c 1.125
## 15       a2, b, c 1.125
## 16 a1, a2, b, c 1.125
```

```
## 2. Simulate the data. Here we use the "McFL" model and set
##    explicitly parameters for mutation rate, initial size, size
##    of the population that will end the simulations, etc

RNGkind("Mersenne-Twister")
set.seed(983)
ep1 <- oncoSimulIndiv(sv2, model = "McFL",
                      mu = 5e-6,
                      sampleEvery = 0.025,
                      keepEvery = 0.5,
                      initSize = 2000,
                      finalTime = 3000,
                      onlyCancer = FALSE)
```

```
## 3. We will not analyze those data any further. We will only plot
## them.  For the sake of a small plot, we thin the data.
plot(ep1, show = "drivers", xlim = c(0, 1500),
     thinData = TRUE, thinData.keep = 0.5)
```
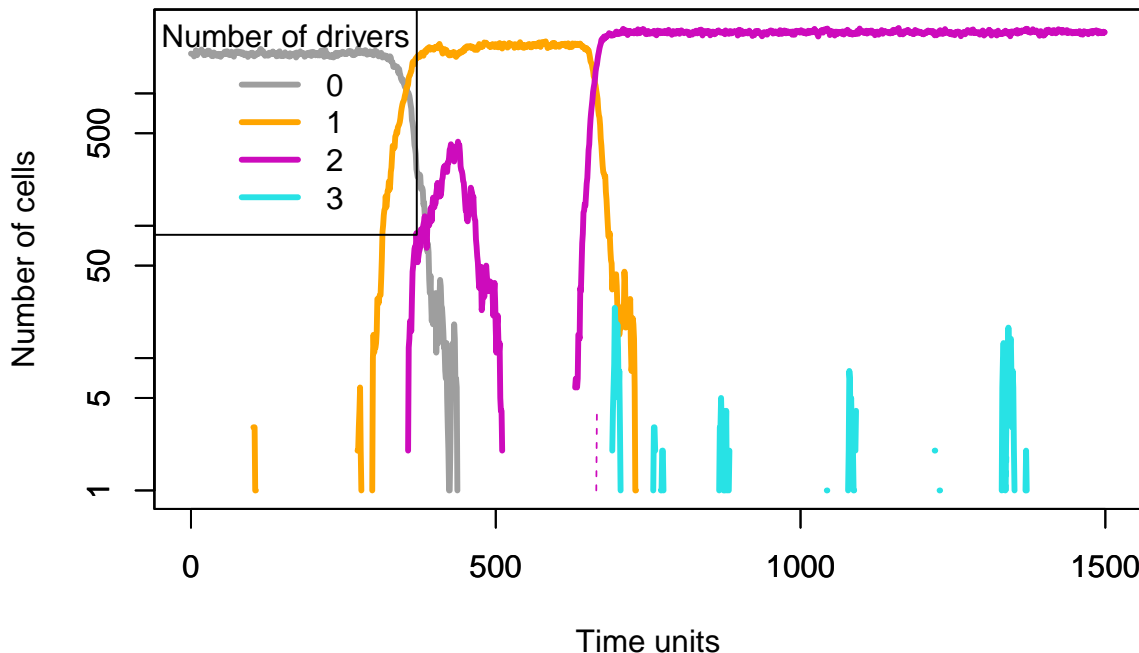

Figure 2: Plot of drivers of an epistasis simulation.

As a second example, we will use a model where we specify **restrictions in the order of accumulation of mutations using a DAG** with the pancreatic cancer poset in Gerstung, Eriksson, et al. (2011) (see more details in section 5.5):

```
## 1. Fitness effects:
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 = 0.1,
               sh = -0.9,
               typeDep = "MN"),
    drvNames = c("KRAS", "SMAD4", "CDNK2A", "TP53",
                 "MLL3", "TGFBR2", "PXDN"))

## Plot the DAG of the fitnessEffects object
plot(pancr)
```
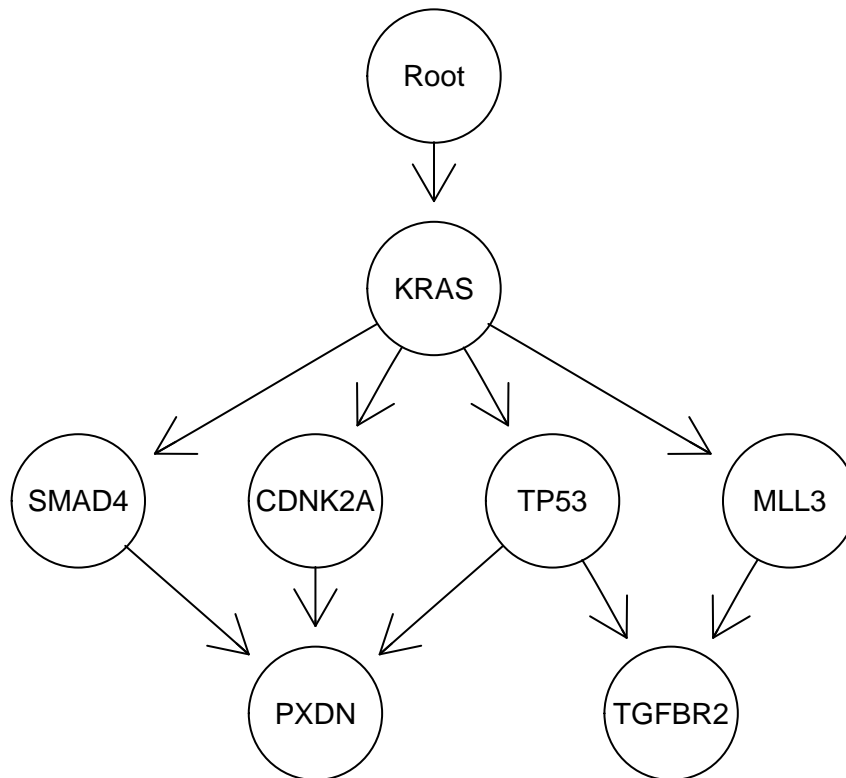
Figure 3: Plot of DAG corresponding to fitnessEffects object.

```
## 2. Simulate from it. We change several possible options.

set.seed(1) ## Fix the seed, so we can repeat it
## We set a small finalTime to speed up the vignette


ep2 <- oncoSimulIndiv(pancr, model = "McFL",
                      mu = 1e-6,
                      sampleEvery = 0.02,
                      keepEvery = 1,
                      initSize = 1000,
                      finalTime = 20000,
                      detectionDrivers = 3,
                      onlyCancer = FALSE)
```

```
## 3. What genotypes and drivers we get? And play with limits
##    to show only parts of the data. We also aggressively thin
##    the data.
par(cex = 0.7)
plot(ep2, show = "genotypes", xlim = c(500, 1800),
     ylim = c(0, 2400),
     thinData = TRUE, thinData.keep = 0.3)
```
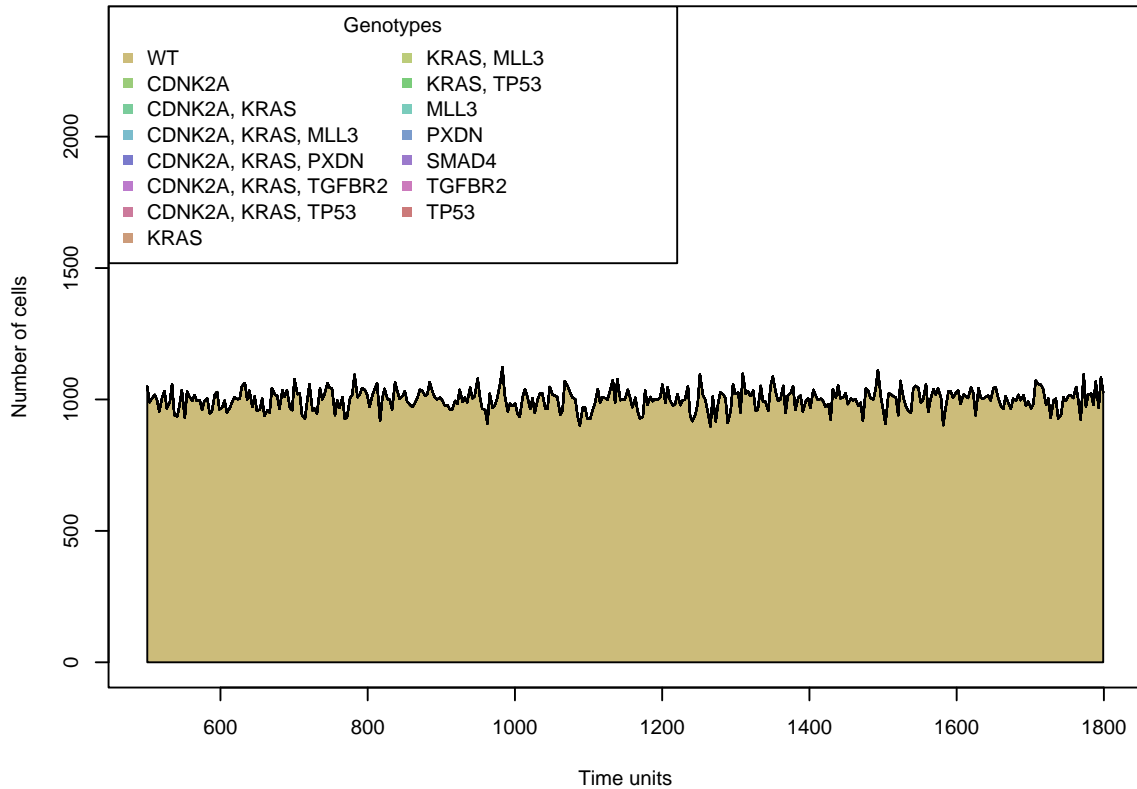
Figure 4: Plot of genotypes of a simulation from a DAG.

The rest of this vignette explores all of those functions and arguments in much more detail.

1.8 Citing OncoSimulR and other documentation

In R, you can do

```
citation("OncoSimulR")
## If you use OncoSimulR, please cite the OncoSimulR
## Bioinformatics paper.  OncoSimulR has been used in three
## large comparative studies of methods to infer restrictions
## in the order of accumulation of mutations (cancer
## progression models) published in PLoS Computational Biology,
## Bioinformatics and BMC Bioinformatics; you might want to
## cite those too, if appropriate, such as when referring to
## using evolutionary simulations to assess oncogenetic
## tree/cancer progression methods performance.
##
##   R Diaz-Uriarte. OncoSimulR: genetic simulation with
##   arbitrary epistasis and mutator genes in asexual
##   populations.  2017. Bioinformatics, 33, 1898--1899.
##   https://doi.org/10.1093/bioinformatics/btx077.
```

```
##
##   R Diaz-Uriarte and C. Vasallo. Every which way? On
##   predicting tumor evolution using cancer progression models
##   2019 PLoS Computational Biology
##   https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.100724
##
##   R Diaz-Uriarte. Cancer progression models and fitness
##   landscapes: a many-to-many relationship 2017
##   Bioinformatics.
##   https://academic.oup.com/bioinformatics/advance-article/doi/10.1093/bioinfor
##
##   R Diaz-Uriarte. Identifying restrictions in the order of
##   accumulation of mutations during tumor progression:
##   effects of passengers, evolutionary models, and sampling
##   2015. BMC Bioinformatics, 16(41).
##
## To see these entries in BibTeX format, use
## 'print(<citation>, bibtex=TRUE)', 'toBibtex(.)', or set
## 'options(citation.bibtex.max=999)'.
```

which will tell you how to cite the package. Please, do cite the Bionformatics paper if you use the package in publications.

This is the URL for the Bioinformatics paper: https://doi.org/10.1093/bioinformatics/btx077 (there is also an early preprint at bioRxiv, but it should now point to the Bioinformatics paper).

1.8.1 HTML and PDF versions of the vignette

A PDF version of this vignette is available from https://rdiaz02.github.io/OncoSimul/pdfs/OncoSimulR.pdf. And an HTML version from https://rdiaz02.github.io/OncoSimul/OncoSimulR.html. These files should correspond to the most recent, GitHub version, of the package (i.e., they might include changes not yet available from the BioConductor package). Beware that the PDF might have figures and R code that do not fit on the page, etc.

1.9 Testing, code coverage, and other examples

OncoSimulR includes more than 2000 tests that are run at every check cycle. These tests provide a code coverage of more than 90% including both the C++ and R code. Another set of over 500 long-running (several hours) tests can be run on demand (see directory '/tests/manual'). In addition to serving as test cases, some of that code also provides further examples of usage.

1.10 Versions

In this vignette and the documentation I often refer to version 1 (v.1) and version 2 of OncoSimulR. Version 1 is the version available up to, and including, BioConductor v. 3.1. Version 2 of OncoSimulR is available starting from BioConductor 3.2 (and, of course, available too from development versions of BioC). So, if you are using the current stable or development version of BioConductor, or you grab the sources from GitHub (https://github.com/rdiaz02/OncoSimul) you are using what we call version 2. **The functionality of version 1 has been removed.**

Version 3 (for BioConductor 3.13) made frequency dependent fitness available in the stable version.

Version 4 (BioConductor 3.16) introduces interventions and the possibility to specify, separately, birth and death (including frequency dependence).

2 Running time and space consumption of On-coSimulR

Time to complete the simulations and size of returned objects (space consumption) depend on several, interacting factors. The usual rule of "experiment before launching a large number of simulations" applies, but here we will walk through several cases to get a feeling for the major factors that affect speed and size. Many of the comments on this section need to use ideas discussed in other places of this document; if you read this section first, you might want to come back after reading the relevant parts.

Speed will depend on:

- Your hardware, of course.
- The evolutionary model.
- The granularity of how often you keep data (`keepEvery` argument). Note that the default, which is to keep as often as you sample (so that we preserve all history) can lead to slow execution times.
- The mutation rate, because higher mutation rates lead to more clones, and more clones means we need to iterate over, well, more clones, and keep larger data structures.
- The fitness specification: more complex fitness specifications tend to be slightly slower but specially different fitness specifications can have radically different effects on the evolutionary trajectories, accessibility of fast growing genotypes and, generally, the evolutionary dynamics.
- The stopping conditions (`detectionProb`, `detectionDrivers`, `detectionSize` arguments) and whether or not simulations are run until cancer is reached (`onlyCancer` argument).
- Most of the above factors can interact in complex ways.

Size of returned objects will depend on:

- Any factor that affects the number of clones tracked/returned, in particular: initial sizes and stopping conditions, mutation rate, and how often you keep data (the `keepEvery` argument can make a huge difference here).
- Whether or not you keep the complete genealogy of clones (this affects slightly the size of returned object, not speed).

In the sections that follow, we go over several cases to understand some of the main settings that affect running time (or execution time) and space consumption (the size of returned objects). It should be understood, however, that many of the examples shown below do not represent typical use cases of OncoSimulR and are used only to identify what and how affects running time and space consumption. As we will see in most examples in this vignette, typical use cases of OncoSimulR involve hundreds to thousands of genes on population sizes up to 10^5 to 10^7.

Note that most of the code in this section is not executed during the building of the vignette to keep vignette build time reasonable and prevent using huge amounts of RAM. All of the code, ready to be sourced and run, is available from the 'inst/miscell' directory (and the summary output from some of the benchmarks is available from

the 'miscell-files/vignette_bench_Rout' directory of the main OncoSimul repository at https://github.com/rdiaz02/OncoSimul).

2.1 Exp and McFL with "detectionProb" and pancreas example

To get familiar with some of they factors that affect time and size, we will use the fitness specification from section 1.7, with the `detectionProb` stopping mechanism (see 6.3.2). We will use the two main growth models (exponential and McFarland). Each model will be run with two settings of `keepEvery`. With `keepEvery = 1` (runs `exp1` and `mc1`), population samples are stored at time intervals of 1 (even if most of the clones in those samples later become extinct). With `keepEvery = NA` (runs `exp2` and `mc2`) no intermediate population samples are stored, so clones that become extinct at any sampling period are pruned and only the existing clones at the end of the simulation are returned (see details in 18.2.1).

Will run 100 simulations. The results I show are for a laptop with an 8-core Intel Xeon E3-1505M CPU, running Debian GNU/Linux (the results from these benchmarks are available as `data(benchmark_1)`).

```
## Specify fitness
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 = 0.1,
               sh = -0.9,
               typeDep = "MN"),
    drvNames = c("KRAS", "SMAD4", "CDNK2A", "TP53",
              "MLL3", "TGFBR2", "PXDN"))

Nindiv <- 100 ## Number of simulations run.
               ## Increase this number to decrease sampling variation

## keepEvery = 1
t_exp1 <- system.time(
    exp1 <- oncoSimulPop(Nindiv, pancr,
                        onlyCancer = TRUE,
                        detectionProb = "default",
                        detectionSize = NA,
                        detectionDrivers = NA,
                            finalTime = NA,
                            keepEvery = 1,
                            model = "Exp",
```

```
                                          mc.cores = 1))["elapsed"]/Nindiv


t_mc1 <- system.time(
    mc1 <- oncoSimulPop(Nindiv, pancr,
                        onlyCancer = TRUE,
                        detectionProb = "default",
                        detectionSize = NA,
                        detectionDrivers = NA,
                            finalTime = NA,
                            keepEvery = 1,
                            model = "McFL",
                            mc.cores = 1))["elapsed"]/Nindiv

## keepEvery = NA
t_exp2 <- system.time(
    exp2 <- oncoSimulPop(Nindiv, pancr,
                         onlyCancer = TRUE,
                         detectionProb = "default",
                         detectionSize = NA,
                             detectionDrivers = NA,
                             finalTime = NA,
                             keepEvery = NA,
                             model = "Exp",
                             mc.cores = 1))["elapsed"]/Nindiv


t_mc2 <- system.time(
    mc2 <- oncoSimulPop(Nindiv, pancr,
                        onlyCancer = TRUE,
                        detectionProb = "default",
                        detectionSize = NA,
                            detectionDrivers = NA,
                            finalTime = NA,
                            keepEvery = NA,
                            model = "McFL",
                            mc.cores = 1))["elapsed"]/Nindiv
```

We can obtain times, sizes of objects, and summaries of numbers of clones, iterations, and final times doing, for instance:

```
cat("\n\n\n t_exp1 = ", t_exp1, "\n")
object.size(exp1)/(Nindiv * 1024^2)
cat("\n\n")
summary(unlist(lapply(exp1, "[[", "NumClones")))
summary(unlist(lapply(exp1, "[[", "NumIter")))
```

```
summary(unlist(lapply(exp1, "[[", "FinalTime")))
summary(unlist(lapply(exp1, "[[", "TotalPopSize")))
```

The above runs yield the following:

Table 2: Benchmarks of Exp and McFL models using the default `detectionProb` with two settings of `keepEvery`.

	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, max.	keepEvery
exp1	0	0.04	2	254	252	1,058	11,046	1
mc1	0.74	3.9	12	816,331	20,406	696	979	1
exp2	0	0.01	1	296	294	1,021	21,884	NA
mc2	0.7	0.01	1	694,716	17,366	692	888	NA

The above table shows that a naive comparison (looking simply at execution time) might conclude that the McFL model is much, much slower than the Exp model. But that is not the complete story: using the `detectionProb` stopping mechanism (see 6.3.2) will lead to stopping the simulations very quickly in the exponential model because as soon as a clone with fitness > 1 appears it starts growing exponentially. In fact, we can see that the number of iterations and the final time are much smaller in the Exp than in the McFL model. We will elaborate on this point below (section 2.2.1), when we discuss the setting for `checkSizePEvery` (here left at its default value of 20): checking the exiting condition more often (smaller `checkSizePEvery`) would probably be justified here (notice also the very large final times) and would lead to a sharp decrease in number of iterations and, thus, running time.

This table also shows that the `keepEvery = NA` setting, which was in effect in simulations `exp2` and `mc2`, can make a difference especially for the McFL models, as seen by the median number of clones and the size of the returned object. Models `exp2` and `mc2` do not store any intermediate population samples so clones that become extinct at any sampling period are pruned and only the existing clones at the end of the simulation are returned. In contrast, models `exp1` and `mc1` store population samples at time intervals of 1 (`keepEvery = 1`), even if many of those clones eventually become extinct. We will return to this issue below as execution time and object size depend strongly on the number of clones tracked.

We can run the exponential model again modifying the arguments of the `detectionProb` mechanism; in two of the models below (`exp3` and `exp4`) no detection can take place unless populations are at least 100 times larger than the initial population size, and probability of detection is 0.1 with a population size 1,000 times larger than the initial one (`PDBaseline = 5e4, n2 = 5e5`). In the other two models (`exp5` and `exp6`), no detection can take place unless populations are at least 1,000 times larger than the initial population size, and probability of detection is 0.1 with a population size 100,000 times larger than the initial one (`PDBaseline = 5e5, n2 = 5e7`)[2]. In runs `exp3` and `exp5` we set `keepEvery = 1` and in runs `exp4` and `exp6` we set `keepEvery = NA`.

```
t_exp3 <- system.time(
    exp3 <- oncoSimulPop(Nindiv, pancr,
                         onlyCancer = TRUE,
                         detectionProb = c(PDBaseline = 5e4,
                                           p2 = 0.1, n2 = 5e5,
                                           checkSizePEvery = 20),
                         detectionSize = NA,
                         detectionDrivers = NA,
                         finalTime = NA,
                         keepEvery = 1,
                         model = "Exp",
                         mc.cores = 1))["elapsed"]/Nindiv
```

---

[2]Again, these are not necessarily reasonable or common settings. We are using them to understand what and how affects running time and space consumption.

```
t_exp4 <- system.time(
    exp4 <- oncoSimulPop(Nindiv, pancr,
                         onlyCancer = TRUE,
                         detectionProb = c(PDBaseline = 5e4,
                                           p2 = 0.1, n2 = 5e5,
                                           checkSizePEvery = 20),
                         detectionSize = NA,
                         detectionDrivers = NA,
                             finalTime = NA,
                             keepEvery = NA,
                             model = "Exp",
                             mc.cores = 1))["elapsed"]/Nindiv


t_exp5 <- system.time(
    exp5 <- oncoSimulPop(Nindiv, pancr,
                         onlyCancer = TRUE,
                         detectionProb = c(PDBaseline = 5e5,
                                           p2 = 0.1, n2 = 5e7),
                         detectionSize = NA,
                         detectionDrivers = NA,
                             finalTime = NA,
                             keepEvery = 1,
                             model = "Exp",
                             mc.cores = 1))["elapsed"]/Nindiv

t_exp6 <- system.time(
    exp6 <- oncoSimulPop(Nindiv, pancr,
                         onlyCancer = TRUE,
                         detectionProb = c(PDBaseline = 5e5,
                                           p2 = 0.1, n2 = 5e7),
                         detectionSize = NA,
                         detectionDrivers = NA,
                             finalTime = NA,
                             keepEvery = NA,
                             model = "Exp",
                             mc.cores = 1))["elapsed"]/Nindiv
```

Table 3: Benchmarks of Exp and McFL models modifying the default `detectionProb` with two settings of `keepEvery`.

	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, max.	keepEvery	PDBaseline	n2
exp3	0.01	0.41	14	2,754	1,890	6,798,358	2.7e+08	1	50,000	5e+05
exp4	0.01	0.02	8	2,730	2,090	7,443,812	1.7e+08	NA	50,000	5e+05
exp5	0.84	0.91	34	54,332	2,026	1.4e+09	4.2e+10	1	5e+05	5e+07
exp6	0.54	0.02	27	44,288	2,026	1.2e+09	3.3e+10	NA	5e+05	5e+07

As above, `keepEvery = NA` (in `exp4` and `exp6`) leads to much smaller object sizes and slightly smaller numbers of clones and execution times. Changing the exiting conditions (by changing `detectionProb` arguments) leads to large increases in number of iterations (in this case by factors of about 15x to 25x) and a corresponding increase in execution time as well as much larger population sizes (in some cases $> 10^{10}$).

In some of the runs of `exp5` and `exp6` we get the (recoverable) exception message from the C++ code: `Recoverable exception ti set to DBL_MIN. Rerunning`, which is related to those simulations reaching total population sizes $> 10^{10}$; we return to this below (section 2.4). You might also wonder why total and median population sizes are so large in these two runs, given the exiting conditions. One of the reasons is that we are using the default `checkSizePEvery = 20`, so the interval between successive checks of the exiting condition is large; this is discussed at greater length in section 2.2.1.

All the runs above used the default value `onlyCancer = TRUE`. This means that simulations will be repeated until the exiting conditions are reached (see details in section 6.3) and, therefore, any simulation that ends up in extinction will be repeated. This setting can thus have a large effect on the exponential models, because when the initial population size is not very large and we start from the wildtype, it is not uncommon for simulations to become extinct (when birth and death rates are equal and the population size is small, it is easy to reach extinction before a mutation in a gene that increases fitness occurs). But this is rarely the case in the McFarland model (unless we use really tiny initial population sizes) because of the dependency of death rate on total population size (see section 3.2.1).

The number of attempts until cancer was reached in the above models is shown in Table 4 (the values can be obtained from any of the above runs doing, for instance, `median(unlist(lapply(exp1, function(x) x$other$attemptsUsed)))`):

Table 4: Number of attempts until cancer.

	Attempts until Cancer, median	Attempts until Cancer, mean	Attempts until Cancer, max.	PDBaseline	n2
exp1	1	1.9	7	600	1,000
mc1	1	1	1	600	1,000
exp2	2	2.2	16	600	1,000
mc2	1	1	1	600	1,000
exp3	6	7.7	40	50,000	5e+05
exp4	6	8	39	50,000	5e+05
exp5	5	8.3	41	5e+05	5e+07
exp6	5	7.2	30	5e+05	5e+07

The McFL models finish in a single attempt. The exponential model simulations where we can exit with small population sizes (`exp1`, `exp2`) need many fewer attempts to reach cancer than those where large population sizes are required (`exp3` to `exp6`).

There is no relevant different among those last four, which is what we would expect: a population that has already reached a size of 50,000 cells from an initial population size of 500 is obviously a growing population where there is at least one mutant with positive fitness; thus, it unlikely to go extinct and therefore having to grow up to at least 500,000 will not significantly increase the risk of extinction.

We will now rerun all of the above models with argument `onlyCancer = FALSE`. The results are shown in Table 5 (note that the differences between this table and Table 2 for the McFL models are due only to sampling variation).

Table 5: Benchmarks of models in Table 2 and 3 when run with `onlyCancer = FALSE`.

	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, mean	Total Population Size, max.	keepEvery	PDBaseline	n2
exp1__noc	0.001	0.041	1.5	394	393	0	708	18,188	1	600	1,000
mc1__noc	0.69	3.9	12	673,910	16,846	692	700	983	1	600	1,000
exp2__noc	0.001	0.012	1	320	319	726	870	26,023	NA	600	1,000
mc2__noc	0.65	0.014	1	628,683	15,716	694	704	910	NA	600	1,000
exp3__noc	0.002	0.15	2	718	694	0	2,229,519	5.7e+07	1	50,000	5e+05
exp4__noc	0.002	0.013	0	600	599	0	3,122,765	1.3e+08	NA	50,000	5e+05
exp5__noc	0.17	0.22	3	848	777	0	5.9e+08	1.5e+10	1	5e+05	5e+07
exp6__noc	0.068	0.013	0	784	716	0	4.1e+08	1.3e+10	NA	5e+05	5e+07

Now most simulations under the exponential model end up in extinction, as seen by the median population size of 0 (but not all, as the mean and max. population size are clearly away from zero). Consequently, simulations under the exponential model are now faster (and the size of the average returned object is smaller). Of course, whether one should run simulations with `onlyCancer = TRUE` or `onlyCancer = FALSE` will depend on the question being asked (see, for example, section 1.3.5 for a question where we will naturally want to use `onlyCancer = FALSE`).

To make it easier to compare results with those of the next section, Table 6 shows all the runs so far.

Table 6: Benchmarks of all models in Tables 2, 3, and 5.

	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, mean	Total Population Size, max.	keepEvery	PDBaseline	n2	onlyCance
exp1	0.001	0.037	2	254	252	1,058	1,277	11,046	1	600	1,000	TRUE
mc1	0.74	3.9	12	816,331	20,406	696	702	979	1	600	1,000	TRUE
exp2	0.001	0.012	1	296	294	1,021	1,392	21,884	NA	600	1,000	TRUE
mc2	0.7	0.014	1	694,716	17,366	692	698	888	NA	600	1,000	TRUE
exp3	0.01	0.41	14	2,754	1,890	6,798,358	1.7e+07	2.7e+08	1	50,000	5e+05	TRUE
exp4	0.009	0.016	8	2,730	2,090	7,443,812	1.5e+07	1.7e+08	NA	50,000	5e+05	TRUE
exp5	0.84	0.91	34	54,332	2,026	1.4e+09	3.5e+09	4.2e+10	1	5e+05	5e+07	TRUE
exp6	0.54	0.021	27	44,288	2,026	1.2e+09	3.2e+09	3.3e+10	NA	5e+05	5e+07	TRUE
exp1__noc	0.001	0.041	1.5	394	393	0	708	18,188	1	600	1,000	FALSE
mc1__noc	0.69	3.9	12	673,910	16,846	692	700	983	1	600	1,000	FALSE
exp2__noc	0.001	0.012	1	320	319	726	870	26,023	NA	600	1,000	FALSE
mc2__noc	0.65	0.014	1	628,683	15,716	694	704	910	NA	600	1,000	FALSE
exp3__noc	0.002	0.15	2	718	694	0	2,229,519	5.7e+07	1	50,000	5e+05	FALSE
exp4__noc	0.002	0.013	0	600	599	0	3,122,765	1.3e+08	NA	50,000	5e+05	FALSE
exp5__noc	0.17	0.22	3	848	777	0	5.9e+08	1.5e+10	1	5e+05	5e+07	FALSE
exp6__noc	0.068	0.013	0	784	716	0	4.1e+08	1.3e+10	NA	5e+05	5e+07	FALSE

2.1.1 Changing fitness: $s = 0.1$ and $s = 0.05$

In the above fitness specification the fitness effect of each gene (when its restrictions are satisfied) is $s = 0.1$ (see section 3.2 for details). Here we rerun all the above benchmarks using $s = 0.05$ (the results from these benchmarks are available as `data(benchmark_1_0.05)`) and results are shown below in Table 7.

Table 7: Benchmarks of all models in Table 6 using $s = 0.05$ (instead of $s = 0.1$).

	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, mean	Total Population Size, max.	keepEvery	PDBaseline	n2	onlyCance
exp1	0.002	0.043	2	316	315	1,104	1,181	3,176	1	600	1,000	TRUE
mc1	1.7	11	17	2e+06	50,696	644	647	761	1	600	1,000	TRUE
exp2	0.001	0.012	1	274	273	1,129	1,281	7,608	NA	600	1,000	TRUE
mc2	1.6	0.016	1	1,615,197	40,376	644	651	772	NA	600	1,000	TRUE
exp3	0.012	0.63	15	3,995	2,919	3,798,540	5,892,376	4.5e+07	1	50,000	5e+05	TRUE
exp4	0.011	0.017	9	4,288	3,276	4,528,072	6,551,319	3.2e+07	NA	50,000	5e+05	TRUE
exp5	0.3	1.2	34	68,410	2,751	6.8e+08	1e+09	8.2e+09	1	5e+05	5e+07	TRUE
exp6	0.26	0.022	23	44,876	2,499	4.3e+08	8.9e+08	7.3e+09	NA	5e+05	5e+07	TRUE
exp1__noc	0.001	0.039	2	310	308	0	522	2,239	1	600	1,000	FALSE
mc1__noc	1.6	11	17	2e+06	50,776	638	643	757	1	600	1,000	FALSE
exp2__noc	0.001	0.012	0	340	336	0	599	3,994	NA	600	1,000	FALSE
mc2__noc	1.7	0.017	1	2,102,439	52,556	645	650	740	NA	600	1,000	FALSE
exp3__noc	0.002	0.11	2	618	615	0	150,978	6,093,498	1	50,000	5e+05	FALSE
exp4__noc	0.002	0.013	0	813	812	0	558,225	2.3e+07	NA	50,000	5e+05	FALSE
exp5__noc	0.031	0.23	3	917	914	0	1.1e+08	3.7e+09	1	5e+05	5e+07	FALSE
exp6__noc	0.046	0.013	0	628	610	0	1.7e+08	5.1e+09	NA	5e+05	5e+07	FALSE

As expected, having a smaller s leads to slower processes in most cases, since it takes longer to reach the exiting conditions sooner. Particularly noticeable are the runs for the McFL models (notice the increases in population size and number of iterations —see also below).

That is not the case, however, for `exp5` and `exp6` (and `exp5_noc` and `exp6_noc`). When running with $s = 0.05$ the simulations exit at a later time (see column "Final Time") but they exit with smaller population sizes. Here we have an interaction between sampling frequency, speed of growth of the population, mutation events and number of clones. In populations that grow much faster mutation events will happen more often (which will trigger further iterations of the algorithm); in addition, more new clones will be created, even if they only exist for short times and become extinct by the following sampling period (so they are not reflected in the `pops.by.time` matrix). These differences are proportionally larger the larger the rate of growth of the population. Thus, they are larger between, say, the `exp5` at $s = 0.1$ and $s = 0.05$ than between the `exp4` at the two different s: the `exp5` exit conditions can only be satisfied at much larger population sizes so at populations sizes when growth is much faster (recall we are dealing with exponential growth).

Recall also that with the default settings in `detectionProb`, we assess the exiting condition every 20 time periods (argument `checkSizePEvery`); this means that for fast growing populations, the increase in population size between successive checks of the exit conditions will be much larger (this phenomenon is also discussed in section 2.2.1).

Thus, what is happening in the `exp5` and `exp6` with $s = 0.1$ is that close to the time the exit conditions could be satisfied, they are growing very fast, accumulating mutants, and incurring in additional iterations. They exit sooner in terms of time periods, but they do much more work before arriving there.

The setting of `checkSizePEvery` is also having a huge effect on the McFL model simulations (the number of iterations is $> 10^6$). Even more than in the previous section, checking the exiting condition more often (smaller `checkSizePEvery`) would probably be justified here (notice also the very large final times) and would lead to a sharp decrease in number of iterations and, thus, running time.

The moral here is that in complex simulations like this (and most simulations are complex), the effects of some parameters (s in this case) might look counter-intuitive at first. Thus the need to "experiment before launching a large number of simulations".

2.2   Several "common use cases" runs

Let us now execute some simulations under more usual conditions. We will use seven different fitness specifications: the pancreas example, two random fitness landscapes, and four sets of independent genes (200 to 4000 genes) with fitness effects randomly drawn from exponential distributions:

```
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 = 0.1,
              sh = -0.9,
              typeDep = "MN"),
    drvNames = c("KRAS", "SMAD4", "CDNK2A", "TP53",
              "MLL3", "TGFBR2", "PXDN"))


## Random fitness landscape with 6 genes
## At least 50 accessible genotypes
rfl6 <- rfitness(6, min_accessible_genotypes = 50)
attributes(rfl6)$accessible_genotypes ## How many accessible
rf6 <- allFitnessEffects(genotFitness = rfl6)


## Random fitness landscape with 12 genes
## At least 200 accessible genotypes
rfl12 <- rfitness(12, min_accessible_genotypes = 200)
attributes(rfl12)$accessible_genotypes ## How many accessible
rf12 <- allFitnessEffects(genotFitness = rfl12)




## Independent genes; positive fitness from exponential distribution
## with mean around 0.1, and negative from exponential with mean
## around -0.02. Half of genes positive fitness effects, half
## negative.

ng <- 200 re_200 <- allFitnessEffects(noIntGenes = c(rexp(ng/2, 10),
-rexp(ng/2, 50)))

ng <- 500
re_500 <- allFitnessEffects(noIntGenes = c(rexp(ng/2, 10),
                                           -rexp(ng/2, 50)))


ng <- 2000
re_2000 <- allFitnessEffects(noIntGenes = c(rexp(ng/2, 10),
                                           -rexp(ng/2, 50)))
```

```
ng <- 4000
re_4000 <- allFitnessEffects(noIntGenes = c(rexp(ng/2, 10),
                                            -rexp(ng/2, 50)))
```

2.2.1 Common use cases, set 1.

We will use the Exp and the McFL models, run with different parameters. The script is provided as 'benchmark_2.R', under '/inst/miscell', with output in the 'miscell-files/vignette_bench_Rout' directory of the main OncoSimul repository at https://github.com/rdiaz02/OncoSimul. The data are available as data(benchmark_2).

For the Exp model the call will be

```
oncoSimulPop(Nindiv,
            fitness,
            detectionProb = NA,
            detectionSize = 1e6,
            initSize = 500,
            detectionDrivers = NA,
            keepPhylog = TRUE,
            model = "Exp",
            errorHitWallTime = FALSE,
            errorHitMaxTries = FALSE,
            finalTime = 5000,
            onlyCancer = FALSE,
            mc.cores = 1,
            sampleEvery = 0.5,
            keepEvery = 1)
```

And for McFL:
```
initSize <- 1000
oncoSimulPop(Nindiv,
              fitness,
               detectionProb = c(
                   PDBaseline = 1.4 * initSize,
                   n2 = 2 * initSize,
                   p2 = 0.1,
                   checkSizePEvery = 4),
              initSize = initSize,
              detectionSize = NA,
              detectionDrivers = NA,
              keepPhylog = TRUE,
              model = "McFL",
              errorHitWallTime = FALSE,
              errorHitMaxTries = FALSE,
              finalTime = 5000,
```

```
            max.wall.time = 10,
            onlyCancer = FALSE,
            mc.cores = 1,
            keepEvery = 1)
```

For the exponential model we will stop simulations when populations have $> 10^6$ cells (simulations start from 500 cells). For the McFarland model we will use the `detectionProb` mechanism (see section 6.3.2 for details); we could have used as stopping mechanism `detectionSize = 2 * initSize` (which would be basically equivalent to reaching cancer, as argued in (McFarland et al., 2013)) but we want to provide further examples under the `detectionProb` mechanism. We will start from 1000 cells, not 500 (starting from 1000 we almost always reach cancer in a single run).

Why not use the `detectionProb` mechanism with the `Exp` models? Because it can be hard to intuitively understand what are reasonable settings for the parameters of the `detectionProb` mechanism when used in a population that is growing exponentially, especially if different genes have very different effects on fitness. Moreover, we are using fitness specifications that are very different (compare the fitness landscape of six genes, the pancreas specification, and the fitness specification with 4000 genes with fitness effects drawn from an exponential distribution —`re_4000`). In contrast, the `detectionProb` mechanism might be simpler to reason about in a population that is growing under a model of carrying capacity with possibly large periods of stasis. Let us emphasize that it is not that the `detectionProb` mechanism does not make sense with the Exp model; it is simply that the parameters might need finer adjustment for them to make sense, and in these benchmarks we are dealing with widely different fitness specifications.

Note also that we specify `checkSizePEvery = 4` (instead of the default, which is 20). Why? Because the fitness specifications where fitness effects are drawn from exponential distributions (`re_200` to `re_4000` above) include many genes (well, up to 4000) some of them with possibly very large effects. In these conditions, simulations can run very fast in the sense of "units of time". If we check exiting conditions every 20 units the population could have increased its size several orders of magnitude in between checks (this is also discussed in sections 2.1.1 and 6.3.2). You can verify this by running the script with other settings for `checkSizePEvery` (and being aware that large settings might require you to wait for a long time). To ensure that populations have really grown, we have increased the setting of `PDBaseline` so that no simulation can be considered for stopping unless its size is 1.4 times larger than `initSize`.

In all cases we use `keepEvery = 1` and `keepPhylog = TRUE` (so we store the population sizes of all clones every 1 time unit and we keep the complete genealogy of clones). Finally, we run all models with `errorHitWallTime = FALSE` and `errorHitMaxTries = FALSE` so that we can see results even if stopping conditions are not met.

The results of the benchmarks, using 100 individual simulations, are shown in Table 8.

Table 8: Benchmarks under some common use cases, set 1.

Model Fitness	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, mean	Total Population Size, max.
Exp pancr	0.002	0.12	3	1,397	697	0	164,222	1,053,299
McFL pancr	0.12	0.56	8	2e+05	5,000	1,037	1,144	1,938
Exp rf6	0.002	0.064	6	783	391	1e+06	594,899	1,309,497
McFL rf6	0.019	0.071	3	23,297	582	1,884	1,975	4,636
Exp rf12	0.01	0.13	4	1,178	542	0	287,669	1,059,141
McFL rf12	0.14	0.82	18	2e+05	5,000	1,252	1,295	1,695
Exp re_200	0.013	0.67	230	1,185	223	1,060,944	859,606	1,536,242
McFL re_200	0.018	0.22	47	9,679	240	2,166	2,973	29,301
Exp re_500	0.09	2.7	771	2,732	152	1,068,732	959,026	1,285,522
McFL re_500	0.024	0.44	91	7,056	172	2,148	2,578	8,234
Exp re_2000	0.91	29	3,376	7,412	70	1,163,990	1,143,041	1,741,492
McFL re_2000	0.031	1.9	186	3,546	80	2,870	3,704	13,248
Exp re_4000	3.3	113	7,088	12,216	52	1,217,568	1,309,185	2,713,200
McFL re_4000	0.063	6.5	326	2,731	52	4,592	13,601	729,611

In most cases, simulations run reasonably fast (under 0.1 seconds per individual simulation) and the returned objects are small. I will only focus on a few cases.

The McFL model with random fitness landscape `rf12` and with `pancr` does not satisfy the conditions of `detectionProb` in most cases: its median final time is 5000, which was the maximum final time specified. This suggests that the fitness landscape is such that it is unlikely that we will reach population sizes > 1400 (remember we the setting for `PDBaseline`) before 5000 time units. There is nothing particular about using a fitness landscape of 12 genes and other runs in other 12-gene random fitness landscapes do not show this pattern. However, complex fitness landscapes might be such that genotypes of high fitness (those that allow reaching a large population size quickly) are not easily accessible[3] so reaching them might take a long time. This does not affect the exponential model in the same way because, well, because there is exponential growth in that model: any genotype with fitness > 1 will grow exponentially (of course, at possibly very different rates). You might want to play with the script and modify the call to `rfitness` (using different values of `reference` and `c`, for instance) to have simpler paths to a maximum or modify the call to `oncoSimulPop` (with, say, `finalTime` to much larger values). Some of these issues are related to more general questions about fitness landscapes and accessibility (see section 1.3.2 and references therein).

You could also set `onlyCancer = TRUE`. This might make sense if you are interested in only seeing simulations that "reach cancer" (where "reach cancer" means reaching a state you define as a function of population size or drivers). However, if you are exploring fitness landscapes, `onlyCancer = TRUE` might not always be reasonable as reaching a particular population size, for instance, might just not be possible under some fitness landscapes (this phenomenon is of course not restricted to random fitness landscapes —see also section 2.3.3).

As we anticipated above, the `detectionProb` mechanism has to be used with care: some of the simulations run in very short "time units", such as those for the fitness specifications with 2000 and 4000 genes. Having used a `checkSizePEvery = 20` probably would not have made sense.

Finally, it is interesting that in the cases examined here, the two slowest running simulations are from "Exp", with fitnesses `re_2000` and `re_4000` (and the third slowest is also Exp, under `re_500`). These are also the cases with the largest number of clones. Why? In the "Exp" model there is no competition, and fitness specifications `re_2000` and `re_4000` have genomes with many genes with positive fitness contributions. It is thus very easy to obtain, from the wildtype ancestor, a large number of clones all of which have birth rates > 1 and, thus, clones that are unlikely to become extinct.

---

[3]By easily accessible I mean that there are many, preferably short, paths of non-decreasing fitness from the wildtype to this genotype. See definitions and discussion in, e.g., Franke et al. (2011).

2.2.2 Common use cases, set 2.

We will now rerun the simulations above changing the following:

- `finalTime` set to 25000.
- `onlyCancer` set to TRUE.
- The "Exp" models will stop when population size $> 10^5$.

This is in script 'benchmark_3.R', under '/inst/miscell', with output in the 'miscell-files/vignette_bench_Rout' directory of the main OncoSimul repository at https://github.com/rdiaz02/OncoSimul. The data are available as `data(benchmark_3)`.

Table 9: Benchmarks under some common use cases, set 2.

Model Fitness	Elapsed Time, average per simulation (s)	Object Size, average per simulation (MB)	Number of Clones, median	Number of Iterations, median	Final Time, median	Total Population Size, median	Total Population Size, mean	Total Population Size, max.
Exp pancr	0.012	0.32	10	3,480	1,718	1e+05	1e+05	108,805
McFL pancr	0.41	1.7	14	4e+05	9,955	1,561	1,555	1,772
Exp rf6	0.003	0.058	4	866	430	107,492	109,774	135,257
McFL rf6	0.033	0.12	4	35,216	880	2,003	2,010	3,299
Exp rf12	0.012	0.098	9	1,138	561	1e+05	1e+05	112,038
McFL rf12	0.17	0.76	16	1e+05	2,511	1,486	1,512	1,732
Exp re_200	0.004	0.39	106	723	252	1e+05	105,586	122,338
McFL re_200	0.026	0.33	61	13,484	335	1,830	2,049	3,702
Exp re_500	0.007	0.61	168	490	117	110,311	112,675	134,860
McFL re_500	0.018	0.33	70	5,157	126	2,524	3,455	19,899
Exp re_2000	0.046	5.7	651	1,078	68	106,340	109,081	153,146
McFL re_2000	0.029	1.8	186	3,444	80	2,837	4,009	37,863
Exp re_4000	0.1	19	1,140	1,722	51	111,256	113,499	168,958
McFL re_4000	0.057	6.7	325	3,081	60	3,955	8,892	265,183

Since we increased the maximum final time and forced runs to "reach cancer" the McFL run with the pancreas fitness specification takes a bit longer because it also has to do a larger number of iterations. Interestingly, notice that the median final time is close to 10000, so the runs in 2.2.1 with maximum final time of 5000 would have had a hard time finishing with `onlyCancer = TRUE`.

Forcing simulations to "reach cancer" and just random differences between the random fitness landscape also affect the McFL run under `rf12`: final time is below 5000 and the median number of iterations is about half of what was above.

Finally, by stopping the Exp simulations at 10^5, simulations with `re_2000` and `re_4000` finish now in much shorter times (but they still take longer than their McFL counterparts) and the number of clones created is much smaller.

2.3   Can we use a large number of genes?

Yes. In fact, in OncoSimulR there is no pre-set limit on genome size. However, large numbers of genes can lead to unacceptably large returned object sizes and/or running time. We discuss several examples next that illustrate some of the major issues to consider. Another example with 50,000 genes is shown in section 6.5.3.

We have seen in 2.1 and 2.2.1 that for the Exp model, benchmark results using `detectionProb` require a lot of care and can be misleading. Here, we will fix initial population sizes (to 500) and all final population sizes will be set to $\geq 10^6$. In addition, to avoid the confounding factor of the `onlyCancer = TRUE` argument, we will set it to FALSE, so we measure directly the time of individual runs.

2.3.1   Exponential model with 10,000 and 50,000 genes

2.3.1.1   Exponential, 10,000 genes, example 1   We will start with 10000 genes and an exponential model, where we stop when the population grows over 10^6 individuals:

```
ng <- 10000
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng/2),
                                      rep(-0.1, ng/2)))


t_e_10000 <- system.time(
    e_10000 <- oncoSimulPop(5, u, model = "Exp", mu = 1e-7,
                            detectionSize = 1e6,
                            detectionDrivers = NA,
                            detectionProb = NA,
                            keepPhylog = TRUE,
                            onlyCancer = FALSE,
                            mutationPropGrowth = TRUE,
                            mc.cores = 1))

t_e_10000
##    user  system elapsed
```

```
##    4.368    0.196    4.566

summary(e_10000)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      5017      1180528       415116       143    7547
## 2      3726      1052061       603612       131    5746
## 3      4532      1100721       259510       132    6674
## 4      4150      1283115       829728        99    6646
## 5      4430      1139185       545958       146    6748

print(object.size(e_10000), units = "MB")
## 863.9 Mb
```

Each simulation takes about 1 second but note that the number of clones for most simulations is already over 4000 and that the size of the returned object is close to 1 GB (a more detailed explanation of where this 1 GB comes from is deferred until section 2.3.1.6).

2.3.1.2 Exponential, 10,000 genes, example 2 We can decrease the size of the returned object if we use the `keepEvery = NA` argument (this setting was explained in detail in section 2.1):

```
t_e_10000b <- system.time(
    e_10000b <- oncoSimulPop(5,
                             u,
                             model = "Exp",
                             mu = 1e-7,
                             detectionSize = 1e6,
                             detectionDrivers = NA,
                             detectionProb = NA,
                             keepPhylog = TRUE,
                             onlyCancer = FALSE,
                             keepEvery = NA,
                             mutationPropGrowth = TRUE,
                             mc.cores = 1
                             ))
```

```
t_e_10000b
##    user  system elapsed
##   5.484   0.100   5.585

summary(e_10000b)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      2465      1305094       727989        91    6447
## 2      2362      1070225       400329       204    8345
## 3      2530      1121164       436721       135    8697
## 4      2593      1206293       664494       125    8149
```

```
## 5        2655       1186994        327835      191    8572
```

```
print(object.size(e_10000b), units = "MB")
## 488.3 Mb
```

2.3.1.3 Exponential, 50,000 genes, example 1 Let's use 50,000 genes.
To keep object sizes reasonable we use `keepEvery = NA`. For now, we also set
`mutationPropGrowth = FALSE` so that the mutation rate does not become really
large in clones with many mutations but, of course, whether or not this is a
reasonable decision depends on the problem; see also below.

```
ng <- 50000
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng/2),
                                      rep(-0.1, ng/2)))
t_e_50000 <- system.time(
    e_50000 <- oncoSimulPop(5,
                            u,
                            model = "Exp",
                            mu = 1e-7,
                            detectionSize = 1e6,
                            detectionDrivers = NA,
                            detectionProb = NA,
                            keepPhylog = TRUE,
                            onlyCancer = FALSE,
                            keepEvery = NA,
                            mutationPropGrowth = FALSE,
                            mc.cores = 1
                            ))


t_e_50000
##    user  system elapsed
##  44.192   1.684  45.891


summary(e_50000)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      7367      1009949       335455     75.00   18214
## 2      8123      1302324       488469     63.65   17379
## 3      8408      1127261       270690     72.57   21144
## 4      8274      1138513       318152     80.59   20994
## 5      7520      1073131       690814     70.00   18569


print(object.size(e_50000), units = "MB")
## 7598.6 Mb
```

Of course, simulations now take longer and the size of the returned object is over 7
GB (we are keeping more than 7,000 clones, even if when we prune all those that

went extinct).

2.3.1.4 Exponential, 50,000 genes, example 2 What if we had not pruned?

```
ng <- 50000
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng/2),
                                      rep(-0.1, ng/2)))
t_e_50000np <- system.time(
    e_50000np <- oncoSimulPop(5,
                              u,
                              model = "Exp",
                              mu = 1e-7,
                              detectionSize = 1e6,
                              detectionDrivers = NA,
                              detectionProb = NA,
                              keepPhylog = TRUE,
                              onlyCancer = FALSE,
                              keepEvery = 1,
                              mutationPropGrowth = FALSE,
                              mc.cores = 1
                              ))

t_e_50000np
##    user  system elapsed
## 42.316   2.764  45.079

summary(e_50000np)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1     13406      1027949       410074     71.97   19469
## 2     12469      1071325       291852     66.00   17834
## 3     11821      1089834       245720     90.00   16711
## 4     14008      1165168       505607     77.61   19675
## 5     14759      1074621       205954     87.68   20597

print(object.size(e_50000np), units = "MB")
## 12748.4 Mb
```

The main effect is not on execution time but on object size (it has grown by 5 GB). We are tracking more than 10,000 clones.

2.3.1.5 Exponential, 50,000 genes, example 3 What about the `mutationPropGrowth` setting? We will rerun the example in 2.3.1.3 leaving `keepEvery = NA` but with the default `mutationPropGrowth`:

```
ng <- 50000
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng/2),
                                      rep(-0.1, ng/2)))
```

```
t_e_50000c <- system.time(
    e_50000c <- oncoSimulPop(5,
                             u,
                             model = "Exp",
                             mu = 1e-7,
                             detectionSize = 1e6,
                             detectionDrivers = NA,
                             detectionProb = NA,
                             keepPhylog = TRUE,
                             onlyCancer = FALSE,
                             keepEvery = NA,
                             mutationPropGrowth = TRUE,
                             mc.cores = 1
                             ))

t_e_50000c
##    user  system elapsed
## 84.228   2.416  86.665

summary(e_50000c)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1     11178      1241970       344479     84.74   27137
## 2     12820      1307086       203544     91.94   33448
## 3     10592      1126091       161057     83.81   26064
## 4     11883      1351114       148986     65.68   25396
## 5     10518      1101392       253523     99.79   26082

print(object.size(e_50000c), units = "MB")
## 10904.9 Mb
```

As expected (because the mutation rate per unit time is increasing in the fastest growing clones), we have many more clones, larger objects, and longer times of execution here: we almost double the time and the size of the object increases by almost 3 GB.

What about larger population sizes or larger mutation rates? The number of clones starts growing fast, which means much slower execution times and much larger returned objects (see also the examples below).

2.3.1.6 Interlude: where is that 1 GB coming from? In section 2.3.1.1 we have seen an apparently innocuous simulation producing a returned object of almost 1 GB. Where is that coming from? It means that each simulation produced almost 200 MB of output.

Let us look at one simulation in more detail:

```
r1 <- oncoSimulIndiv(u,
                     model = "Exp",
                     mu = 1e-7,
                     detectionSize = 1e6,
                     detectionDrivers = NA,
                     detectionProb = NA,
                     keepPhylog = TRUE,
                     onlyCancer = FALSE,
                     mutationPropGrowth = TRUE
                     )
summary(r1)[c(1, 8)]
##   NumClones  FinalTime
## 1      3887        345

print(object.size(r1), units = "MB")
## 160 Mb

## Size of the two largest objects inside:
sizes <- lapply(r1, function(x) object.size(x)/(1024^2))
sort(unlist(sizes), decreasing = TRUE)[1:2]
## Genotypes pops.by.time
##        148.28         10.26


dim(r1$Genotypes)
## [1] 10000  3887
```

The above shows the reason: the `Genotypes` matrix is a 10,000 by 3,887 integer matrix (with a 0 and 1 indicating not-mutated/mutated for each gene in each genotype) and in R integers use 4 bytes each. The `pops.by.time` matrix is 346 by 3,888 (the 1 in $346 = 345 + 1$ comes from starting at 0 and going up to the final time, both included; the 1 in $3888 = 3887 + 1$ is from the column of time) double matrix and doubles use 8 bytes[4].

2.3.2 McFarland model with 50,000 genes; the effect of `keepEvery`

We show an example of McFarland's model with 50,000 genes in section 6.5.3. We will show here a few more examples with those many genes but with a different fitness specification and changing several other settings.

2.3.2.1 McFarland, 50,000 genes, example 1   Let's start with `mutationPropGrowth` = FALSE and `keepEvery = NA`. Simulations end when population size $\geq 10^6$.

```
ng <- 50000
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng/2),
```

---

[4]These matrices do not exist during most of the execution of the C++ code; they are generated right before returning from the C++ code.

```
                                        rep(-0.1, ng/2)))

t_mc_50000_nmpg <- system.time(
    mc_50000_nmpg <- oncoSimulPop(5,
                                  u,
                                  model = "McFL",
                                  mu = 1e-7,
                                  detectionSize = 1e6,
                                  detectionDrivers = NA,
                                  detectionProb = NA,
                                  keepPhylog = TRUE,
                                  onlyCancer = FALSE,
                                  keepEvery = NA,
                                  mutationPropGrowth = FALSE,
                                  mc.cores = 1
                                  ))
t_mc_50000_nmpg
##   user  system elapsed
##  30.46    0.54   31.01


summary(mc_50000_nmpg)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      1902      1002528       582752     284.2   31137
## 2      2159      1002679       404858     274.8   36905
## 3      2247      1002722       185678     334.5   42429
## 4      2038      1009606       493574     218.4   32519
## 5      2222      1004661       162628     291.0   38470

print(object.size(mc_50000_nmpg), units = "MB")
## 2057.6 Mb
```

We are already dealing with 2000 clones.

2.3.2.2 McFarland, 50,000 genes, example 2 Setting `keepEvery = 1` (i.e., keeping track of clones with an interval of 1):

```
t_mc_50000_nmpg_k <- system.time(
    mc_50000_nmpg_k <- oncoSimulPop(5,
                                    u,
                                    model = "McFL",
                                    mu = 1e-7,
                                    detectionSize = 1e6,
                                    detectionDrivers = NA,
                                    detectionProb = NA,
                                    keepPhylog = TRUE,
```

```
                                         onlyCancer = FALSE,
                                         keepEvery = 1,
                                         mutationPropGrowth = FALSE,
                                         mc.cores = 1
                                         ))

t_mc_50000_nmpg_k
##    user  system elapsed
##  30.000   1.712  31.714

summary(mc_50000_nmpg_k)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      8779      1000223       136453     306.7   38102
## 2      7442      1006563       428150     345.3   35139
## 3      8710      1003509       224543     252.3   35659
## 4      8554      1002537       103889     273.7   36783
## 5      8233      1003171       263005     301.8   35236

print(object.size(mc_50000_nmpg_k), units = "MB")
## 8101.4 Mb
```

Computing time increases slightly but the major effect is seen on the size of the returned object, that increases by a factor of about 4x, up to 8 GB, corresponding to the increase in about 4x in the number of clones being tracked (see details of where the size of this object comes from in section 2.3.1.6).

2.3.2.3   McFarland, 50,000 genes, example 3   We will set keepEvery = NA again, but we will now increase detection size by a factor of 3 (so we stop when total population size becomes $\geq 3*10^6$).

```
ng <- 50000
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng/2),
                                      rep(-0.1, ng/2)))

t_mc_50000_nmpg_3e6 <- system.time(
    mc_50000_nmpg_3e6 <- oncoSimulPop(5,
                                      u,
                                      model = "McFL",
                                      mu = 1e-7,
                                      detectionSize = 3e6,
                                      detectionDrivers = NA,
                                      detectionProb = NA,
                                      keepPhylog = TRUE,
                                      onlyCancer = FALSE,
                                      keepEvery = NA,
                                      mutationPropGrowth = FALSE,
                                      mc.cores = 1
```

```
                                                  ))
t_mc_50000_nmpg_3e6
##    user  system elapsed
##  77.240   1.064  78.308


summary(mc_50000_nmpg_3e6)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      5487      3019083       836793     304.5   65121
## 2      4812      3011816       789146     286.3   53087
## 3      4463      3016896      1970957     236.6   45918
## 4      5045      3028142       956026     360.3   63464
## 5      4791      3029720       916692     358.1   55012


print(object.size(mc_50000_nmpg_3e6), units = "MB")
## 4759.3 Mb
```

Compared with the first run (2.3.2.1) we have approximately doubled computing time, number of iterations, number of clones, and object size.

2.3.2.4   McFarland, 50,000 genes, example 4 Let us use the same `detectionSize = 1e6` as in the first example (2.3.2.1), but with 5x the mutation rate:

```
t_mc_50000_nmpg_5mu <- system.time(
    mc_50000_nmpg_5mu <- oncoSimulPop(5,
                                      u,
                                      model = "McFL",
                                      mu = 5e-7,
                                      detectionSize = 1e6,
                                      detectionDrivers = NA,
                                      detectionProb = NA,
                                      keepPhylog = TRUE,
                                      onlyCancer = FALSE,
                                      keepEvery = NA,
                                      mutationPropGrowth = FALSE,
                                      mc.cores = 1
                                      ))


t_mc_50000_nmpg_5mu
##    user  system elapsed
## 167.332   1.796 169.167


summary(mc_50000_nmpg_5mu)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      7963      1004415       408352     99.03   57548
## 2      8905      1010751       120155    130.30   74738
```

```
## 3      8194      1005465      274661      96.98    58546
## 4      9053      1014049      119943     112.23    75379
## 5      8982      1011817       95047      99.95    76757

print(object.size(mc_50000_nmpg_5mu), units = "MB")
## 8314.4 Mb
```

The number of clones we are tracking is about 4x the number of clones of the first
example (2.3.2.1), and roughly similar to the number of clones of the second example
(2.3.2.2), and size of the returned object is similar to that of the second example. But
computing time has increased by a factor of about 5x and iterations have increased
by a factor of about 2x. Iterations increase because mutation is more frequent; in
addition, at each sampling period each iteration needs to do more work as it needs
to loop over a larger number of clones and this larger number includes clones that
are not shown here, because they are pruned (they are extinct by the time we exit
the simulation —again, pruning is discussed with further details in 18.2.1).

2.3.2.5 McFarland, 50,000 genes, example 5   Now let's run the above example
but with keepEvery = 1:

```
t_mc_50000_nmpg_5mu_k <- system.time(
    mc_50000_nmpg_5mu_k <- oncoSimulPop(5,
                                        u,
                                        model = "McFL",
                                        mu = 5e-7,
                                        detectionSize = 1e6,
                                        detectionDrivers = NA,
                                        detectionProb = NA,
                                        keepPhylog = TRUE,
                                        onlyCancer = FALSE,
                                        keepEvery = 1,
                                        mutationPropGrowth = FALSE,
                                        mc.cores = 1
                                        ))

t_mc_50000_nmpg_5mu_k
##    user  system elapsed
## 174.404   5.068 179.481

summary(mc_50000_nmpg_5mu_k)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1     25294      1001597       102766     123.4   74524
## 2     23766      1006679       223010     124.3   71808
## 3     21755      1001379       203638     114.8   62609
## 4     24889      1012103       161003     119.3   75031
## 5     21844      1002927       255388     108.8   64556
```

70

```
print(object.size(mc_50000_nmpg_5mu_k), units = "MB")
## 22645.8 Mb
```

We have already seen these effects before in section 2.3.2.2: using `keepEvery = 1` leads to a slight increase in execution time. What is really affected is the size of the returned object which increases by a factor of about 3x (and is now over 20GB). That 3x corresponds, of course, to the increase in the number of clones being tracked (now over 20,000). This, by the way, also allows us to understand the comment above, where we said that in these two cases (where we have increased mutation rate) at each iteration we need to do more work as at every update of the population the algorithm needs to loop over a much larger number of clones (even if many of those are eventually pruned).

2.3.2.6   McFarland, 50,000 genes, example 6   Finally, we will run the example in section 2.3.2.1 with the default of `mutationPropGrowth = TRUE`:

```
t_mc_50000 <- system.time(
    mc_50000 <- oncoSimulPop(5,
                             u,
                             model = "McFL",
                             mu = 1e-7,
                             detectionSize = 1e6,
                             detectionDrivers = NA,
                             detectionProb = NA,
                             keepPhylog = TRUE,
                             onlyCancer = FALSE,
                             keepEvery = NA,
                             mutationPropGrowth = TRUE,
                             mc.cores = 1
                             ))

t_mc_50000
##     user   system elapsed
## 303.352    2.808 306.223


summary(mc_50000)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1     13928      1010815       219814     210.9   91255
## 2     12243      1003267       214189     178.1   67673
## 3     13880      1014131       124354     161.4   88322
## 4     14104      1012941        75521     205.7   98583
## 5     12428      1005594       232603     167.4   70359


print(object.size(mc_50000), units = "MB")
## 12816.6 Mb
```

Note the huge increase in computing time (related of course to the huge increase

71

in number of iterations) and in the size of the returned object: we have gone from having to track about 2000 clones to tracking over 12000 clones even when we prune all clones without descendants.

2.3.3 Examples with $s = 0.05$

A script with the above runs but using $s = 0.05$ instead of $s = 0.1$ is available from the repository ('miscell-files/vignette_bench_Rout/large_num_genes_0.05.Rout'). I will single out a couple of cases here.

First, we repeat the run shown in section 2.3.2.5:

```
t_mc_50000_nmpg_5mu_k <- system.time(
    mc_50000_nmpg_5mu_k <- oncoSimulPop(2,
                                        u,
                                        model = "McFL",
                                        mu = 5e-7,
                                        detectionSize = 1e6,
                                        detectionDrivers = NA,
                                        detectionProb = NA,
                                        keepPhylog = TRUE,
                                        onlyCancer = FALSE,
                                        keepEvery = 1,
                                        mutationPropGrowth = FALSE,
                                        mc.cores = 1
                                        ))
t_mc_50000_nmpg_5mu_k
##    user  system elapsed
## 305.512   5.164 310.711

summary(mc_50000_nmpg_5mu_k)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1     61737      1003273       104460  295.8731  204214
## 2     65072      1000540       133068  296.6243  210231

print(object.size(mc_50000_nmpg_5mu_k), units = "MB")
## 24663.6 Mb
```

Note we use only two replicates, since those two already lead to a 24 GB returned object as we are tracking more than 60,000 clones, more than twice those with $s = 0.1$. The reason for the difference in number of clones and iterations is of course the change from $s = 0.1$ to $s = 0.05$: under the McFarland model to reach population sizes of 10^6 starting from an equilibrium population of 500 we need about 43 mutations (whereas only about 22 are needed if $s = 0.1$[5]).

---

[5]Given the dependence of death rates on population size in McFarland's model (section 3.2.1 and 3.2.1.1), if all mutations have the same fitness effects we can calculate the equilibrium population size (where birth and death rates are equal) for a given number of mutated genes as: $K * (e^{(1+s)^p} - 1)$, where K is the initial equilibrium size, s the fitness effect of each mutation, and p the number of

Next, let us rerun 2.3.2.1:

```
t_mc_50000_nmpg <- system.time(
    mc_50000_nmpg <- oncoSimulPop(5,
                                  u,
                                  model = "McFL",
                                  mu = 1e-7,
                                  detectionSize = 1e6,
                                  detectionDrivers = NA,
                                  detectionProb = NA,
                                  keepPhylog = TRUE,
                                  onlyCancer = FALSE,
                                  keepEvery = NA,
                                  mutationPropGrowth = FALSE,
                                  mc.cores = 1
                                  ))
t_mc_50000_nmpg
##     user   system  elapsed
## 111.236    0.596  111.834

summary(mc_50000_nmpg)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      2646      1000700       217188   734.475  108566
## 2      2581      1001626       209873   806.500  107296
## 3      2903      1001409       125148   841.700  120859
## 4      2310      1000146       473948   906.300   91519
## 5      2704      1001290       448409   838.800  103556

print(object.size(mc_50000_nmpg), units = "MB")
## 2638.3 Mb
```

Using $s = 0.05$ leads to a large increase in final time and number of iterations. However, as we are using the `keepEvery = NA` setting, the increase in number of clones tracked and in size of returned object is relatively small.

2.3.4 The different consequences of `keepEvery = NA` in the Exp and McFL models

We have seen that `keepEvery = NA` often leads to much smaller returned objects when using the McFarland model than when using the Exp model. Why? Because in the McFarland model there is strong competition and there can be complete clonal sweeps so that in extreme cases a single clone might be all that is left after some time. This is not the case in the exponential models.

Of course, the details depend on the difference in fitness effects between different genotypes (or clones). In particular, we have seen several examples where even with `keepEvery=NA` there are a lot of clones in the McFL models. In those examples many

---

mutated genes.

clones had identical fitness (the fitness effects of all genes with positive fitness was the same, and ditto for the genes with negative fitness effects), so no clone ends up displacing all the others.

2.3.5   Are we keeping the complete history (genealogy) of the clones?

Yes we are if we run with `keepPhylog = TRUE`, regardless of the setting for `keepEvery`. As explained in section 18.2, OncoSimulR prunes clones that never had a population size larger than zero at any sampling period (so they are not reflected in the `pops.by.time` matrix in the output). And when we set `keepEvery = NA` we are telling OncoSimulR to discard all sampling periods except the very last one (i.e., the `pops.by.time` matrix contains only the clones with 1 or more cells at the end of the simulation).

`keepPhylog` operates differently: it records the exact time at which a clone appeared and the clone that gave rise to it. This information is kept regardless of whether or not those clones appear in the `pops.by.time` matrix.

Keeping the complete genealogy might be of limited use if the `pops.by.time` matrix only contains the very last period. However, you can use `plotClonePhylog` and ask to be shown only clones that exist in the very last period (while of course showing all of their ancestors, even if those are now extinct —i.e., regardless of their abundance).

For instance, in run 2.3.1.3 we could have looked at the information stored about the genealogy of clones by doing (we look at the first "individual" of the simulation, of the five "individuals" we simulated):

```
head(e_50000[[1]]$other$PhylogDF)
##   parent child   time
## 1         3679 0.8402
## 2         4754 1.1815
## 3        20617 1.4543
## 4        15482 2.3064
## 5         4431 3.7130
## 6        41915 4.0628


tail(e_50000[[1]]$other$PhylogDF)
##                              parent                          child time
## 20672               3679, 20282               3679, 20282, 22359 75.0
## 20673         3679, 17922, 22346       3679, 17922, 22346, 35811 75.0
## 20674               2142, 3679              2142, 3679, 25838 75.0
## 20675         3679, 17922, 19561       3679, 17922, 19561, 43777 75.0
## 20676 3679, 15928, 19190, 20282 3679, 15928, 19190, 20282, 49686 75.0
## 20677         2142, 3679, 16275       2142, 3679, 16275, 24201 75.0
```

where each row corresponds to one event of appearance of a new clone, the column labeled "parent" are the mutated genes in the parent, and the column labeled "child" are the mutated genes in the child.

And we could plot the genealogical relationships of clones that have a population size of at least one in the last period (again, while of course showing all of their ancestors, even if those are now extinct —i.e., regardless of their current numbers) doing:

```
plotClonePhylog(e_50000[[1]]) ## plot not shown
```

What is the cost of keep the clone genealogies? In terms of time it is minor. In terms of space, and as shown in the example above, we can end up storing a data frame with tends of thousands of rows and three columns (two factors, one float). In the example above the size of that data frame is approximately 2 MB for a single simulation. This is much smaller than the `pops.by.time` or `Genotypes` matrices, but it can quickly build up if you routinely launch, say, 1000 simulations via `oncoSimulPop`. That is why the default is `keepPhylog = FALSE` as this information is not needed as often as that in the other two matrices (`pops.by.time` and `Genotypes`).

2.4 Population sizes $\geq 10^{10}$

We have already seen examples where population sizes reach 10^8 to 10^{10}, as in Tables 3, 5, 7. What about even larger population sizes?

The C++ code will unconditionally alert if population sizes exceed $4 * 10^{15}$ as in those cases loosing precision (as we are using doubles) would be unavoidable, and we would also run into problems with the generation of binomial random variates (code that illustrates and discusses this problem is available in file "example-binom-problems.cpp", in directory "/inst/miscell"). However, well before we reach $4 * 10^{15}$ we loose precision from other sources. One of the most noticeable ones is that when we reach population sizes around 10^{11} the C++ code will often alert us by throwing exceptions with the message `Recoverable exception ti set to DBL_MIN. Rerunning.` I throw this exception because t_i, the random variable for time to next mutation, is less than `DBL_MIN`, the minimum representable floating-point number. This happens because, unless we use really tiny mutation rates, the time to a mutation starts getting closer to zero as population sizes grow very large. It might be possible to ameliorate these problems somewhat by using long doubles (instead of doubles) or special purpose libraries that provide more precision. However, this would make it harder to run the same code in different operating systems and would likely decrease execution speed on the rest of the common scenarios for which OncoSimulR has been designed.

The following code shows some examples where we use population sizes of 10^{10} or larger. Since we do not want simulations in the exponential model to end because of extinction, I use a fitness specification where all genes have a positive fitness effect and we start all simulations from a large population (to make it unlikely that the population will become extinct before cells mutate and start increasing in numbers). We set the maximum running time to 10 minutes. We keep the genealogy of the clones and use `keepEvery = 1`.

```
ng <- 50
u <- allFitnessEffects(noIntGenes = c(rep(0.1, ng)))
```

```r
t_mc_k_50_1e11 <- system.time(
    mc_k_50_1e11 <- oncoSimulPop(5,
                                 u,
                                 model = "McFL",
                                 mu = 1e-7,
                                 detectionSize = 1e11,
                                 initSize = 1e5,
                                 detectionDrivers = NA,
                                 detectionProb = NA,
                                 keepPhylog = TRUE,
                                 onlyCancer = FALSE,
                                 mutationPropGrowth = FALSE,
                                 keepEvery = 1,
                                 finalTime = 5000,
                                 mc.cores = 1,
                                 max.wall.time = 600
                                 ))

## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.

t_mc_k_50_1e11
## user   system elapsed
## 613.612    0.040 613.664

summary(mc_k_50_1e11)[, c(1:3, 8, 9)]
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      5491 100328847809  44397848771  1019.950  942764
## 2      3194 100048090441  34834178374   789.675  888819
## 3      5745 100054219162  24412502660   927.950  929231
## 4      4017 101641197799  60932177160   750.725  480938
## 5      5393 100168156804  41659212367   846.250  898245

## print(object.size(mc_k_50_1e11), units = "MB")
## 177.8 Mb
```

We get to 10^{11}. But notice the exception with the warning about t_i. Notice also that this takes a long time and we run a very large number of iterations (getting close to one million in some cases).

Now the exponential model with `detectionSize = 1e11`:

```r
t_exp_k_50_1e11 <- system.time(
    exp_k_50_1e11 <- oncoSimulPop(5,
                                  u,
                                  model = "Exp",
                                  mu = 1e-7,
```

```
                                            detectionSize = 1e11,
                                            initSize = 1e5,
                                            detectionDrivers = NA,
                                            detectionProb = NA,
                                            keepPhylog = TRUE,
                                            onlyCancer = FALSE,
                                            mutationPropGrowth = FALSE,
                                            keepEvery = 1,
                                            finalTime = 5000,
                                            mc.cores = 1,
                                            max.wall.time = 600,
                                            errorHitWallTime = FALSE,
                                            errorHitMaxTries = FALSE
                                            ))

## Recoverable exception ti set to DBL_MIN. Rerunning.
## Hitted wall time. Exiting.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Hitted wall time. Exiting.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Recoverable exception ti set to DBL_MIN. Rerunning.
## Hitted wall time. Exiting.
## Hitted wall time. Exiting.

t_exp_k_50_1e11
##     user   system  elapsed
## 2959.068    0.128 2959.556
try(summary(exp_k_50_1e11)[, c(1:3, 8, 9)])
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1      6078  65172752616  16529682757  235.7590 1883438
## 2      5370 106476643712  24662446729  232.0000 2516675
## 3      2711  21911284363  17945303353  224.8608  543698
## 4      2838  13241462284   2944300245  216.8091  372298
## 5      7289  76166784312  10941729810  240.0217 1999489

print(object.size(exp_k_50_1e11), units = "MB")
## 53.5 Mb
```

Note that we almost reached `max.wall.time` (600 * 5 = 3000). What if we wanted to go up to 10^{12}? We would not be able to do it in 10 minutes. We could set `max.wall.time` to a value larger than 600 to allow us to reach larger sizes but

then we would be waiting for a possibly unacceptable time for simulations to finish. Moreover, this would eventually fail as simulations would keep hitting the t_i exception without ever being able to complete. Finally, even if we were very patient, hitting that t_i exception should make us worry about possible biases in the samples.

2.5 A summary of some determinants of running time and space consumption

To summarize this section, we have seen:

- Both McFL and Exp can be run in short times over a range of sizes for the `detectionProb` and `detectionSize` mechanisms using a complex fitness specification with moderate numbers of genes. These are the typical or common use cases of OncoSimulR.

- The `keepEvery` argument can have a large effect on time in the McFL models and specially on object sizes. If only the end result of the simulation is to be used, you should set `keepEvery = NA`.

- The distribution of fitness effects and the fitness landscape can have large effects on running times. Sometimes these are intuitive and simple to reason about, sometimes they are not as they interact with other factors (e.g., stopping mechanism, numbers of clones, etc). In general, there can be complex interactions between different settings, from mutation rate to fitness effects to initial size. As usual, test before launching a massive simulation.

- Simulations start to slow down and lead to a very large object size when we keep track of around 6000 to 10000 clones. Anything that leads to these patterns will slow down the simulations.

- OncoSimulR needs to keep track of genotypes (or clones), not just numbers of drivers and passengers, because it allows you to use complex fitness and mutation specifications that depend on specific genotypes. The `keepEvery = NA` is an approach to store only the minimal information needed, but it is unavoidable that during the simulations we might be forced to deal with many thousands of different clones.

3 Specifying fitness effects

OncoSimulR uses a standard continuous time model, where individual cells divide, die, and mutate with rates that can depend on genotype and population size; over time the abundance of the different genotypes changes by the action of selection (due to differences in net growth rates among genotypes), drift, and mutation. As a result of a mutation in a pre-existing clone new clones arise, and the birth rate of a newly arisen clone is determined at the time of its emergence as a function of its genotype. Simulations can use an use exponential growth model or a model with carrying capacity that follows McFarland et al. (2013). For the exponential growth model, the death rate is fixed at one whereas in the model with carrying capacity death rate increases with population size. In both cases, therefore, fitness differences among genotypes in a given population at a given time are due to differences in the mapping between genotype and birth rate. There is second exponential model (called "Bozic") where birth rate is fixed at one, and genotype determines death rate instead of birth rate (see details in 3.2). So when we discuss specifying fitness effects or the effects of genes on fitness, we are actually referring to specifying effects on birth (or death) rates, which then translate into differences in fitness (since the other rate, death or birth, is either fixed, as in the Exp and Bozic models, or depends on the population size). This is also shown in Table 1, in the rows for "Fitness components", under "Evolutionary Features".

In the case of frequency-dependent fitness simulations (see section 10), the fitness effects must be reevaluated frequently so that birth rate, death rate, or both, depending the model used, are updated. To do this it is necessary to use a short step to reevaluate fitness; this is done using a small value for `sampleEvery` parameter in `oncoSimulindv` (see 18.8 for more details), as is the case when using McFarland model.

Incidentally, notice that with OncoSimulR we do not directly specify fitness itself (even if, for the sake of simplicity, we often refer to fitness in the documentation) as fitness is, arguably, a derived quantity (Doebeli et al., 2017). Rather, we specify how birth and/or death rates, which are the actual mechanistic drivers of evolutionary dynamics, are related to genotypes (or to the frequencies of the different genotypes).

3.1 Introduction to the specification of fitness effects

With OncoSimulR you can specify different types of effects on fitness:

- A special type of epistatic effect that is particularly amenable to be represented as a graph (a DAG). In this graph having, say, "B" be a child of "A" means that a mutation in B can only accumulate if a mutation in A is already present. This is what OT (Desper et al., 1999; Szabo & Boucher, 2008), CBN (Beerenwinkel, Eriksson, et al., 2007; Gerstung et al., 2009; Gerstung, Eriksson, et al., 2011), progression networks (Farahani & Lagergren, 2013), and other similar models (Korsunsky et al., 2014) generally mean. Details are provided in section 3.4. Note that this is not an order effect (discussed below): the fitness of a genotype from this DAGs is a function of whether or not the restrictions in the graph

are satisfied, not the historical sequence of how they were satisfied.

- Effects where the order in which mutations are acquired matters, as illustrated in section 3.6. There is, in fact, empirical evidence of these effects (Ortmann et al., 2015). For instance, the fitness of genotype "A, B" would differ depending on whether A or B was acquired first (or, as in the actual example in (Ortmann et al., 2015), the fitness of the mutant with JAK2 and TET2 mutated will depend on which of the genes was mutated first).

- General epistatic effects (e.g., section 3.7), including synthetic viability (e.g., section 3.9) and synthetic lethality/mortality (e.g., section 3.10).

- Genes that have independent effects on fitness (section 3.3).

- Modules (see section 3.5) allow you to specify any of the above effects (except those for genes without interactions, as it would not make sense there) in terms of modules (sets of genes), not individual genes. We will introduce them right after 3.4, and we will continue using them thereafter.

A guiding design principle of OncoSimulR is to try to make the specification of those effects as simple as possible but also as flexible as possible. Thus, there are two main ways of specifying fitness effects:

- Combining different types of effects in a single specification. For instance, you can combine epistasis with order effects with no interaction genes with modules. What you would do here is specify the effects that different mutations (or their combinations) have on fitness (the fitness effects) and then have OncoSimulR take care of combining them as if each of these were lego pieces. We will refer to this as the **lego system of fitness effects**. (As explained above, I find this an intuitive and very graphical analogy, which I have copied from Hothorn et al. (2006) and Hothorn et al. (2008)).

- Explicitly passing to OncoSimulR a mapping of genotypes to fitness. Here you specify the fitness of each genotype. We will refer to this as the **explicit mapping of genotypes to fitness**. This includes frequency-dependent fitness (section 10).

Both approaches have advantages and disadvantages. Here I emphasize some relevant differences.

- With the lego system you can specify huge genomes with an enormous variety of interactions, since the possible genotypes are not constructed in advance. You would not be able to do this with the explicit mapping of genotypes to fitness if you wanted to, say, construct that mapping for a modest genotype of 500 genes (you'd have more genotypes than particles in the observable Universe).

- For many models/data you often intuitively start with the fitness of the genotypes, not the fitness consequences of the different mutations. In these cases, you'd need to do the math to specify the terms you want if you used the lego system so you'll probably use the specification with the direct mapping genotype $\rightarrow$ fitness.

- Likewise, sometimes you already have a moderate size genotype → fitness mapping and you certainly do not want to do the math by hand: here the lego system would be painful to use.

- But sometimes we do think in terms of "the effects on fitness of such and such mutations are" and that immediately calls for the lego system, where you focus on the effects, and let OncoSimulR take care of doing the math of combining.

- If you want to use order effects, you must use the lego system (at least for now).

- If you want to specify modules, you must use the lego system (the explicit mapping of genotypes is, by its very nature, ill-suited for this).

- The lego system might help you see what your model really means: in many cases, you can obtain fairly succinct specifications of complex fitness models with just a few terms. Similarly, depending on what your emphasis is, you can often specify the same fitness landscape in several different ways.

Regardless of the route, you need to get that information into OncoSimulR's functions. The main function we will use is `allFitnessEffects`: this is the function in charge of reading the fitness specifications. We also need to discuss how, what, and where you have to pass to `allFitnessEffects`.

3.1.1 Explicit mapping of genotypes to fitness

Conceptually, the simplest way to specify fitness is to specify the mapping of all genotypes to fitness explicitly. An example will make this clear. Let's suppose you have a simple two-gene scenario, so a total of four genotypes, and you have a data frame with genotypes and fitness, where genoytpes are specified as character vectors, with mutated genes separated by commas:

```
m4 <- data.frame(G = c("WT", "A", "B", "A, B"), F = c(1, 2, 3, 4))
```

Now, let's give that to the `allFitnessEffects` function:

```
fem4 <- allFitnessEffects(genotFitness = m4)
## Column names of object not Genotype and Birth Renaming them assuming that is w
```
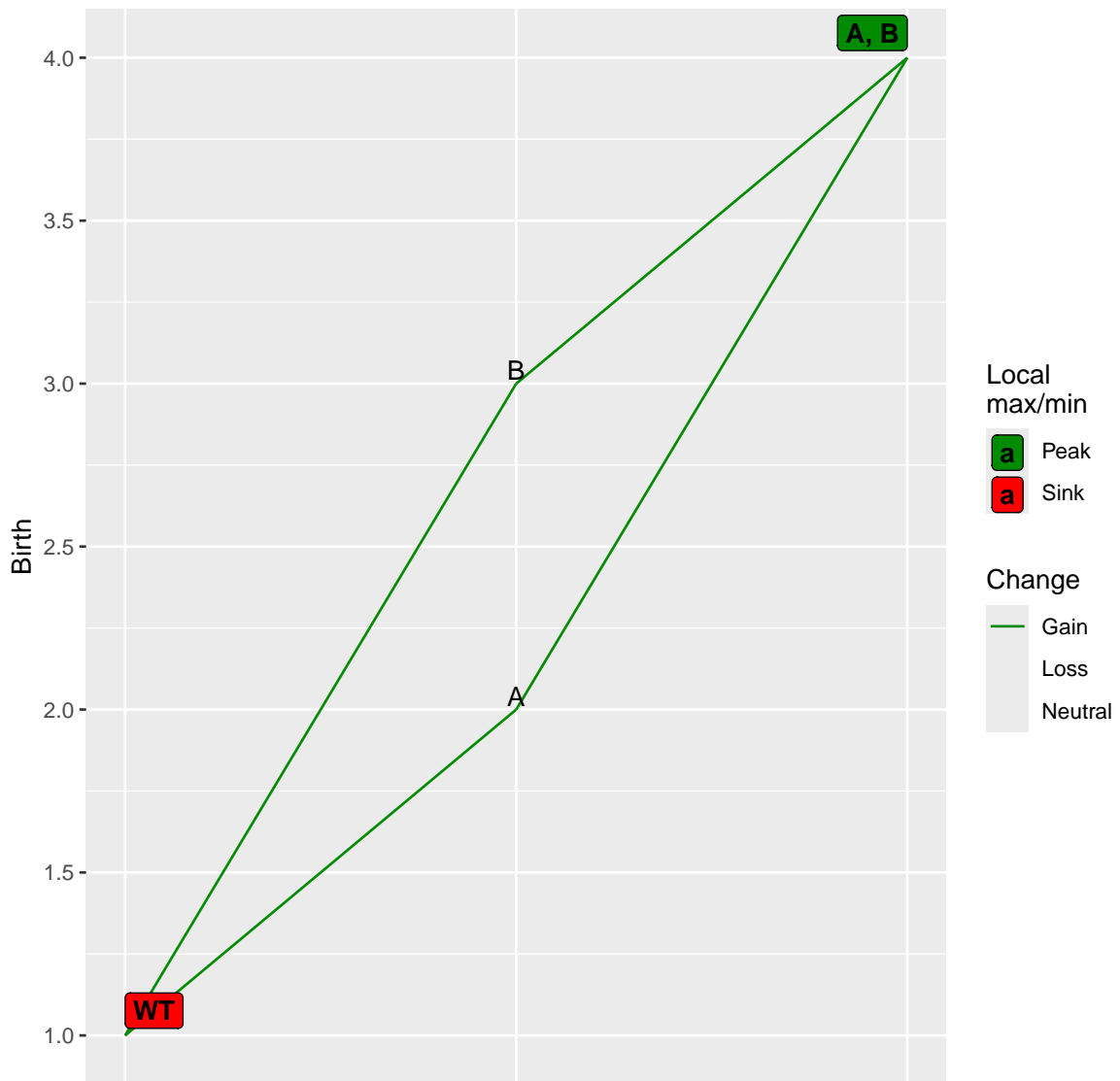
(The message is just telling you what the program guessed you wanted.)

That's it. You can try to plot that fitnessEffects object

```
try(plot(fem4))
## Error in plot.fitnessEffects(fem4) :
##   This fitnessEffects object can not be ploted this way. It is probably one wit
```

In this case, you probably want to plot the fitness landscape.

```
plotFitnessLandscape(evalAllGenotypes(fem4))
```
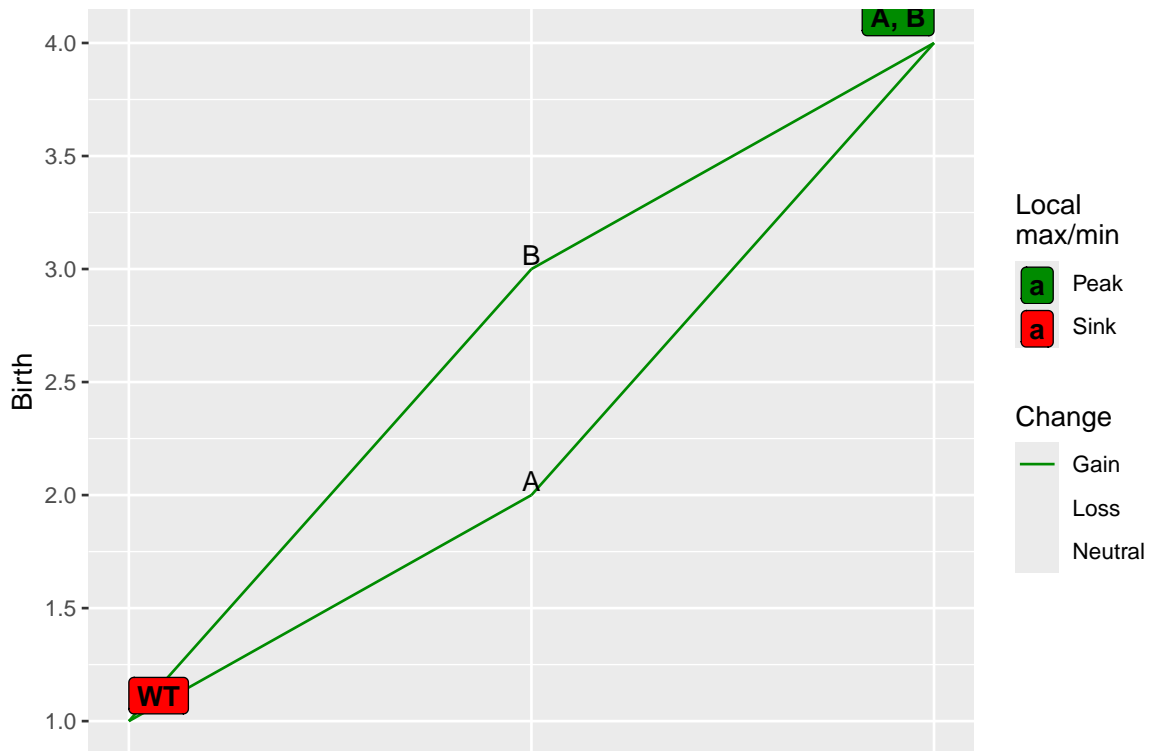
You can also check what OncoSimulR thinks the fitnesses are, with the `evalAllGenotypes` function that we will use repeatedly below (of course, here we should see the same fitnesses we entered):

```
evalAllGenotypes(fem4, addwt = TRUE)
##   Genotype Birth
## 1       WT     1
## 2        A     2
## 3        B     3
## 4     A, B     4
```

And you can plot the fitness landscape:

```
plotFitnessLandscape(evalAllGenotypes(fem4))
```

To specify the mapping you can also use a matrix (or data frame) with $g+1$ columns; each of the first g columns contains a 1 or a 0 indicating that the gene of that column is mutated or not. Column $g+1$ contains the fitness values. And you do not even need to specify all the genotypes: the missing genotypes are assigned a fitness 0 —except for the WT genotype which, if missing, is assigned a fitness of 1:

```r
m6 <- cbind(c(1, 1), c(1, 0), c(2, 3))
fem6 <- allFitnessEffects(genotFitness = m6)
## No column names: assigning gene names from LETTERS
## Warning in to_genotFitness_std(genotFitness,
## frequencyDependentBirth = FALSE, : No wildtype in the fitness
## landscape!!! Adding it with birth 1.
```

```r
evalAllGenotypes(fem6, addwt = TRUE)
##   Genotype Birth
## 1       WT     1
## 2        A     3
## 3        B     0
## 4     A, B     2
```

```r
## plot(fem6)
```

```r
plotFitnessLandscape(evalAllGenotypes(fem6))
```
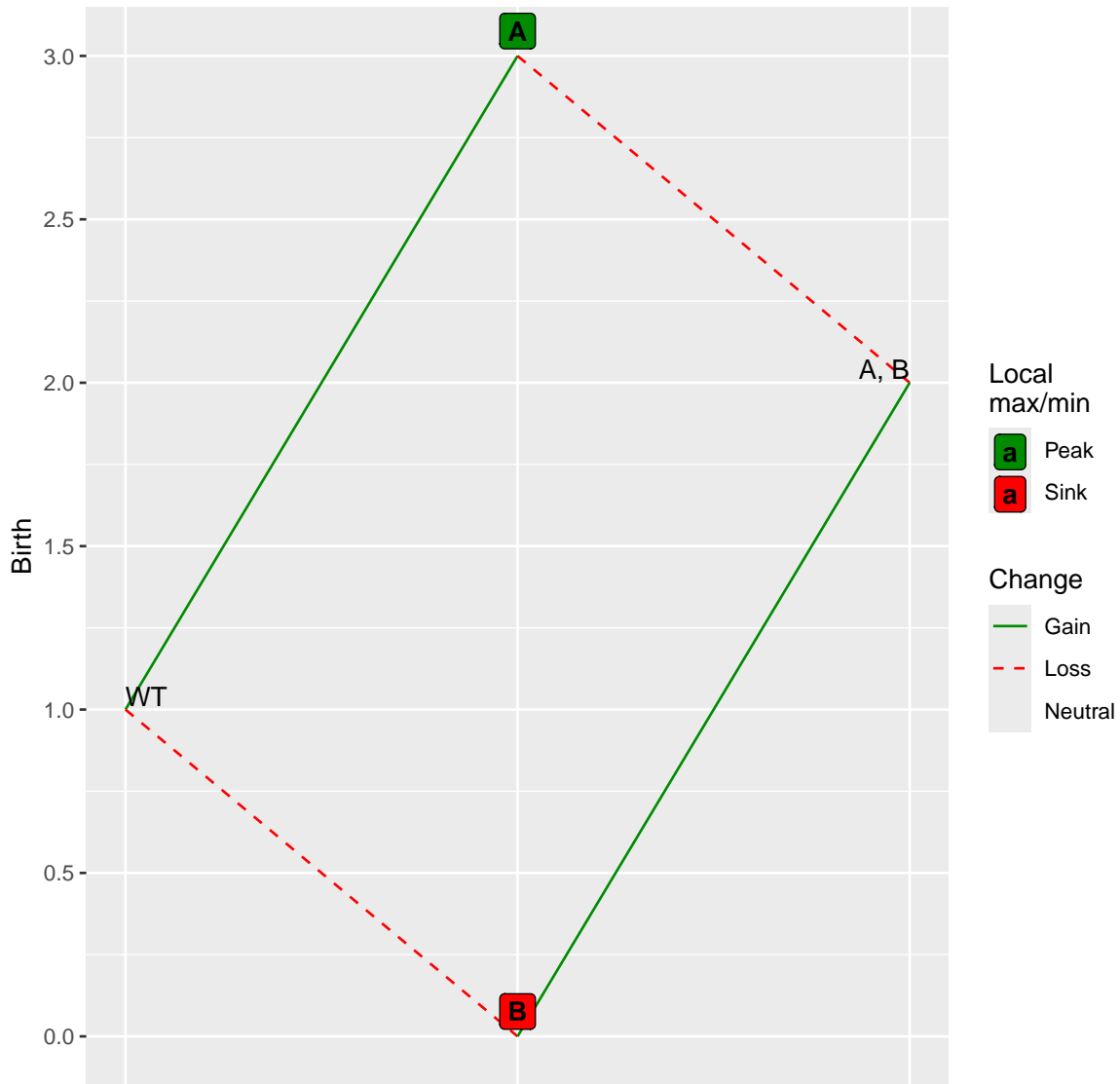
This way of giving a fitness specification to OncoSimulR might be ideal if you directly generate random mappings of genotypes to fitness (or random fitness landscapes), as we will do in section 9. Specially when the fitness landscape contains many non-viable genotypes (which are considered those with fitness —birth rate— $< 1e - 9$) this can result in considerable savings as we only need to check the fitness of the viable genotypes in a table (a C++ map). Note, however, that using the Bozic model with the fitness landscape specification is not tested. In addition, for speed, missing genotypes from the fitness landscape specification are taken to be non-viable genotypes (**beware!! this is a breaking change relative to versions $< 2.9.1$**)[6].

In the case of frequency-dependent fitness situations, the only way to specify fitness effects is using `genoFitnes` as we have shown before, but now you need to set `frequencyDependentFitness = TRUE` in `allFitnessEffects`. The fundamental

---

[6]Note for curious readers: it used to be the case that we converted the table of fitness of genotypes to a fitness specification with all possible epistatic interactions; you can take a look at the test file `test.genot_fitness_to_epistasis.R` that uses the `fem6` object. We no longer do that but instead pass directly the fitness landscape.

difference is the Fitness column in `genoFitnes`. Now this column must be a character vector and each element (character also) is a function whose variables are the relative frequencies of the clones in the population. You must specify the variables like f_, for frequency of wild type, f_1 or f_A for frequency of mutant A or position 1, f_1_2 or f_A_B for double mutant, and so on. Mathematical operations and symbols allowed are described in the documentation of C++ library ExprTk (http://www.partow.net/programming/exprtk/). ExprTk is the library used to parse and evaluate the fitness equations. The numeric vector `spPopSizes` is only necesary to evaluate genotypes through `evalGenotype` or `evalAllGenotypes` functions because population sizes are needed to calculate the clone's frequencies.

```r
r <- data.frame(Genotype = c("WT", "A", "B", "A, B"),
                Fitness = c("10 * f_",
                            "10 * f_1",
                            "50 * f_2",
                            "200 * (f_1 + f_2) + 50 * f_1_2"))


afe <- allFitnessEffects(genotFitness = r,
                         frequencyDependentFitness = TRUE,
                         frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = r,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```r
plotFitnessLandscape(evalAllGenotypes(afe,
                                      spPopSizes = c(WT = 2500, A = 2000,
                                                     B = 5500, "A, B" = 700)))
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
```
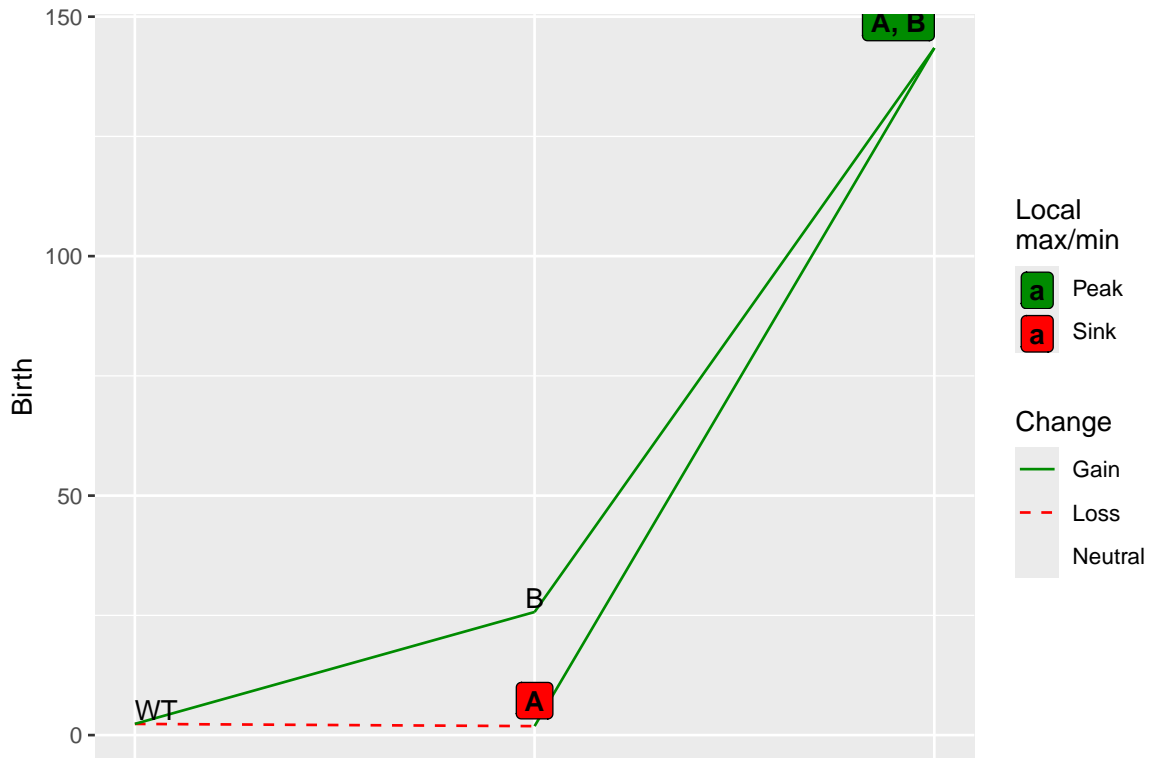
The above example is simple enough in terms of genes and genotypes that using `f_1` is OK. But it will be better, as examples get more complex, to use:

```r
r <- data.frame(Genotype = c("WT", "A", "B", "A, B"),
                Fitness = c("10 * f_",
                            "10 * f_A",
                            "50 * f_B",
                            "200 * (f_A + f_B) + 50 * f_A_B"))
```

which makes explicit what depends on what (i.e., you do not need to keep in mind the mapping of letters to numbers). In other words, we write `f_genotype expressed as combination of gene names`, with the gene names we are actually using. And those `f_something_other`, will match the genotypes given in `Genotype` (there will a `something, other` genotype).

3.1.2 How to specify fitness effects with the lego system

An alternative general approach followed in many genetic simulators is to specify how particular combinations of alleles modify the wildtype genotype or the genotype that contains the individual effects of the interacting genes (e.g., see equation 1 in the supplementary material for FFPopSim (Zanini & Neher, 2012)). For example, if we specify that a mutation in "A" contributes 0.04, a mutation in "B" contributes 0.03, and the double mutation "A:B" contributes 0.1, that means that the fitness of the "A, B" genotype (the genotype with A and B mutated) is that of the wildtype (1, by default), plus (actually, times —see section 3.2— but plus on the log scale) the effects of having A mutated, plus (times) the effects of having B mutated, plus (times) the effects of "A:B" both being mutated.

We will see below that with the "lego system" it is possible to do something very similar to the explicit mapping of section 3.1.1. But this will sometimes require a more cumbersome notation (and sometimes also will require your doing some math). We will see examples in sections 3.7.1, 3.7.2 and 3.7.3 or the example in 5.4.2. But then, if we can be explicit about (at least some of) the mappings $genotype \rightarrow fitness$, how are these procedures different? When you use the "lego system" you can combine both a partial explicit mapping of genotypes to fitness with arbitrary fitness effects of other genes/modules. In other words, with the "lego system" OncoSimulR makes it simple to be explicit about the mapping of specific genotypes, while also using the "how this specific effects modifies previous effects" logic, leading to a flexible specification. This also means that in many cases the same fitness effects can be specified in several different ways.

Most of the rest of this section is devoted to explaining how to combine those pieces. Before that, however, we need to discuss the fitness model we use.

3.2 Numeric values of fitness effects

We evaluate fitness using the usual (Beerenwinkel, Eriksson, et al., 2007; Datta et al., 2013; Gillespie, 1993; Zanini & Neher, 2012) multiplicative model: fitness is $\prod(1 + s_i)$ where s_i is the fitness effect of gene (or gene interaction) i. In all models except Bozic, this fitness refers to the growth rate (the death rate being fixed to 1^7). The original model of McFarland et al. (2013) has a slightly different parameterization, but you can go easily from one to the other (see section 3.2.1).

For the Bozic model (Bozic et al., 2010), however, the birth rate is set to 1, and the death rate then becomes $\prod(1 - s_i)$.

3.2.1 McFarland parameterization

In the original model of McFarland et al. (2013), the effects of drivers contribute to the numerator of the birth rate, and those of the (deleterious) passengers to the denominator as: $\frac{(1+s)^d}{(1+s_p)^p}$, where d and p are, respectively, the total number of drivers and passengers in a genotype, and here the fitness effects of all drivers is the same (s) and that of all passengers the same too (s_p). Note that, as written above, and as explicitly said in McFarland et al. (2013) (see p. 2911) and McFarland (2014) (see p. 9), "(...) s_p is the fitness disadvantage conferred by a passenger". In other words, the larger the s_p the more deleterious the passenger.

This is obvious, but I make it explicit because in our parameterization a positive s means fitness advantage, whereas fitness disadvantages are associated with negative s. Of course, if you rewrite the above expression as $\frac{(1+s)^d}{(1-s_p)^p}$ then we are back to the "positive means fitness advantage and negative means fitness disadvantage".

As McFarland (2014) explains (see p. 9, bottom), we can rewrite the above expression so that there are no terms in the denominator. McFarland writes it as (I copy verbatim

---

[7] You can change this if you really want to.

from the fourth and fifth lines from the bottom on his p. 9) $(1 + s_d)^{n_d}(1 - s'_p)^{n_p}$ where $s'_p = s_p/(1 + s_p)$.

However, if we want to express everything as products (no ratios) and use the "positive s means advantage and negative s means disadvantage" rule, we want to write the above expression as $(1 + s_d)^{n_d}(1 + s_{pp})^{n_p}$ where $s_{pp} = -s_p/(1 + s_p)$. And this is actually what we do in v.2. There is an example, for instance, in section 6.5.2 where you will see:

```
sp <- 1e-3
spp <- -sp/(1 + sp)
```

so we are going from the "(. . .) s_p is the fitness disadvantage conferred by a passenger" in McFarland et al. (2013) (p. 2911) and McFarland (2014) (p. 9) to the expression where we have a product $\prod(1+s_i)$, with the "positive s means advantage and negative s means disadvantage" rule. This reparameterization applies to v.2. In v.1 we used the same parameterization as in the original one in McFarland et al. (2013), but with the "positive s means advantage and negative s means disadvantage" rule (so we are using expression $\frac{(1+s)^d}{(1-s_p)^p}$).

3.2.1.1 Death rate under the McFarland model

For death rate, we use the expression that McFarland et al. (2013) (see their p. 2911) use "(. . .) for large cancers (grown to 10^6 cells)": $D(N) = \log(1 + N/K)$ where K is the initial equilibrium population size. As the authors explain, for large N/K the above expression "(. . .) recapitulates Gompertzian dynamics observed experimentally for large tumors".

By default, OncoSimulR uses a value of $K = initSize/(e^1 - 1)$ so that the starting population is at equilibrium.

A consequence of the above expression for death rate is that if the population size decreases the death rate decreases. This is not relevant in most cases (as mutations, or some mutations, will inexorably lead to population size increases). And this prevents the McFL model from resulting in extinction even with very small population sizes as long as birth rate $\geq$ death rate. (For small population sizes, it is likely that the population will become extinct if birth rate = death rate; you can try this with the exponential model).

But this is not what we want in some other models, such as frequency-dependent ones, where modeling population collapse (which will happen if birth rate < death rate) can be important (as in the example in 10.3). Here, it makes sense to set $D(N) = \max(1, \log(1 + N/K))$ so that the death rate never decreases below 1. (Using 1 is reasonable if we consider the equilibrium birth rate in the absence of any mutants to be 1). You can specify this behaviour using model `McFLD` (a shorthand for `McFarlandLogD`).

3.2.2 No viability of clones and types of models

For all models where fitness affects directly the birth rate (all except Bozic), if you specify that some event (say, mutating gene A) has $s_A \leq -1$, if that event happens

then birth rate becomes zero. This is taken to indicate that the clone is not even viable and thus disappears immediately without any chance for mutation[8].

Models based on Bozic, however, have a birth rate of 1 and mutations affect the death rate. In this case, a death rate larger than birth rate, *per se*, does not signal immediate extinction and, moreover, even for death rates that are a few times larger than birth rates, the clone could mutate before becoming extinct[9].

In general, if you want to identify some mutations or some combinations of mutations as leading to immediate extinction (i.e., no viability), of the affected clone, set it to $-\infty$ as this would work even if how birth rates of 0 are handled changes. Most examples below evaluate fitness by its effects on the birth rate. You can see one where we do it both ways in Section 3.11.1.

3.3 Genes without interactions

This is a simple scenario. Each gene i has a fitness effect s_i if mutated. The s_i can come from any distribution you want. As an example let's use three genes. We know there are no order effects, but we will also see what happens if we examine genotypes as ordered.

```
ai1 <- evalAllGenotypes(allFitnessEffects(
    noIntGenes = c(0.05, -.2, .1), frequencyDependentFitness = FALSE), order = FALS
## Warning in allFitnessEffects(noIntGenes = c(0.05, -0.2, 0.1),
## frequencyDependentFitness = FALSE): v2 functionality detected.
## Adapting to v3 functionality.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

We can easily verify the first results:

```
ai1
##   Genotype Fitness
## 1        1   1.050
```

---

[8]This is a shortcut that we take because we think that it is what you mean. Note, however, that technically a clone with birth rate of 0 might have a non-zero probability of mutating before becoming extinct because in the continuous time model we use mutation is not linked to reproduction. In the present code, we are not allowing for any mutation when birth rate is 0. There are other options, but none which I find really better. An alternative implementation makes a clone immediately extinct if and only if any of the $s_i = -\infty$. However, we still need to handle the case with $s_i < -1$ as a special case. We either make it identical to the case with any $s_i = -\infty$ or for any $s_i > -\infty$ we set $(1 + s_i) = \max(0, 1 + s_i)$ (i.e., if $s_i < -1$ then $(1 + s_i) = 0$), to avoid obtaining negative birth rates (that make no sense) and the problem of multiplying an even number of negative numbers. I think only the second would make sense as an alternative.

[9]We said "a few times". For a clone of population size 1 —which is the size at which all clones start from mutation—, if death rate is, say, 90 but birth rate is 1, the probability of mutating before becoming extinct is very, very close to zero for all reasonable values of mutation rate}. How do we signal immediate extinction or no viability in this case? You can set the value of $s = -\infty$.

```
## 2        2    0.800
## 3        3    1.100
## 4     1, 2    0.840
## 5     1, 3    1.155
## 6     2, 3    0.880
## 7  1, 2, 3    0.924
```

```
all(ai1[, "Fitness"]  == c( (1 + .05), (1 - .2), (1 + .1),
        (1 + .05) * (1 - .2),
        (1 + .05) * (1 + .1),
        (1 - .2) * (1 + .1),
        (1 + .05) * (1 - .2) * (1 + .1)))
## [1] TRUE
```

And we can see that considering the order of mutations (see section 3.6) makes no difference:

```
(ai2 <- evalAllGenotypes(allFitnessEffects(
    noIntGenes = c(0.05, -.2, .1)), order = TRUE,
    addwt = TRUE))
##      Genotype Birth
## 1          WT 1.000
## 2           1 1.050
## 3           2 0.800
## 4           3 1.100
## 5       1 > 2 0.840
## 6       1 > 3 1.155
## 7       2 > 1 0.840
## 8       2 > 3 0.880
## 9       3 > 1 1.155
## 10      3 > 2 0.880
## 11 1 > 2 > 3 0.924
## 12 1 > 3 > 2 0.924
## 13 2 > 1 > 3 0.924
## 14 2 > 3 > 1 0.924
## 15 3 > 1 > 2 0.924
## 16 3 > 2 > 1 0.924
```

(The meaning of the notation in the output table is as follows: "WT" denotes the wild-type, or non-mutated clone. The notation $x > y$ means that a mutation in "x" happened before a mutation in "y". A genotype $x > y \_ z$ means that a mutation in "x" happened before a mutation in "y"; there is also a mutation in "z", but that is a gene for which order does not matter).

And what if I want genes without interactions but I want modules (see section 3.5)? Go to section 3.8.

3.4 Using DAGs: Restrictions in the order of mutations as extended posets

3.4.1 AND, OR, XOR relationships

The literature on Oncogenetic trees, CBNs, etc, has used graphs as a way of showing the restrictions in the order in which mutations can accumulate. The meaning of "convergent arrows" in these graphs, however, differs. In Figure 1 of Korsunsky et al. (2014) we are shown a simple diagram that illustrates the three basic different meanings of convergent arrows using two parental nodes. We will illustrate it here with three. Suppose we focus on node "g" in the following figure (we will create it shortly)

```
data(examplesFitnessEffects)
plot(examplesFitnessEffects[["cbn1"]])
## This graph was created by an old(er) igraph version.
##   Call upgrade_graph() on it to use with the current igraph version
##   For now we convert it on the fly...
```
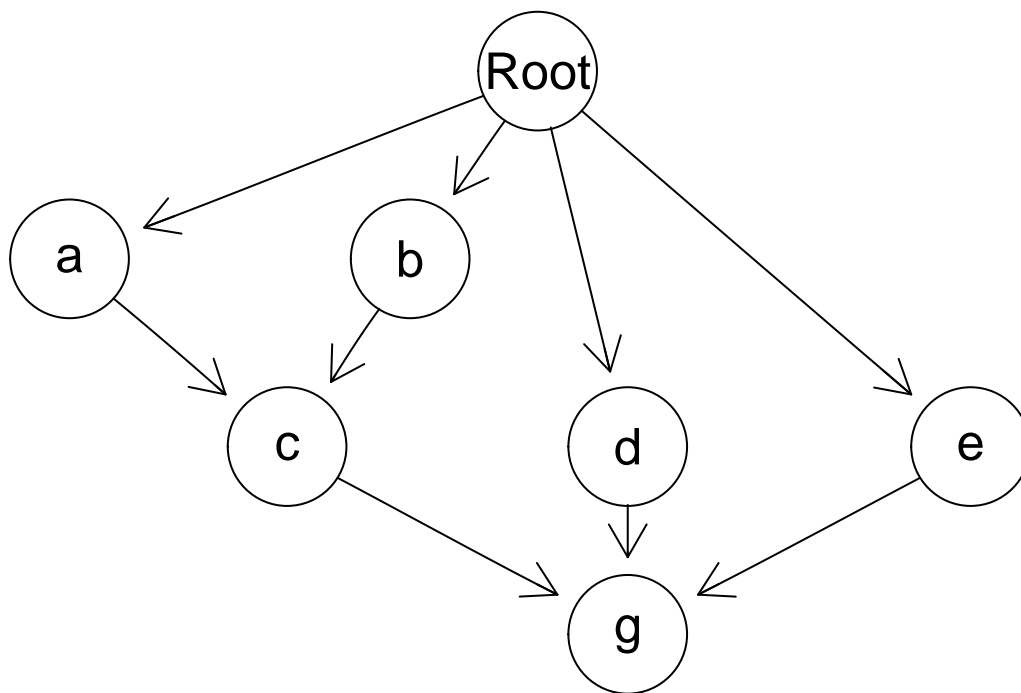

- In relationships of the type used in **Conjunctive Bayesian Networks (CBN)** (e.g., Gerstung et al., 2009), we are modeling an **AND** relationship, also called **CMPN** by Korsunsky et al. (2014) or **monotone** relationship by Farahani & Lagergren (2013). If the relationship in the graph is fully respected, then "g" will only appear if all of "c", "d", and "e" are already mutated.

- **Semimonotone** relationships *sensu* Farahani & Lagergren (2013) or **DMPN** *sensu* Korsunsky et al. (2014) are **OR** relationships: "g" will appear if one or more of "c", "d", or "e" are already mutated.

- **XMPN** relationships (Korsunsky et al., 2014) are **XOR** relationships: "g" will

be present only if exactly one of "c", "d", or "e" is present.

Note that Oncogenetic trees (Desper et al., 1999; Szabo & Boucher, 2008) need not deal with the above distinctions, since the DAGs are trees: no node has more than one incoming connection or more than one parent[10].

To have a flexible way of specifying all of these restrictions, we will want to be able to say what kind of dependency each child node has on its parents.

3.4.2 Fitness effects

Those DAGs specify dependencies and, as explained in Diaz-Uriarte (2015), it is simple to map them to a simple evolutionary model: any set of mutations that does not conform to the restrictions encoded in the graph will have a fitness of 0. However, we might not want to require absolute compliance with the DAG. This means we might want to allow deviations from the DAG with a corresponding penalization that is, however, not identical to setting fitness to 0 (again, see Diaz-Uriarte, 2015). This we can do by being explicit about the fitness effects of the deviations from the restrictions encoded in the DAG. We will use below a column of `s` for the fitness effect when the restrictions are satisfied and a column of `sh` when they are not. (See also 3.2 for the details about the meaning of the fitness effects).

That way of specifying fitness effects makes it also trivial to use the model in Hjelm et al. (2006) where all mutations might be allowed to occur, but the presence of some mutations increases the probability of occurrence of other mutations. For example, the values of `sh` could be all small positive ones (or for mildly deleterious effects, small negative numbers), while the values of `s` are much larger positive numbers.

3.4.3 Extended posets

In version 1 of this package we used posets in the sense of Beerenwinkel, Eriksson, et al. (2007) and Gerstung et al. (2009), as explained in the help for `poset`. The functionality for simulating directly from such two column matrices has been removed. Instead, we use what we call extended posets.

With the extended posets, we continue using two columns, that specify parents and children, but we add columns for the specific values of fitness effects (both s and sh —i.e., fitness effects for what happens when restrictions are and are not satisfied) and for the type of dependency as explained in section 3.4.1.

We can now illustrate the specification of different fitness effects using DAGs.

3.4.4 DAGs: A first conjunction (AND) example

```
cs <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
```

---

[10]OTs and CBNs have some other technical differences about the underlying model they assume, such as the exponential waiting time in CBNs. We will not discuss them here.
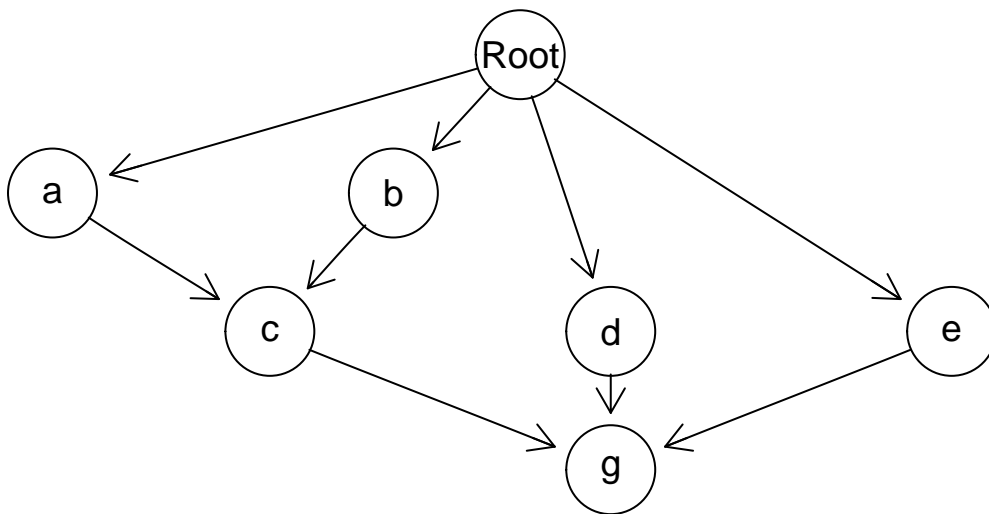
```
                    s = 0.1,
                    sh = -0.9,
                    typeDep = "MN")
```

```
cbn1 <- allFitnessEffects(cs)
```

(We skip one letter, just to show that names need not be consecutive or have any particular order.)
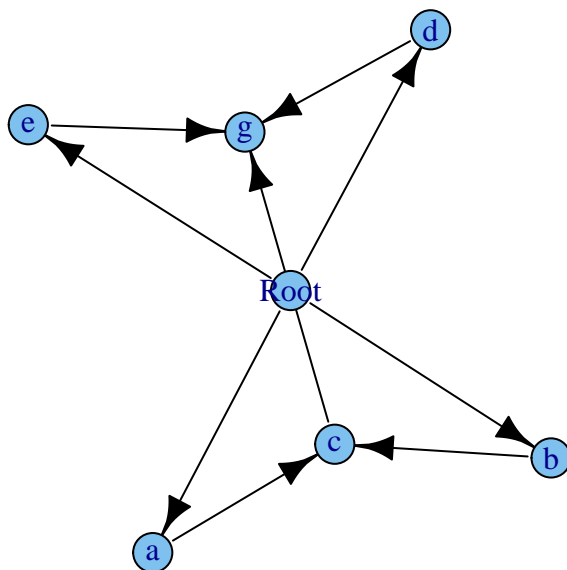
We can get a graphical representation using the default "graphNEL"
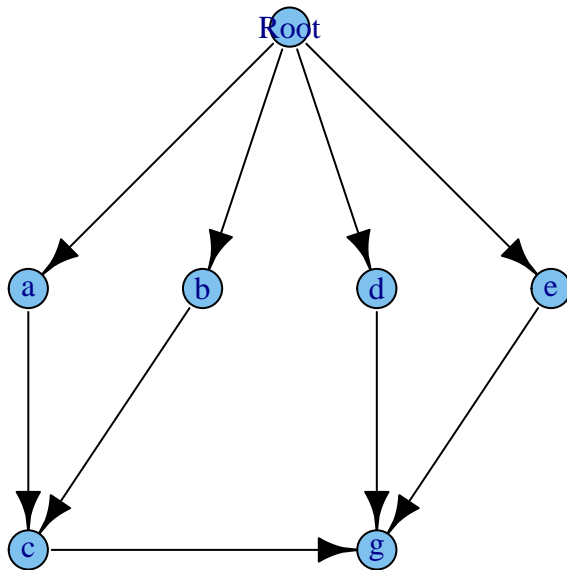
```
plot(cbn1)
```


or one using "igraph":

```
plot(cbn1, "igraph")
```


Since we have a parent and children, the reingold.tilford layout is probably the best here, so you might want to use that:

```
library(igraph) ## to make the reingold.tilford layout available
plot(cbn1, "igraph", layout = layout.reingold.tilford)
```


And what is the fitness of all genotypes?

```
gfs <- evalAllGenotypes(cbn1, order = FALSE, addwt = TRUE)

gfs[1:15, ]
##    Genotype Birth
## 1       WT  1.00
## 2        a  1.10
## 3        b  1.10
## 4        c  0.10
## 5        d  1.10
## 6        e  1.10
## 7        g  0.10
## 8     a, b  1.21
## 9     a, c  0.11
## 10    a, d  1.21
## 11    a, e  1.21
## 12    a, g  0.11
## 13    b, c  0.11
## 14    b, d  1.21
## 15    b, e  1.21
```

You can verify that for each genotype, if a mutation is present without all of its
dependencies present, you get a $(1-0.9)$ multiplier, and you get a $(1+0.1)$ multiplier
for all the rest with its direct parents satisfied. For example, genotypes "a", or "b",
or "d", or "e" have fitness $(1+0.1)$, genotype "a, b, c" has fitness $(1+0.1)^3$, but
genotype "a, c" has fitness $(1+0.1)(1-0.9) = 0.11$.

3.4.5 DAGs: A second conjunction example

Let's try a first attempt at a somewhat more complex example, where the fitness consequences of different genes differ.

```
c1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.1, -.2), c(-.05, -.06, -.07)),
                 typeDep = "MN")


try(fc1 <- allFitnessEffects(c1))
## Error in FUN(X[[i]], ...) : Not all sh identical within a child
```

If you try this, you'll get an error. There is an error because the "sh" varies within a child, and we do not allow that for a poset-type specification, as it is ambiguous. If you need arbitrary fitness values for arbitrary combinations of genotypes, you can specify them using epistatic effects as in section 3.7 and order effects as in section 3.6.

Why do we need to specify as many "s" and "sh" as there are rows (or a single one, that gets expanded to those many) when the "s" and "sh" are properties of the child node, not of the edges? Because, for ease, we use a data.frame.

We fix the error in our specification. Notice that the "sh" is not set to -1 in these examples. If you want strict compliance with the poset restrictions, you should set $sh = -1$ or, better yet, $sh = -\infty$ (see section 3.2.2), but having an $sh > -1$ will lead to fitnesses that are > 0 and, thus, is a way of modeling small deviations from the poset (see discussion in Diaz-Uriarte, 2015).

Note that for those nodes that depend only on "Root" the type of dependency is irrelevant.

```
c1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.9, -.9), rep(-.95, 3)),
                 typeDep = "MN")


cbn2 <- allFitnessEffects(c1)
```

We could get graphical representations but the figures would be the same as in the example in section 3.4.4, since the structure has not changed, only the numeric values.

What is the fitness of all possible genotypes? Here, order of events *per se* does not matter, beyond that considered in the poset. In other words, the fitness of genotype "a, b, c" is the same no matter how we got to "a, b, c". What matters is whether or not the genes on which each of "a", "b", and "c" depend are present or not (I only show the first 10 genotypes)

```
gcbn2 <- evalAllGenotypes(cbn2, order = FALSE)
gcbn2[1:10, ]
##     Genotype Birth
## 1          a 1.010
## 2          b 1.020
## 3          c 0.100
## 4          d 1.030
## 5          e 1.040
## 6          g 0.050
## 7       a, b 1.030
## 8       a, c 0.101
## 9       a, d 1.040
## 10      a, e 1.050
```

Of course, if we were to look at genotypes but taking into account order of occurrence of mutations, we would see no differences

```
gcbn2o <- evalAllGenotypes(cbn2, order = TRUE, max = 1956)
gcbn2o[1:10, ]
##     Genotype Birth
## 1          a 1.010
## 2          b 1.020
## 3          c 0.100
## 4          d 1.030
## 5          e 1.040
## 6          g 0.050
## 7      a > b 1.030
## 8      a > c 0.101
## 9      a > d 1.040
## 10     a > e 1.050
```

(The $max = 1956$ is there so that we show all the genotypes, even if they are more than 256, the default.)

You can check the output and verify things are as they should. For instance:

```
all.equal(
        gcbn2[c(1:21, 22, 28, 41, 44, 56, 63 ) , "Fitness"],
        c(1.01, 1.02, 0.1, 1.03, 1.04, 0.05,
          1.01 * c(1.02, 0.1, 1.03, 1.04, 0.05),
          1.02 * c(0.10, 1.03, 1.04, 0.05),
          0.1 * c(1.03, 1.04, 0.05),
          1.03 * c(1.04, 0.05),
          1.04 * 0.05,
          1.01 * 1.02 * 1.1,
          1.01 * 0.1 * 0.05,
          1.03 * 1.04 * 0.05,
          1.01 * 1.02 * 1.1 * 0.05,
```

```
          1.03 * 1.04 * 1.2 * 0.1, ## notice this
          1.01 * 1.02 * 1.03 * 1.04 * 1.1 * 1.2
          ))
## [1] "target is NULL, current is numeric"
```

A particular one that is important to understand is genotype with mutated genes "c, d, e, g":

```
gcbn2[56, ]
##      Genotype  Birth
## 56 c, d, e, g 0.1285
```

```
all.equal(gcbn2[56, "Fitness"], 1.03 * 1.04 * 1.2 * 0.10)
## [1] "target is NULL, current is numeric"
```

where "g" is taken as if its dependencies are satisfied (as "c", "d", and "e" are present) even when the dependencies of "c" are not satisfied (and that is why the term for "c" is 0.9).

3.4.6 DAGs: A semimonotone or "OR" example

We will reuse the above example, changing the type of relationship:

```
s1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.9, -.9), rep(-.95, 3)),
                 typeDep = "SM")

smn1 <- allFitnessEffects(s1)
```
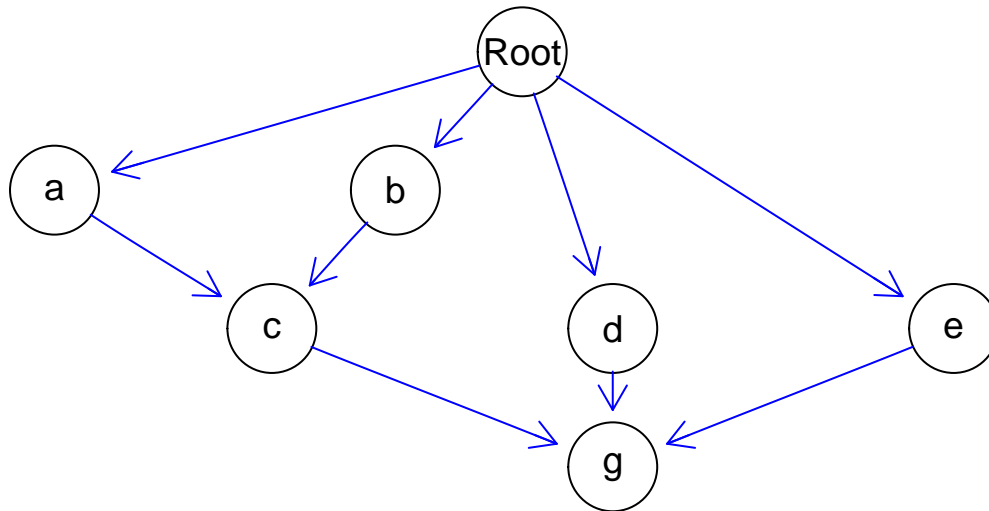
It looks like this (where edges are shown in blue to denote the semimonotone relationship):

```
plot(smn1)
```

```
gsmn1 <- evalAllGenotypes(smn1, order = FALSE)
```

Having just one parental dependency satisfied is now enough, in contrast to what happened before. For instance:

```
gcbn2[c(8, 12, 22), ]
##    Genotype Birth
## 8      a, c 0.101
## 12     b, c 0.102
## 22  a, b, c 1.133
```

```
gsmn1[c(8, 12, 22), ]
##    Genotype Birth
## 8      a, c 1.111
## 12     b, c 1.122
## 22  a, b, c 1.133
```

```
gcbn2[c(20:21, 28), ]
##    Genotype   Birth
## 20     d, g 0.05150
## 21     e, g 0.05200
## 28  a, c, g 0.00505
```

```
gsmn1[c(20:21, 28), ]
##    Genotype Birth
## 20     d, g 1.236
## 21     e, g 1.248
## 28  a, c, g 1.333
```

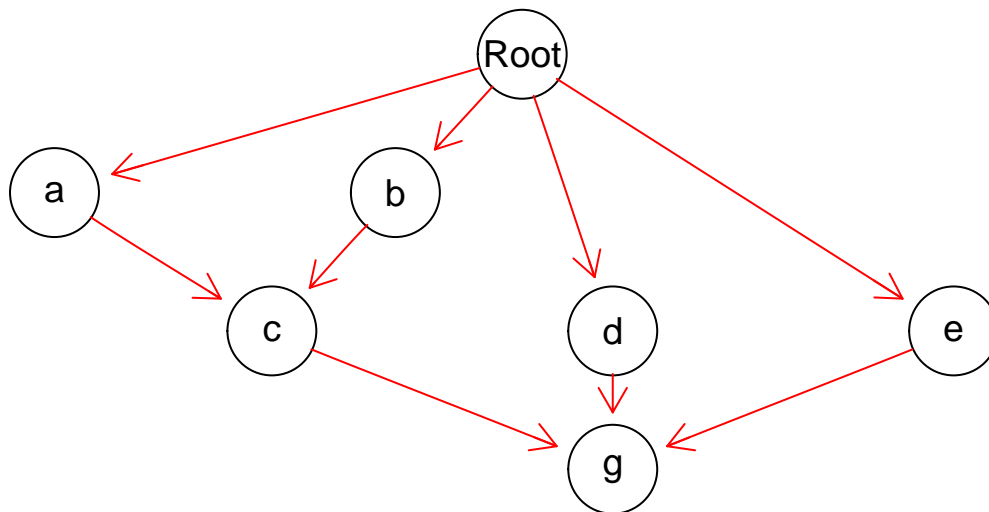### 3.4.7 An "XMPN" or "XOR" example

Again, we reuse the example above, changing the type of relationship:

```
x1 <- data.frame(parent = c(rep("Root", 4), "a", "b", "d", "e", "c"),
                 child = c("a", "b", "d", "e", "c", "c", rep("g", 3)),
                 s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, rep(0.2, 3)),
                 sh = c(rep(0, 4), c(-.9, -.9), rep(-.95, 3)),
                 typeDep = "XMPN")

xor1 <- allFitnessEffects(x1)
```

It looks like this (edges in red to denote the "XOR" relationship):

```
plot(xor1)
```



```
gxor1 <- evalAllGenotypes(xor1, order = FALSE)
```

Whenever "c" is present with both "a" and "b", the fitness component for "c" will be $(1 - 0.1)$. Similarly for "g" (if more than one of "d", "e", or "c" is present, it will show as $(1 - 0.05)$). For example:

```
gxor1[c(22, 41), ]
##     Genotype   Birth
## 22  a, b, c 0.10302
## 41  d, e, g 0.05356
```

```
c(1.01 * 1.02 * 0.1, 1.03 * 1.04 * 0.05)
## [1] 0.10302 0.05356
```

However, having just both "a" and "b" is identical to the case with CBN and the monotone relationship (see sections 3.4.5 and 3.4.6). If you want the joint presence of "a" and "b" to result in different fitness than the product of the individual terms, without considering the presence of "c", you can specify that using general epistatic effects (section 3.7).

We also see a very different pattern compared to CBN (section 3.4.5) here:

```
gxor1[28, ]
##    Genotype Birth
## 28  a, c, g 1.333
```

```
1.01 * 1.1 * 1.2
## [1] 1.333
```

as exactly one of the dependencies for both "c" and "g" are satisfied.

But

```
gxor1[44, ]
##       Genotype  Birth
## 44 a, b, c, g 0.1236
```

```
1.01 * 1.02 * 0.1 * 1.2
## [1] 0.1236
```

is the result of a 0.1 for "c" (and a 1.2 for "g" that has exactly one of its dependencies satisfied).

3.4.8   Posets: the three types of relationships

```
p3 <- data.frame(
    parent = c(rep("Root", 4), "a", "b", "d", "e", "c", "f"),
    child = c("a", "b", "d", "e", "c", "c", "f", "f", "g", "g"),
    s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
    sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
    typeDep = c(rep("--", 4),
                "XMPN", "XMPN", "MN", "MN", "SM", "SM"))
fp3 <- allFitnessEffects(p3)
```

This is how it looks like:

```
plot(fp3)
```
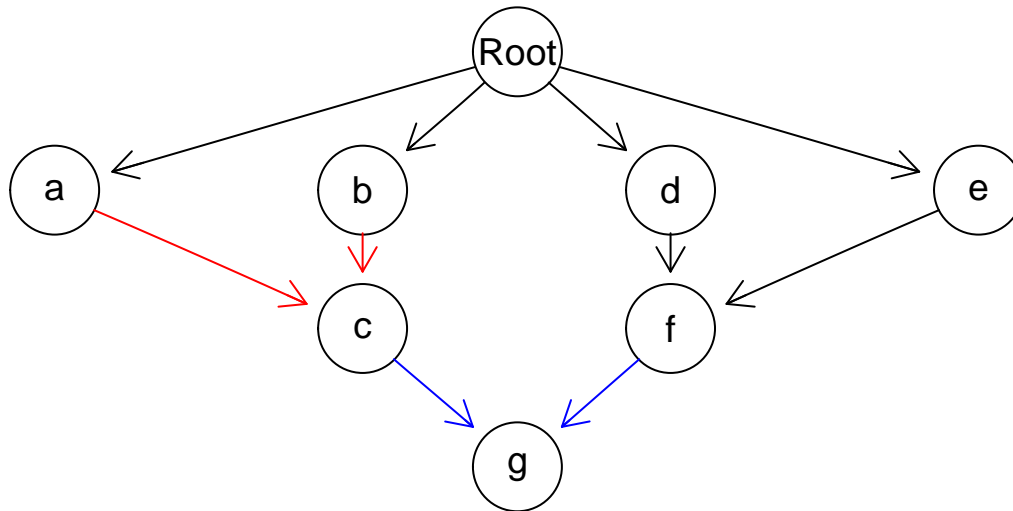
We can also use "igraph":

```
plot(fp3, "igraph", layout.reingold.tilford)
```



```
gfp3 <- evalAllGenotypes(fp3, order = FALSE)
```

Let's look at a few:

```
gfp3[c(9, 24, 29, 59, 60, 66, 119, 120, 126, 127), ]
##              Genotype     Birth
## 9               a, c 1.111000
## 24              d, f 0.051500
## 29           a, b, c 0.103020
```
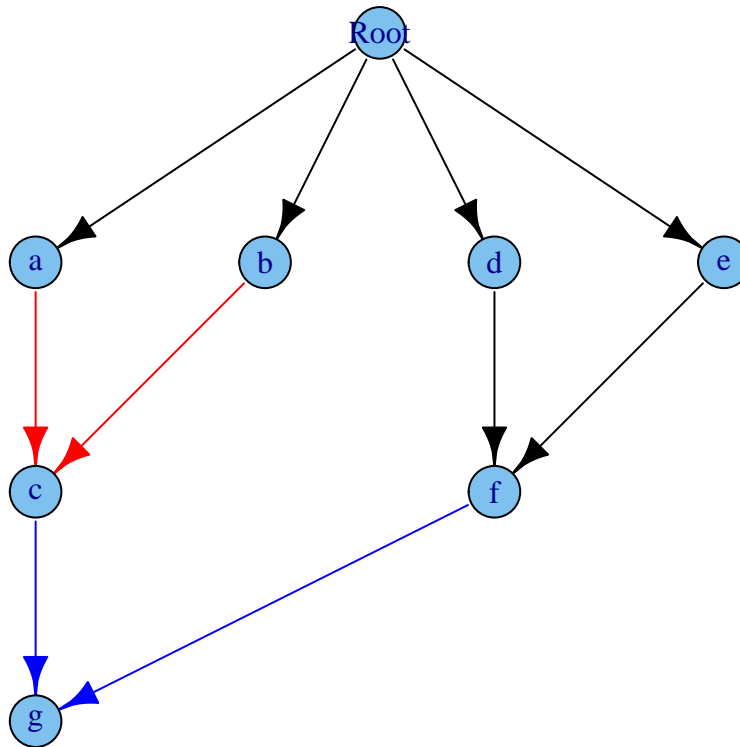
```
## 59              c, f, g 0.006500
## 60              d, e, f 1.285440
## 66           a, b, c, f 0.005151
## 119       c, d, e, f, g 0.167107
## 120    a, b, c, d, e, f 0.132426
## 126    b, c, d, e, f, g 1.874943
## 127 a, b, c, d, e, f, g 0.172154
```

```r
c(1.01 * 1.1, 1.03 * .05, 1.01 * 1.02 * 0.1, 0.1 * 0.05 * 1.3,
  1.03 * 1.04 * 1.2, 1.01 * 1.02 * 0.1 * 0.05,
  0.1 * 1.03 * 1.04 * 1.2 * 1.3,
  1.01 * 1.02 * 0.1 * 1.03 * 1.04 * 1.2,
  1.02 * 1.1 * 1.03 * 1.04 * 1.2 * 1.3,
  1.01 * 1.02 * 1.03 * 1.04 * 0.1 * 1.2 * 1.3)
## [1] 1.111000 0.051500 0.103020 0.006500 1.285440 0.005151 0.167107
## [8] 0.132426 1.874943 0.172154
```

As before, looking at the order of mutations makes no difference (look at the test directory to see a test that verifies this assertion).

3.5 Modules

As already mentioned, we can think of all the effects of fitness in terms not of individual genes but, rather, modules. This idea is discussed in, for example, Raphael & Vandin (2015), Gerstung, Eriksson, et al. (2011): the restrictions encoded in, say, the DAGs can be considered to apply not to genes, but to modules, where each module is a set of genes (and the intersection between modules is the empty set). Modules, then, play the role of a "union operation" over sets of genes. Of course, if we can use modules for the restrictions in the DAGs we should also be able to use them for epistasis and order effects, as we will see later (e.g., 3.6.2).

3.5.1 What does a module provide

Modules can provide very compact ways of specifying relationships when you want to, well, model the existence of modules. For simplicity suppose there is a module, "A", made of genes "a1" and "a2", and a module "B", made of a single gene "b1". Module "B" can mutate if module "A" is mutated, but mutating both "a1" and "a2" provides no additional fitness advantage compared to mutating only a single one of them. We can specify this as:

```r
s <- 0.2
sboth <- (1/(1 + s)) - 1
m0 <- allFitnessEffects(data.frame(
    parent = c("Root", "Root", "a1", "a2"),
    child = c("a1", "a2", "b", "b"),
    s = s,
    sh = -1,
```

```
    typeDep = "OR"),
                        epistasis = c("a1:a2" = sboth))
evalAllGenotypes(m0, order = FALSE, addwt = TRUE)
##     Genotype Birth
## 1        WT  1.00
## 2        a1  1.20
## 3        a2  1.20
## 4         b  0.00
## 5    a1, a2  1.20
## 6     a1, b  1.44
## 7     a2, b  1.44
## 8 a1, a2, b  1.44
```

Note that we need to add an epistasis term, with value "sboth" to capture the idea of "mutating both"a1" and "a2" provides no additional fitness advantage compared to mutating only a single one of them"; see details in section 3.7.

Now, specify it using modules:

```
s <- 0.2
m1 <- allFitnessEffects(data.frame(
    parent = c("Root", "A"),
    child = c("A", "B"),
    s = s,
    sh = -1,
    typeDep = "OR"),
                        geneToModule = c("Root" = "Root",
                                         "A" = "a1, a2",
                                         "B" = "b1"))
evalAllGenotypes(m1, order = FALSE, addwt = TRUE)
##      Genotype Birth
## 1         WT  1.00
## 2         a1  1.20
## 3         a2  1.20
## 4         b1  0.00
## 5     a1, a2  1.20
## 6     a1, b1  1.44
## 7     a2, b1  1.44
## 8 a1, a2, b1  1.44
```

This captures the ideas directly. The typing savings here are small, but they can be large with modules with many genes.

3.5.2 Specifying modules

How do you specify modules? The general procedure is simple: you pass a vector that makes explicit the mapping from modules to sets of genes. We just saw an example. There are several additional examples such as 3.5.3, 3.6.2, 3.7.4.

It is important to note that, once you specify modules, we expect all of the relationships (except those that involve the non interacting genes) to be specified as modules. Thus, all elements of the epistasis, posets (the DAGs) and order effects components should be specified in terms of modules. But you can, of course, specify a module as containing a single gene (and a single gene with the same name as the module).

What about the "Root" node? If you use a "restriction table", that restriction table (that DAG) must have a node named "Root" and in the mapping of genes to module there **must** be a first entry that has a module and gene named "Root", as we saw above with `geneToModule  = c("Root" = "Root", ....` We force you to do this to be explicit about the "Root" node. This is not needed (thought it does not hurt) with other fitness specifications. For instance, if we have a model with two modules, one of them with two genes (see details in section 3.8) we do not need to pass a "Root" as in

```
fnme <- allFitnessEffects(epistasis = c("A" = 0.1,
                                        "B" = 0.2),
                          geneToModule = c("A" = "a1, a2",
                                           "B" = "b1"))
evalAllGenotypes(fnme, order = FALSE, addwt = TRUE)
##      Genotype Birth
## 1          WT  1.00
## 2          a1  1.10
## 3          a2  1.10
## 4          b1  1.20
## 5      a1, a2  1.10
## 6      a1, b1  1.32
## 7      a2, b1  1.32
## 8 a1, a2, b1  1.32
```

but it is also OK to have a "Root" in the `geneToModule`:

```
fnme2 <- allFitnessEffects(epistasis = c("A" = 0.1,
                                         "B" = 0.2),
                           geneToModule = c(
                               "Root" = "Root",
                               "A" = "a1, a2",
                               "B" = "b1"))
evalAllGenotypes(fnme, order = FALSE, addwt = TRUE)
##      Genotype Birth
## 1          WT  1.00
## 2          a1  1.10
## 3          a2  1.10
## 4          b1  1.20
## 5      a1, a2  1.10
## 6      a1, b1  1.32
## 7      a2, b1  1.32
## 8 a1, a2, b1  1.32
```

3.5.3 Modules and posets again: the three types of relationships and modules

We use the same specification of poset, but add modules. To keep it manageable, we only add a few genes for some modules, and have some modules with a single gene. Beware that the number of genotypes is starting to grow quite fast, though. We capitalize to differentiate modules (capital letters) from genes (lowercase with a number), but this is not needed.

```
p4 <- data.frame(
    parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
    child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
    s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
    sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
    typeDep = c(rep("--", 4),
                "XMPN", "XMPN", "MN", "MN", "SM", "SM"))

fp4m <- allFitnessEffects(
    p4,
    geneToModule = c("Root" = "Root", "A" = "a1",
                     "B" = "b1, b2", "C" = "c1",
                     "D" = "d1, d2", "E" = "e1",
                     "F" = "f1, f2", "G" = "g1"))
```

By default, plotting shows the modules:

```
plot(fp4m)
```
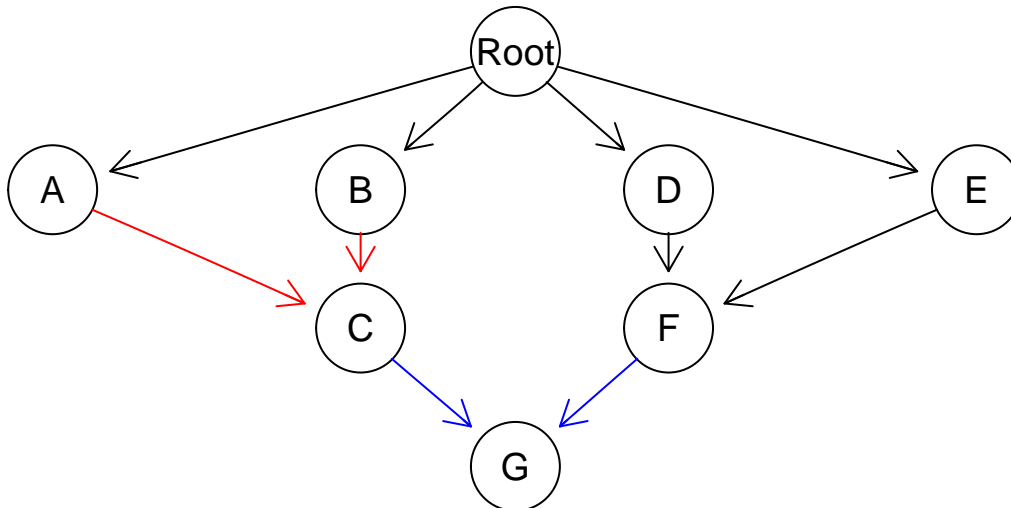

but we can show the gene names instead of the module names:

```
plot(fp4m, expandModules = TRUE)
```
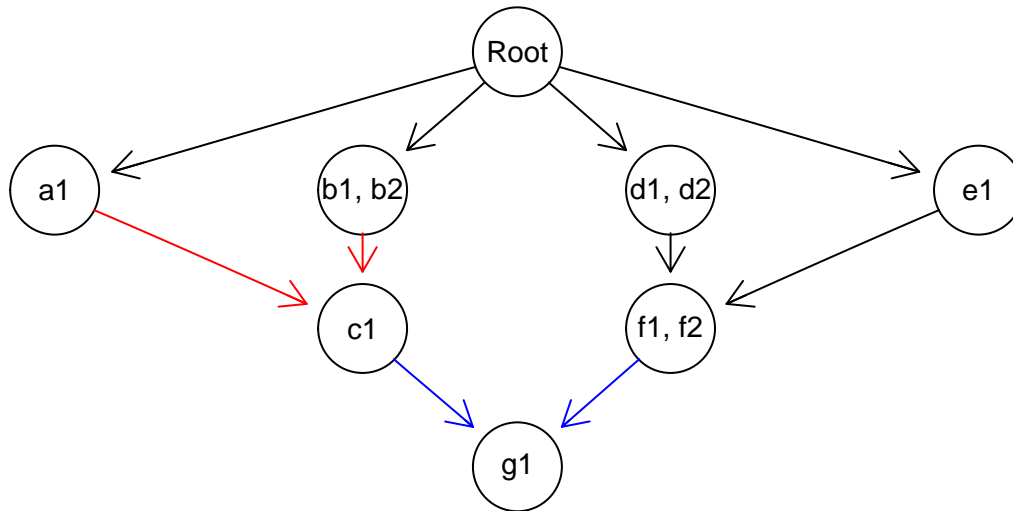
or

```
plot(fp4m, "igraph", layout = layout.reingold.tilford,
     expandModules = TRUE)
```
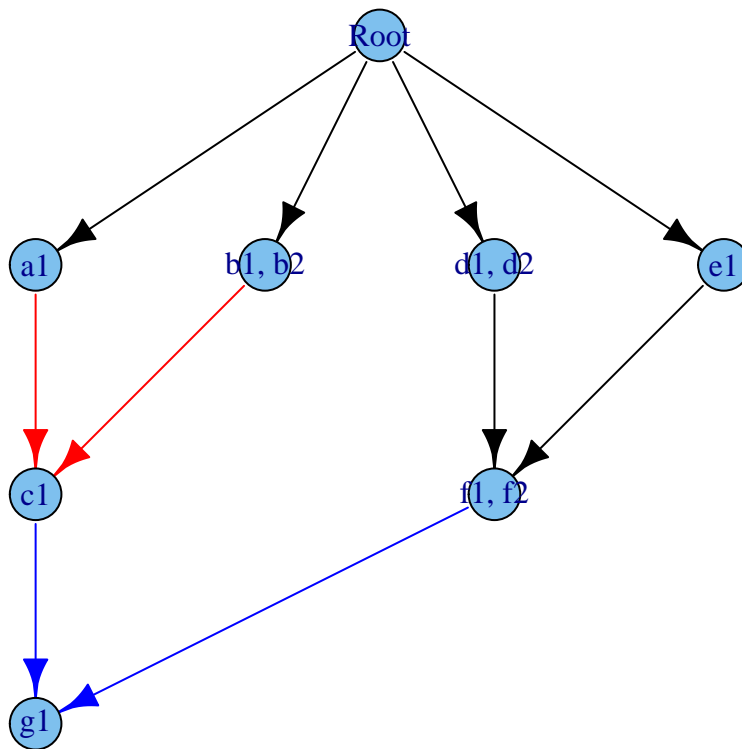

We obtain the fitness of all genotypes in the usual way:

```
gfp4 <- evalAllGenotypes(fp4m, order = FALSE, max = 1024)
```

Let's look at a few of those:

```
gfp4[c(12, 20, 21, 40, 41, 46, 50, 55, 64, 92,
       155, 157, 163, 372, 632, 828), ]
##                     Genotype  Birth
```

```
## 12                   a1, b2 1.0302
## 20                   b1, b2 1.0200
## 21                   b1, c1 1.1220
## 40                   c1, g1 0.1300
## 41                   d1, d2 1.0300
## 46                   d2, e1 1.0712
## 50                   e1, f1 0.0520
## 55                   f2, g1 0.0650
## 64             a1, b2, c1 0.1030
## 92             b1, b2, c1 1.1220
## 155            c1, f2, g1 0.0065
## 157            d1, d2, f1 0.0515
## 163            d1, f1, f2 0.0515
## 372        d1, d2, e1, f2 1.2854
## 632     d1, d2, e1, f1, f2 1.2854
## 828 b2, c1, d1, e1, f2, g1 1.8749
```

```
c(1.01 * 1.02, 1.02, 1.02 * 1.1, 0.1 * 1.3, 1.03,
  1.03 * 1.04, 1.04 * 0.05, 0.05 * 1.3,
  1.01 * 1.02 * 0.1, 1.02 * 1.1, 0.1 * 0.05 * 1.3,
  1.03 * 0.05, 1.03 * 0.05, 1.03 * 1.04 * 1.2, 1.03 * 1.04 * 1.2,
  1.02 * 1.1 * 1.03 * 1.04 * 1.2 * 1.3)
##  [1] 1.0302 1.0200 1.1220 0.1300 1.0300 1.0712 0.0520 0.0650 0.1030
## [10] 1.1220 0.0065 0.0515 0.0515 1.2854 1.2854 1.8749
```

3.6   Order effects

As explained in the introduction (section 1), by order effects we mean a phenomenon such as the one shown empirically by Ortmann et al. (2015): the fitness of a double mutant "A", "B" is different depending on whether "A" was acquired before "B" or "B" before "A". This, of course, can be generalized to more than two genes.

Note that order effects are different from the restrictions in the order of accumulation of mutations discussed in section 3.4. With restrictions in the order of accumulation of mutations we might say that acquiring "B" depends or is facilitated by having "A" mutated (and, unless we allowed for multiple mutations, having "A" mutated means having "A" mutated before "B"). However, once you have the genotype "A, B", its fitness does not depend on the order in which "A" and "B" appeared.

3.6.1   Order effects: three-gene orders

Consider this case, where three specific three-gene orders and two two-gene orders (one of them a subset of one of the three) lead to different fitness compared to the wild-type. We add also modules, to show its usage (but just limit ourselves to using one gene per module here).

Order effects are specified using a $x > y$, which means that that order effect is

satisfied when module x is mutated before module y.

```
o3 <- allFitnessEffects(orderEffects = c(
                        "F > D > M" = -0.3,
                        "D > F > M" = 0.4,
                        "D > M > F" = 0.2,
                        "D > M"     = 0.1,
                        "M > D"     = 0.5),
                 geneToModule =
                     c("M" = "m",
                       "F" = "f",
                       "D" = "d") )



(ag <- evalAllGenotypes(o3, addwt = TRUE, order = TRUE))
##       Genotype Birth
## 1           WT  1.00
## 2            d  1.00
## 3            f  1.00
## 4            m  1.00
## 5        d > f  1.00
## 6        d > m  1.10
## 7        f > d  1.00
## 8        f > m  1.00
## 9        m > d  1.50
## 10       m > f  1.00
## 11 d > f > m  1.54
## 12 d > m > f  1.32
## 13 f > d > m  0.77
## 14 f > m > d  1.50
## 15 m > d > f  1.50
## 16 m > f > d  1.50
```

(The meaning of the notation in the output table is as follows: "WT" denotes the wild-type, or non-mutated clone. The notation $x > y$ means that a mutation in "x" happened before a mutation in "y". A genotype $x > y \_ z$ means that a mutation in "x" happened before a mutation in "y"; there is also a mutation in "z", but that is a gene for which order does not matter).

The values for the first nine genotypes come directly from the fitness specifications. The 10th genotype matches $D > F > M$ $(= (1 + 0.4))$ but also $D > M$ $((1 + 0.1))$. The 11th matches $D > M > F$ and $D > M$. The 12th matches $F > D > M$ but also $D > M$. Etc.

3.6.2 Order effects and modules with multiple genes

Consider the following case:

```
ofe1 <- allFitnessEffects(
    orderEffects = c("F > D" = -0.3, "D > F" = 0.4),
    geneToModule =
        c("F" = "f1, f2",
          "D" = "d1, d2") )

ag <- evalAllGenotypes(ofe1, order = TRUE)
```

There are four genes, $d1, d2, f1, f2$, where each d belongs to module D and each f belongs to module F.

What to expect for cases such as $d1 > f1$ or $f1 > d1$ is clear, as shown in

```
ag[5:16,]
##     Genotype Birth
## 5    d1 > d2   1.0
## 6    d1 > f1   1.4
## 7    d1 > f2   1.4
## 8    d2 > d1   1.0
## 9    d2 > f1   1.4
## 10   d2 > f2   1.4
## 11   f1 > d1   0.7
## 12   f1 > d2   0.7
## 13   f1 > f2   1.0
## 14   f2 > d1   0.7
## 15   f2 > d2   0.7
## 16   f2 > f1   1.0
```

Likewise, cases such as $d1 > d2 > f1$ or $f2 > f1 > d1$ are clear, because in terms of modules they map to $ D > F$ or $F > D$: the observed order of mutation $d1 > d2 > f1$ means that module D was mutated first and module F was mutated second. Similar for $d1 > f1 > f2$ or $f1 > d1 > d2$: those map to $D > F$ and $F > D$. We can see the fitness of those four case in:

```
ag[c(17, 39, 19, 29), ]
##           Genotype Birth
## 17 d1 > d2 > f1   1.4
## 39 f2 > f1 > d1   0.7
## 19 d1 > f1 > d2   1.4
## 29 f1 > d1 > d2   0.7
```

and they correspond to the values of those order effects, where $F > D = (1 - 0.3)$ and $D > F = (1 + 0.4)$:

```
ag[c(17, 39, 19, 29), "Fitness"] == c(1.4, 0.7, 1.4, 0.7)
## logical(0)
```

What if we match several patterns? For example, $d1 > f1 > d2 > f2$ and $d1 > f1 > f2 > d2$? The first maps to $D > F > D > F$ and the second to $D > F > D$. But

109

since we are concerned with which one happened first and which happened second we should expect those two to correspond to the same fitness, that of pattern $D > F$, as is the case:

```
ag[c(43, 44),]
##               Genotype Birth
## 43 d1 > f1 > d2 > f2   1.4
## 44 d1 > f1 > f2 > d2   1.4
```

```
ag[c(43, 44), "Fitness"] == c(1.4, 1.4)
## logical(0)
```

More generally, that applies to all the patterns that start with one of the "d" genes:

```
all(ag[41:52, "Fitness"] == 1.4)
## [1] TRUE
```

Similar arguments apply to the opposite pattern, $F > D$, which apply to all the possible gene mutation orders that start with one of the "f" genes. For example:

```
all(ag[53:64, "Fitness"] == 0.7)
## [1] TRUE
```

3.6.3   Order and modules with 325 genotypes

We can of course have more than two genes per module. This just repeats the above, with five genes (there are 325 genotypes, and that is why we pass the "max" argument to `evalAllGenotypes`, to allow for more than the default 256).

```
ofe2 <- allFitnessEffects(
    orderEffects = c("F > D" = -0.3, "D > F" = 0.4),
    geneToModule =
        c("F" = "f1, f2, f3",
          "D" = "d1, d2") )
ag2 <- evalAllGenotypes(ofe2, max = 325, order = TRUE)
```

We can verify that any combination that starts with a "d" gene and then contains at least one "f" gene will have a fitness of $1 + 0.4$. And any combination that starts with an "f" gene and contains at least one "d" genes will have a fitness of $1 - 0.3$. All other genotypes have a fitness of 1:

```
all(ag2[grep("^d.*f.*", ag2[, 1]), "Fitness"] == 1.4)
## [1] TRUE
```

```
all(ag2[grep("^f.*d.*", ag2[, 1]), "Fitness"] == 0.7)
## [1] TRUE
```

```
oe <- c(grep("^f.*d.*", ag2[, 1]), grep("^d.*f.*", ag2[, 1]))
all(ag2[-oe, "Fitness"] == 1)
## [1] TRUE
```

110

3.6.4   Order effects and genes without interactions

We will now look at both order effects and interactions. To make things more interesting, we name genes so that the ordered names do split nicely between those with and those without order effects (this, thus, also serves as a test of messy orders of names).

```
foi1 <- allFitnessEffects(
    orderEffects = c("D>B" = -0.2, "B > D" = 0.3),
    noIntGenes = c("A" = 0.05, "C" = -.2, "E" = .1))
```

You can get a verbose view of what the gene names and modules are (and their automatically created numeric codes) by:

```
foi1[c("geneModule", "long.geneNoInt")]
## $geneModule
##   Gene Module GeneNumID ModuleNumID
## 1 Root   Root         0           0
## 2    B      B         1           1
## 3    D      D         2           2
##
## $long.geneNoInt
##   Gene GeneNumID     s
## A    A         3  0.05
## C    C         4 -0.20
## E    E         5  0.10
```

We can get the fitness of all genotypes (we set $max = 325$ because that is the number of possible genotypes):

```
agoi1 <- evalAllGenotypes(foi1,  max = 325, order = TRUE)
head(agoi1)
##   Genotype Birth
## 1        B  1.00
## 2        D  1.00
## 3        A  1.05
## 4        C  0.80
## 5        E  1.10
## 6    B > D  1.30
```

Now:

```
rn <- 1:nrow(agoi1)
names(rn) <- agoi1[, 1]

agoi1[rn[LETTERS[1:5]], "Fitness"] == c(1.05, 1, 0.8, 1, 1.1)
## logical(0)
```

According to the fitness effects we have specified, we also know that any genotype with only two mutations, one of which is either "A", "C" "E" and the other is "B"

or "D" will have the fitness corresponding to "A", "C" or "E", respectively:

```
agoi1[grep("^A > [BD]$", names(rn)), "Fitness"] == 1.05
## logical(0)
```

```
agoi1[grep("^C > [BD]$", names(rn)), "Fitness"] == 0.8
## logical(0)
```

```
agoi1[grep("^E > [BD]$", names(rn)), "Fitness"] == 1.1
## logical(0)
```

```
agoi1[grep("^[BD] > A$", names(rn)), "Fitness"] == 1.05
## logical(0)
```

```
agoi1[grep("^[BD] > C$", names(rn)), "Fitness"] == 0.8
## logical(0)
```

```
agoi1[grep("^[BD] > E$", names(rn)), "Fitness"] == 1.1
## logical(0)
```

We will not be playing many additional games with regular expressions, but let us check those that start with "D" and have all the other mutations, which occupy rows 230 to 253; fitness should be equal (within numerical error, because of floating point arithmetic) to the order effect of "D" before "B" times the other effects $(1 - 0.3) * 1.05 * 0.8 * 1.1 = 0.7392$

```
all.equal(agoi1[230:253, "Fitness"] ,
          rep((1 - 0.2) * 1.05 * 0.8 * 1.1, 24))
## [1] "target is NULL, current is numeric"
```

and that will also be the value of any genotype with the five mutations where "D" comes before "B" such as those in rows 260 to 265, 277, or 322 and 323, but it will be equal to $(1 + 0.3) * 1.05 * 0.8 * 1.1 = 1.2012$ in those where "B" comes before "D". Analogous arguments apply to four, three, and two mutation genotypes.

3.7 Epistasis

3.7.1 Epistasis: two alternative specifications

We want the following mapping of genotypes to fitness:

A	B	Fitness
wt	wt	1
wt	M	$1 + s_b$
M	wt	$1 + s_a$
M	M	$1 + s_{ab}$

Suppose that the actual numerical values are $s_a = 0.2, s_b = 0.3, s_{ab} = 0.7$.

We specify the above as follows:

```
sa <- 0.2
sb <- 0.3
sab <- 0.7

e2 <- allFitnessEffects(epistasis =
                            c("A: -B" = sa,
                              "-A:B" = sb,
                              "A : B" = sab))
evalAllGenotypes(e2, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT   1.0
## 2         A   1.2
## 3         B   1.3
## 4      A, B   1.7
```

That uses the "-" specification, so we explicitly exclude some patterns: with "A:-B" we say "A when there is no B".

But we can also use a specification where we do not use the "-". That requires a different numerical value of the interaction, because now, as we are rewriting the interaction term as genotype "A is mutant, B is mutant" the double mutant will incorporate the effects of "A mutant", "B mutant" and "both A and B mutants". We can define a new s_2 that satisfies $(1 + s_{ab}) = (1 + s_a)(1 + s_b)(1 + s_2)$ so $(1 + s_2) = (1 + s_{ab})/((1 + s_a)(1 + s_b))$ and therefore specify as:

```
s2 <- ((1 + sab)/((1 + sa) * (1 + sb))) - 1

e3 <- allFitnessEffects(epistasis =
                            c("A" = sa,
                              "B" = sb,
                              "A : B" = s2))
evalAllGenotypes(e3, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT   1.0
## 2         A   1.2
## 3         B   1.3
## 4      A, B   1.7
```

Note that this is the way you would specify effects with FFPopsim (Zanini & Neher, 2012). Whether this specification or the previous one with "-" is simpler will depend on the model. For synthetic mortality and viability, I think the one using "-" is simpler to map genotype tables to fitness effects. See also section 3.7.2 and 3.7.3 and the example in section 5.4.2.

Finally, note that we can also specify some of these effects by combining the graph and the epistasis, as shown in section 5.2.1 or 5.4.2.

3.7.2 Epistasis with three genes and two alternative specifications

Suppose we have

A	B	C	Fitness
M	wt	wt	$1 + s_a$
wt	M	wt	$1 + s_b$
wt	wt	M	$1 + s_c$
M	M	wt	$1 + s_{ab}$
wt	M	M	$1 + s_{bc}$
M	wt	M	$(1 + s_a)(1 + s_c)$
M	M	M	$1 + s_{abc}$

where missing rows have a fitness of 1 (they have been deleted for conciseness). Note that the mutant for exactly A and C has a fitness that is the product of the individual terms (so there is no epistasis in that case).

```
sa <- 0.1
sb <- 0.15
sc <- 0.2
sab <- 0.3
sbc <- -0.25
sabc <- 0.4

sac <- (1 + sa) * (1 + sc) - 1

E3A <- allFitnessEffects(epistasis =
                         c("A:-B:-C" = sa,
                           "-A:B:-C" = sb,
                           "-A:-B:C" = sc,
                           "A:B:-C" = sab,
                           "-A:B:C" = sbc,
                           "A:-B:C" = sac,
                           "A : B : C" = sabc)
                         )

evalAllGenotypes(E3A, order = FALSE, addwt = FALSE)
##    Genotype Birth
## 1         A  1.10
## 2         B  1.15
## 3         C  1.20
## 4      A, B  1.30
## 5      A, C  1.32
## 6      B, C  0.75
## 7   A, B, C  1.40
```

We needed to pass the s_{ac} coefficient explicitly, even if it that term was just the

product. We can try to avoid using the "-", however (but we will need to do other calculations). For simplicity, I use capital "S" in what follows where the letters differ from the previous specification:

```
sa <- 0.1
sb <- 0.15
sc <- 0.2
sab <- 0.3
Sab <- ( (1 + sab)/((1 + sa) * (1 + sb))) - 1
Sbc <- ( (1 + sbc)/((1 + sb) * (1 + sc))) - 1
Sabc <- ( (1 + sabc)/
          ( (1 + sa) * (1 + sb) * (1 + sc) *
            (1 + Sab) * (1 + Sbc) ) ) - 1


E3B <- allFitnessEffects(epistasis =
                             c("A" = sa,
                               "B" = sb,
                               "C" = sc,
                               "A:B" = Sab,
                               "B:C" = Sbc,
                               ## "A:C" = sac, ## not needed now
                               "A : B : C" = Sabc)
                                            )
evalAllGenotypes(E3B, order = FALSE, addwt = FALSE)
##     Genotype Birth
## 1          A  1.10
## 2          B  1.15
## 3          C  1.20
## 4       A, B  1.30
## 5       A, C  1.32
## 6       B, C  0.75
## 7    A, B, C  1.40
```

The above two are, of course, identical:

```
all(evalAllGenotypes(E3A, order = FALSE, addwt = FALSE) ==
    evalAllGenotypes(E3B, order = FALSE, addwt = FALSE))
## [1] TRUE
```

We avoid specifying the "A:C", as it just follows from the individual "A" and "C" terms, but given a specified genotype table, we need to do a little bit of addition and multiplication to get the coefficients.

3.7.3   Why can we specify some effects with a "-"?

Let's suppose we want to specify the synthetic viability example seen before:

A	B	Fitness
wt	wt	1
wt	M	0
M	wt	0
M	M	$(1 + s)$

where "wt" denotes wild type and "M" denotes mutant.

If you want to directly map the above table to the fitness table for the program, to specify the genotype "A is wt, B is a mutant" you can specify it as "-A,B", not just as "B". Why? Because just the presence of a "B" is also compatible with genotype "A is mutant and B is mutant". If you use "-" you are explicitly saying what should not be there so that "-A,B" is NOT compatible with "A, B". Otherwise, you need to carefully add coefficients. Depending on what you are trying to model, different specifications might be simpler. See the examples in section 3.7.1 and 3.7.2. You have both options.

3.7.4 Epistasis: modules

There is nothing conceptually new, but we will show an example here:

```
sa <- 0.2
sb <- 0.3
sab <- 0.7

em <- allFitnessEffects(epistasis =
                     c("A: -B" = sa,
                       "-A:B" = sb,
                       "A : B" = sab),
                   geneToModule = c("A" = "a1, a2",
                                    "B" = "b1, b2"))
evalAllGenotypes(em, order = FALSE, addwt = TRUE)
##            Genotype Birth
## 1                WT   1.0
## 2                a1   1.2
## 3                a2   1.2
## 4                b1   1.3
## 5                b2   1.3
## 6          a1, a2   1.2
## 7          a1, b1   1.7
## 8          a1, b2   1.7
## 9          a2, b1   1.7
## 10         a2, b2   1.7
## 11         b1, b2   1.3
## 12     a1, a2, b1   1.7
## 13     a1, a2, b2   1.7
```

```
## 14    a1, b1, b2   1.7
## 15    a2, b1, b2   1.7
## 16 a1, a2, b1, b2   1.7
```

Of course, we can do the same thing without using the "-", as in section 3.7.1:

```
s2 <- ((1 + sab)/((1 + sa) * (1 + sb))) - 1


em2 <- allFitnessEffects(epistasis =
                             c("A" = sa,
                               "B" = sb,
                               "A : B" = s2),
                           geneToModule = c("A" = "a1, a2",
                                            "B" = "b1, b2")
                         )
evalAllGenotypes(em2, order = FALSE, addwt = TRUE)
##           Genotype Birth
## 1               WT   1.0
## 2               a1   1.2
## 3               a2   1.2
## 4               b1   1.3
## 5               b2   1.3
## 6           a1, a2   1.2
## 7           a1, b1   1.7
## 8           a1, b2   1.7
## 9           a2, b1   1.7
## 10          a2, b2   1.7
## 11          b1, b2   1.3
## 12      a1, a2, b1   1.7
## 13      a1, a2, b2   1.7
## 14      a1, b1, b2   1.7
## 15      a2, b1, b2   1.7
## 16 a1, a2, b1, b2   1.7
```

3.8 I do not want epistasis, but I want modules!

Sometimes you might want something like having several modules, say "A" and "B", each with a number of genes, but with "A" and "B" showing no interaction.

It is a terminological issue whether we should allow `noIntGenes` (no interaction genes), as explained in section 3.3 to actually be modules. The reasoning for not allowing them is that the situation depicted above (several genes in module A, for example) actually is one of interaction: the members of "A" are combined using an "OR" operator (i.e., the fitness consequences of having one or more genes of A mutated are the same), not just simply multiplying their fitness; similarly for "B". This is why no interaction genes also mean no modules allowed.

So how do you get what you want in this case? Enter the names of the modules

in the `epistasis` component but have no term for ":" (the colon). Let's see an example:

```
fnme <- allFitnessEffects(epistasis = c("A" = 0.1,
                                        "B" = 0.2),
                          geneToModule = c("A" = "a1, a2",
                                           "B" = "b1, b2, b3"))

evalAllGenotypes(fnme, order = FALSE, addwt = TRUE)
##                 Genotype Birth
## 1                     WT  1.00
## 2                     a1  1.10
## 3                     a2  1.10
## 4                     b1  1.20
## 5                     b2  1.20
## 6                     b3  1.20
## 7                 a1, a2  1.10
## 8                 a1, b1  1.32
## 9                 a1, b2  1.32
## 10                a1, b3  1.32
## 11                a2, b1  1.32
## 12                a2, b2  1.32
## 13                a2, b3  1.32
## 14                b1, b2  1.20
## 15                b1, b3  1.20
## 16                b2, b3  1.20
## 17            a1, a2, b1  1.32
## 18            a1, a2, b2  1.32
## 19            a1, a2, b3  1.32
## 20            a1, b1, b2  1.32
## 21            a1, b1, b3  1.32
## 22            a1, b2, b3  1.32
## 23            a2, b1, b2  1.32
## 24            a2, b1, b3  1.32
## 25            a2, b2, b3  1.32
## 26            b1, b2, b3  1.20
## 27        a1, a2, b1, b2  1.32
## 28        a1, a2, b1, b3  1.32
## 29        a1, a2, b2, b3  1.32
## 30        a1, b1, b2, b3  1.32
## 31        a2, b1, b2, b3  1.32
## 32 a1, a2, b1, b2, b3  1.32
```

In previous versions these was possible using the longer, still accepted way of specifying a : with a value of 0, but this is no longer needed:

```r
fnme <- allFitnessEffects(epistasis = c("A" = 0.1,
                                        "B" = 0.2,
                                        "A : B" = 0.0),
                          geneToModule = c("A" = "a1, a2",
                                           "B" = "b1, b2, b3"))


evalAllGenotypes(fnme, order = FALSE, addwt = TRUE)
##                 Genotype Birth
## 1                    WT  1.00
## 2                    a1  1.10
## 3                    a2  1.10
## 4                    b1  1.20
## 5                    b2  1.20
## 6                    b3  1.20
## 7                a1, a2  1.10
## 8                a1, b1  1.32
## 9                a1, b2  1.32
## 10               a1, b3  1.32
## 11               a2, b1  1.32
## 12               a2, b2  1.32
## 13               a2, b3  1.32
## 14               b1, b2  1.20
## 15               b1, b3  1.20
## 16               b2, b3  1.20
## 17           a1, a2, b1  1.32
## 18           a1, a2, b2  1.32
## 19           a1, a2, b3  1.32
## 20           a1, b1, b2  1.32
## 21           a1, b1, b3  1.32
## 22           a1, b2, b3  1.32
## 23           a2, b1, b2  1.32
## 24           a2, b1, b3  1.32
## 25           a2, b2, b3  1.32
## 26           b1, b2, b3  1.20
## 27       a1, a2, b1, b2  1.32
## 28       a1, a2, b1, b3  1.32
## 29       a1, a2, b2, b3  1.32
## 30       a1, b1, b2, b3  1.32
## 31       a2, b1, b2, b3  1.32
## 32 a1, a2, b1, b2, b3  1.32
```

This can, of course, be extended to more modules.

3.9 Synthetic viability

Synthetic viability and synthetic lethality (e.g., Ashworth et al., 2011; Hartman et al., 2001) are just special cases of epistasis (section 3.7) but we deal with them here separately.

3.9.1 A simple synthetic viability example

A simple and extreme example of synthetic viability is shown in the following table, where the joint mutant has fitness larger than the wild type, but each single mutant is lethal.

A	B	Fitness
wt	wt	1
wt	M	0
M	wt	0
M	M	$(1 + s)$

where "wt" denotes wild type and "M" denotes mutant.

We can specify this (setting $s = 0.2$) as (I play around with spaces, to show there is a certain flexibility with them):

```
s <- 0.2
sv <- allFitnessEffects(epistasis = c("-A : B" = -1,
                                      "A : -B" = -1,
                                      "A:B" = s))
```

Now, let's look at all the genotypes (we use "addwt" to also get the wt, which by decree has fitness of 1), and disregard order:

```
(asv <- evalAllGenotypes(sv, order = FALSE, addwt = TRUE))
##   Genotype Birth
## 1       WT   1.0
## 2        A   0.0
## 3        B   0.0
## 4     A, B   1.2
```

Asking the program to consider the order of mutations of course makes no difference:

```
evalAllGenotypes(sv, order = TRUE, addwt = TRUE)
##   Genotype Birth
## 1       WT   1.0
## 2        A   0.0
## 3        B   0.0
## 4    A > B   1.2
## 5    B > A   1.2
```

Another example of synthetic viability is shown in section 5.2.2.

Of course, if multiple simultaneous mutations are not possible in the simulations, it is not possible to go from the wildtype to the double mutant in this model where the single mutants are not viable.

3.9.2 Synthetic viability, non-zero fitness, and modules

This is a slightly more elaborate case, where there is one module and the single mutants have different fitness between themselves, which is non-zero. Without the modules, this is the same as in Misra et al. (2014), Figure 1b, which we go over in section 5.2.

A	B	Fitness
wt	wt	1
wt	M	$1 + s_b$
M	wt	$1 + s_a$
M	M	$1 + s_{ab}$

where $s_a, s_b < 0$ but $s_{ab} > 0$.

```
sa <- -0.1
sb <- -0.2
sab <- 0.25
sv2 <- allFitnessEffects(epistasis = c("-A : B" = sb,
                                       "A : -B" = sa,
                                       "A:B" = sab),
                         geneToModule = c(
                               "A" = "a1, a2",
                               "B" = "b"))
evalAllGenotypes(sv2, order = FALSE, addwt = TRUE)
##     Genotype Birth
## 1         WT  1.00
## 2         a1  0.90
## 3         a2  0.90
## 4          b  0.80
## 5     a1, a2  0.90
## 6      a1, b  1.25
## 7      a2, b  1.25
## 8 a1, a2, b  1.25
```

And if we look at order, of course it makes no difference:

```
evalAllGenotypes(sv2, order = TRUE, addwt = TRUE)
##          Genotype Birth
## 1              WT  1.00
## 2              a1  0.90
## 3              a2  0.90
## 4               b  0.80
```

121

```
## 5        a1 > a2  0.90
## 6         a1 > b   1.25
## 7        a2 > a1  0.90
## 8         a2 > b   1.25
## 9          b > a1  1.25
## 10         b > a2  1.25
## 11 a1 > a2 > b   1.25
## 12 a1 > b > a2   1.25
## 13 a2 > a1 > b   1.25
## 14 a2 > b > a1   1.25
## 15 b > a1 > a2   1.25
## 16 b > a2 > a1   1.25
```

3.10 Synthetic mortality or synthetic lethality

In contrast to section 3.9, here the joint mutant has decreased viability:

A	B	Fitness
wt	wt	1
wt	M	$1 + s_b$
M	wt	$1 + s_a$
M	M	$1 + s_{ab}$

where $s_a, s_b > 0$ but $s_{ab} < 0$.

```
sa <- 0.1
sb <- 0.2
sab <- -0.8
sm1 <- allFitnessEffects(epistasis = c("-A : B" = sb,
                                        "A : -B" = sa,
                                        "A:B" = sab))
evalAllGenotypes(sm1, order = FALSE, addwt = TRUE)
##   Genotype Birth
## 1       WT   1.0
## 2        A   1.1
## 3        B   1.2
## 4     A, B   0.2
```

And if we look at order, of course it makes no difference:

```
evalAllGenotypes(sm1, order = TRUE, addwt = TRUE)
##   Genotype Birth
## 1       WT   1.0
## 2        A   1.1
## 3        B   1.2
```

```
## 4     A > B    0.2
## 5     B > A    0.2
```

3.11 Possible issues with Bozic model

3.11.1 Synthetic viability using Bozic model

If we were to use the above specification with Bozic's models, we might not get what
we think we should get:

```
evalAllGenotypes(sv, order = FALSE, addwt = TRUE, model = "Bozic")
##    Genotype Death_rate
## 1        WT        1.0
## 2         A        2.0
## 3         B        2.0
## 4      A, B        0.8
```

What gives here? The simulation code would alert you of this (see section 3.11.2)
in this particular case because there are "-1", which might indicate that this is not
what you want. The problem is that you probably want the Death rate to be infinity
(the birth rate was 0, so no clone viability, when we used birth rates —section 3.2.2).

Let us say so explicitly:

```
s <- 0.2
svB <- allFitnessEffects(epistasis = c("-A : B" = -Inf,
                                       "A : -B" = -Inf,
                                       "A:B" = s))
evalAllGenotypes(svB, order = FALSE, addwt = TRUE, model = "Bozic")
##    Genotype Death_rate
## 1        WT        1.0
## 2         A        Inf
## 3         B        Inf
## 4      A, B        0.8
```

Likewise, values of s larger than one have no effect beyond setting $s = 1$ (a single
term of $(1 - 1)$ will drive the product to 0, and as we cannot allow negative death
rates negative values are set to 0):

```
s <- 1
svB1 <- allFitnessEffects(epistasis = c("-A : B" = -Inf,
                                        "A : -B" = -Inf,
                                        "A:B" = s))

evalAllGenotypes(svB1, order = FALSE, addwt = TRUE, model = "Bozic")
##    Genotype Death_rate
## 1        WT          1
## 2         A        Inf
```

```
## 3          B          Inf
## 4       A, B           0
```

```
s <- 3
svB3 <- allFitnessEffects(epistasis = c("-A : B" = -Inf,
                                         "A : -B" = -Inf,
                                         "A:B" = s))
```

```
evalAllGenotypes(svB3, order = FALSE, addwt = TRUE, model = "Bozic")
##   Genotype Death_rate
## 1       WT          1
## 2        A        Inf
## 3        B        Inf
## 4     A, B          0
```

Of course, death rates of 0.0 are likely to lead to trouble down the road, when we actually conduct simulations (see section 3.11.2).

3.11.2   Numerical issues with death rates of 0 in Bozic model

As we mentioned above (section 3.11.1) death rates of 0 can lead to trouble when using Bozic's model:

```
i1 <- allFitnessEffects(noIntGenes = c(1, 0.5))
evalAllGenotypes(i1, order = FALSE, addwt = TRUE,
                 model = "Bozic")
##   Genotype Death_rate
## 1       WT        1.0
## 2        1        0.0
## 3        2        0.5
## 4     1, 2        0.0
```

```
i1_b <- oncoSimulIndiv(i1, model = "Bozic", onlyCancer = TRUE)
## Warning in nr_oncoSimul.internal(rFE = fp, birth = birth, death =
## death, : You are using a Bozic model with the new restriction
## specification, and you have at least one s of 1. If that gene is
## mutated, this will lead to a death rate of 0 and the simulations
## will abort when you get a non finite value.
##
##  DEBUG2: Value of rnb = nan
##
##  DEBUG2: Value of m = 1
##
##  DEBUG2: Value of pe = 0
##
##  DEBUG2: Value of pm = 1
```

```
##
##  this is spP
##
##  popSize = 1
##  birth = 1
##  death = 0
##  W = 1
##  R = 1
##  mutation = 1e-10
##  timeLastUpdate = 533.709
##  absfitness = -inf
##  numMutablePos = 0
##
##  Unrecoverable exception: Algo 2: retval not finite. Aborting.
```

Of course, there is no problem in using the above with other models:

```
evalAllGenotypes(i1, order = FALSE, addwt = TRUE,
                 model = "Exp")
##   Genotype Birth
## 1       WT   1.0
## 2        1   2.0
## 3        2   1.5
## 4     1, 2   3.0
```

```
i1_e <- oncoSimulIndiv(i1, model = "Exp", onlyCancer = TRUE)
summary(i1_e)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         3    200196519    200110030             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0     803.7    1204
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA      5e+05      5e+05
##   OccurringDrivers
## 1
```

3.12 A longer example: Poset, epistasis, synthetic mortality and viability, order effects and genes without interactions, with some modules

We will now put together a complex example. We will use the poset from section 3.5.3 but will also add:

- Order effects that involve genes in the poset. In this case, if C happens before F, fitness decreases by $1 - 0.1$. If it happens the other way around, there is no effect on fitness beyond their individual contributions.
- Order effects that involve two new modules, "H" and "I" (with genes "h1,

125

h2" and "i1", respectively), so that if H happens before I fitness increases by $1 + 0.12$.

- Synthetic mortality between modules "I" (already present in the epistatic interaction) and "J" (with genes "j1" and "j2"): the joint presence of these modules leads to cell death (fitness of 0).
- Synthetic viability between modules "K" and "M" (with genes "k1", "k2" and "m1", respectively), so that their joint presence is viable but adds nothing to fitness (i.e., mutation of both has fitness 1), whereas each single mutant has a fitness of $1 - 0.5$.
- A set of 5 driver genes $(n1, \ldots, n5)$ with fitness that comes from an exponential distribution with rate of 10.

As we are specifying many different things, we will start by writing each set of effects separately:

```r
p4 <- data.frame(
    parent = c(rep("Root", 4), "A", "B", "D", "E", "C", "F"),
    child = c("A", "B", "D", "E", "C", "C", "F", "F", "G", "G"),
    s = c(0.01, 0.02, 0.03, 0.04, 0.1, 0.1, 0.2, 0.2, 0.3, 0.3),
    sh = c(rep(0, 4), c(-.9, -.9), c(-.95, -.95), c(-.99, -.99)),
    typeDep = c(rep("--", 4),
                "XMPN", "XMPN", "MN", "MN", "SM", "SM"))

oe <- c("C > F" = -0.1, "H > I" = 0.12)
sm <- c("I:J"  = -1)
sv <- c("-K:M" = -.5, "K:-M" = -.5)
epist <- c(sm, sv)

modules <- c("Root" = "Root", "A" = "a1",
             "B" = "b1, b2", "C" = "c1",
             "D" = "d1, d2", "E" = "e1",
             "F" = "f1, f2", "G" = "g1",
             "H" = "h1, h2", "I" = "i1",
             "J" = "j1, j2", "K" = "k1, k2", "M" = "m1")

set.seed(1) ## for reproducibility
noint <- rexp(5, 10)
names(noint) <- paste0("n", 1:5)

fea <- allFitnessEffects(rT = p4, epistasis = epist,
                         orderEffects = oe,
                         noIntGenes = noint,
                         geneToModule = modules)
```
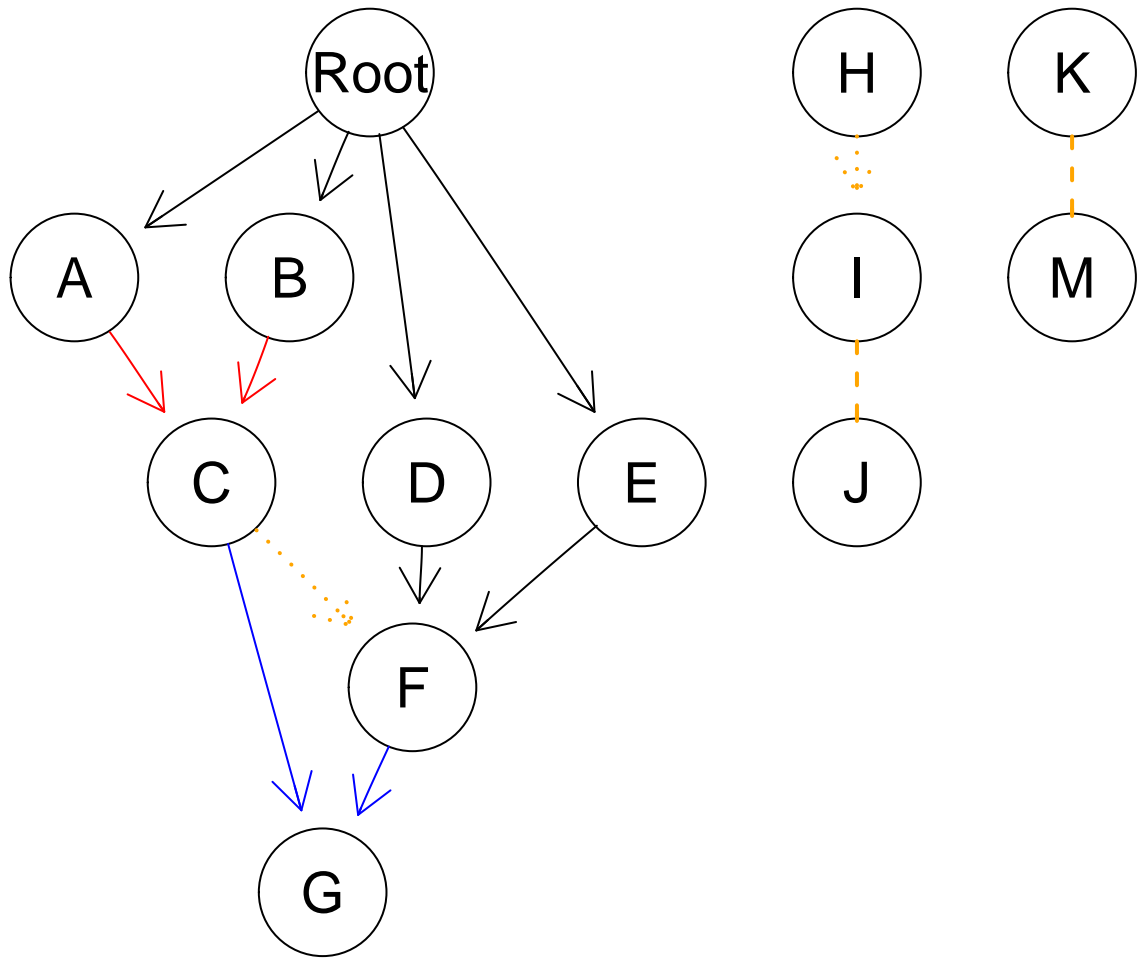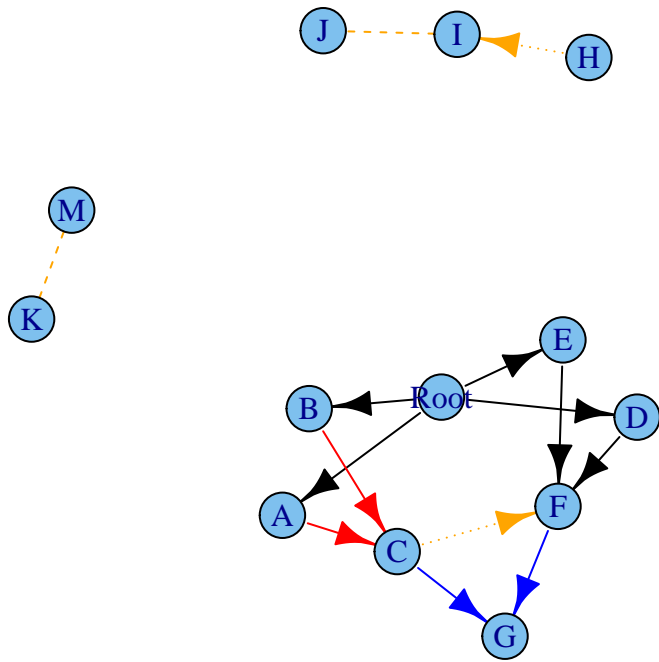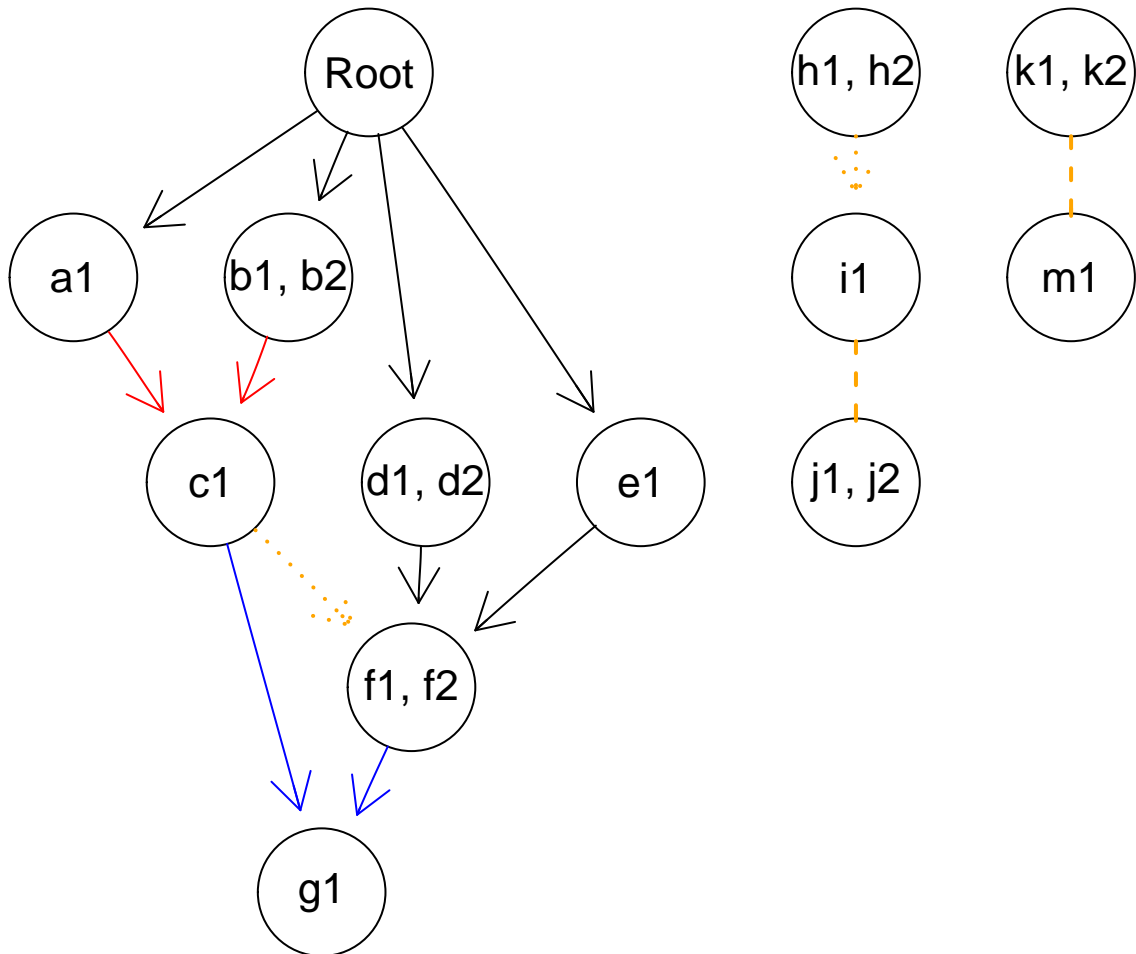
How does it look?
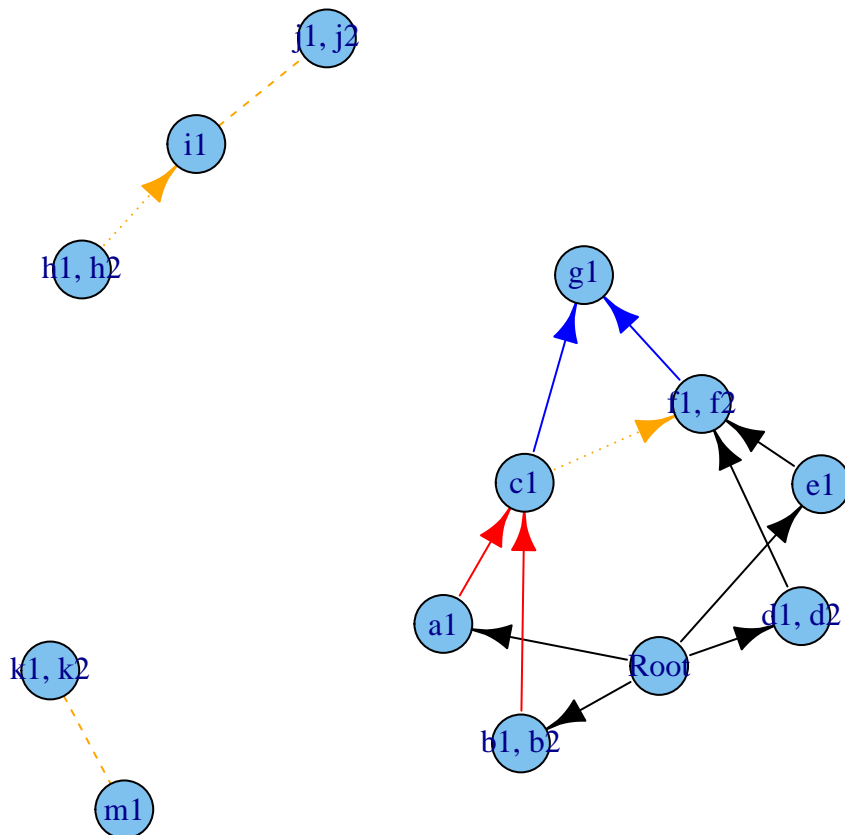
126

```
plot(fea)
```


or

```
plot(fea, "igraph")
```

We can, if we want, expand the modules using a "graphNEL" graph

```
plot(fea, expandModules = TRUE)
```

or an "igraph" one

```r
plot(fea, "igraph", expandModules = TRUE)
```


We will not evaluate the fitness of all genotypes, since the number of all ordered genotypes is $> 7 * 10^{22}$. We will look at some specific genotypes:

```r
evalGenotype("k1 > i1 > h2", fea) ## 0.5
## [1] 0.5
```

```r
evalGenotype("k1 > h1 > i1", fea) ## 0.5 * 1.12
## [1] 0.56
```

```r
evalGenotype("k2 > m1 > h1 > i1", fea) ## 1.12
## [1] 1.12
```

```r
evalGenotype("k2 > m1 > h1 > i1 > c1 > n3 > f2", fea)
## [1] 0.005113
```

```r
## 1.12 * 0.1 * (1 + noint[3]) * 0.05 * 0.9
```

Finally, let's generate some ordered genotypes randomly:

```r
randomGenotype <- function(fe, ns = NULL) {
    gn <- setdiff(c(fe$geneModule$Gene,
                    fe$long.geneNoInt$Gene), "Root")
```

129

```r
    if(is.null(ns)) ns <- sample(length(gn), 1)
    return(paste(sample(gn, ns), collapse = " > "))
}

set.seed(2) ## for reproducibility

evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  j2 > d2 > n4 > f1 > k2 > n1 > h2 > i1 > f2 > b1 > h1 > a1 > b2 > n3
##   Individual s terms are : 0.0755182 0.118164 0.0145707 0.0139795 0.01 0.02 -0.9
##   Fitness:  0
## [1] 0

## Genotype:  k2 > i1 > c1 > n1 > m1
##   Individual s terms are : 0.0755182 -0.9
##   Fitness:  0.107552
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  f2 > j1 > f1 > k1 > i1 > n4
##   Individual s terms are : 0.0139795 -0.95 -1 -0.5
##   Fitness:  0
## [1] 0

## Genotype:  n2 > h1 > h2
##   Individual s terms are : 0.118164
##   Fitness:  1.11816
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  d2 > n1 > f2 > f1 > i1 > n5 > b1 > e1 > k2 > b2 > c1 > j1 > a1 > k1
##   Individual s terms are : 0.0755182 0.0145707 0.0436069 0.01 0.02 -0.9 0.03 0.0
##   Fitness:  0
## [1] 0

## Genotype:  d2 > k2 > c1 > f2 > n4 > m1 > n3 > f1 > b1 > g1 > n5 > h1 > j2
##   Individual s terms are : 0.0145707 0.0139795 0.0436069 0.02 0.1 0.03 -0.95 0.
##   Fitness:  0.0725829
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  a1 > m1 > f1 > c1 > i1 > d1 > b1 > n4 > d2 > n1 > e1 > k2 > j2 > n2
##   Individual s terms are : 0.0755182 0.118164 0.0139795 0.01 0.02 -0.9 0.03 0.04
##   Fitness:  0
## [1] 0

## Genotype:  h2 > c1 > f1 > n2 > b2 > a1 > n1 > i1
##   Individual s terms are : 0.0755182 0.118164 0.01 0.02 -0.9 -0.95 -0.1 0.12
##   Fitness:  0.00624418
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  g1 > j2 > m1 > d2 > n1 > n4 > i1 > b2 > f1
##   Individual s terms are : 0.0755182 0.0139795 0.02 0.03 -0.95 0.3 -1 -0.5
##   Fitness:  0
## [1] 0
```

```
## Genotype:  h2 > j1 > m1 > d2 > i1 > b2 > k2 > d1 > b1 > n3 > n1 > g1 > h1 > c1
##  Individual s terms are : 0.0755182 0.0145707 0.0436069 0.01 0.02 -0.9 0.03 0.0
##  Fitness:  0
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  i1 > k2 > d1 > d2 > n4 > f2 > j2 > c1 > a1 > j1 > n1 > n3 > h1 > m1
##  Individual s terms are : 0.0755182 0.118164 0.0145707 0.0139795 0.0436069 0.0
##  Fitness:  0
## [1] 0

## Genotype:  n1 > m1 > n3 > i1 > j1 > n5 > k1
##  Individual s terms are : 0.0755182 0.0145707 0.0436069 -1
##  Fitness:  0
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  n2 > a1 > c1 > d2 > j1 > e1 > k1 > b2 > d1 > n3 > j2 > f2 > i1 > g1
##  Individual s terms are : 0.0755182 0.118164 0.0145707 0.0139795 0.0436069 0.0
##  Fitness:  0
## [1] 0

## Genotype:  d2 > n1 > g1 > f1 > f2 > c1 > b1 > d1 > k1 > a1 > b2 > i1 > n4 > h2
##  Individual s terms are : 0.0755182 0.118164 0.0139795 0.01 0.02 -0.9 0.03 -0.9
##  Fitness:  0.00420528
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  d2 > a1 > h2
##  Individual s terms are : 0.01 0.03
##  Fitness:  1.04
## [1] 1.04

## Genotype:  j1 > f1 > j2 > a1 > n4 > c1 > n3 > k1 > d1 > h1
##  Individual s terms are : 0.0145707 0.0139795 0.01 0.1 0.03 -0.95 -0.5
##  Fitness:  0.0294308
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  n2 > f2
##  Individual s terms are : 0.118164 -0.95
##  Fitness:  0.05591
## [1] 0.05591

## Genotype:  n5 > f2 > f1 > h2 > n4 > c1 > n3 > b1
##  Individual s terms are : 0.0145707 0.0139795 0.0436069 0.02 0.1 -0.95
##  Fitness:  0.0602298
evalGenotype(randomGenotype(fea), fea, echo = TRUE, verbose = TRUE)
## Genotype:  n5 > n1 > d1 > f1 > c1 > b1 > n2 > i1 > a1 > n3 > n4 > e1 > k2 > b2
##  Individual s terms are : 0.0755182 0.118164 0.0145707 0.0139795 0.0436069 0.0
##  Fitness:  0
## [1] 0

## Genotype:  h1 > d1 > f2
##  Individual s terms are : 0.03 -0.95
##  Fitness:  0.0515
```

3.13 Homozygosity, heterozygosity, oncogenes, tumor suppressors

We are using what is conceptually a single linear chromosome. However, you can use it to model scenarios where the numbers of copies affected matter, by properly duplicating the genes.

Suppose we have a tumor suppressor gene, G, with two copies, one from Mom and one from Dad. We can have a table like:

O_M	O_D	Fitness
wt	wt	1
wt	M	1
M	wt	1
M	M	$(1+s)$

where $s > 0$, meaning that you need two hits, one in each copy, to trigger the clonal expansion.

What about oncogenes? A simple model is that one single hit leads to clonal expansion and additional hits lead to no additional changes, as in this table for gene O, where again the M or D subscript denotes the copy from Mom or from Dad:

O_M	O_D	Fitness
wt	wt	1
wt	M	$(1+s)$
M	wt	$(1+s)$
M	M	$(1+s)$

If you have multiple copies you can proceed similarly. As you can see, these are nothing but special cases of synthetic mortality (3.10), synthetic viability (3.9) and epistasis (3.7).

3.14 Gene-specific mutation rates

You can specify gene-specific mutation rates. Instead of passing a scalar value for `mu`, you pass a named vector. (This does not work with the old v. 1 format, though; yet another reason to stop using that format). This is a simple example (many more are available in the tests, see file `./tests/testthat/test.per-gene-mutation-rates.R`).

```
muvar2 <- c("U" = 1e-6, "z" = 5e-5, "e" = 5e-4, "m" = 5e-3,
            "D" = 1e-4)
ni1 <- rep(0, 5)
names(ni1) <- names(muvar2) ## We use the same names, of course
fe1 <- allFitnessEffects(noIntGenes = ni1)
```

```
bb <- oncoSimulIndiv(fe1,
                     mu = muvar2, onlyCancer = FALSE,
                     initSize = 1e5,
                     finalTime = 25,
                     seed =NULL)
```

3.15   Mutator genes

You can specify mutator/antimutator genes (e.g. Gerrish et al., 2007; Tomlinson et al., 1996). These are genes that, when mutated, lead to an increase/decrease in the mutation rate all over the genome (similar to what happens with, say, mutations in mismatch-repair genes or microsatellite instability in cancer).

The specification is very similar to that for fitness effects, except we do not (at least for now) allow the use of DAGs nor of order effects (we have seen no reference in the literature to suggest any of these would be relevant). You can, however, specify epistasis and use modules. Note that the mutator genes must be a subset of the genes in the fitness effects; if you want to have mutator genes that have no direct fitness effects, give them a fitness effect of 0.

This first is a very simple example with simple fitness effects and modules for mutators. We will specify the fitness and mutator effects and evaluate the fitness and mutator effects:

```
fe2 <- allFitnessEffects(noIntGenes =
                         c(a1 = 0.1, a2 = 0.2,
                           b1 = 0.01, b2 = 0.3, b3 = 0.2,
                           c1 = 0.3, c2 = -0.2))

fm2 <- allMutatorEffects(epistasis = c("A" = 5,
                                       "B" = 10,
                                       "C" = 3),
                         geneToModule = c("A" = "a1, a2",
                                          "B" = "b1, b2, b3",
                                          "C" = "c1, c2"))

## Show the fitness effect of a specific genotype
evalGenotype("a1, c2", fe2, verbose = TRUE)
##
##  Individual s terms are : 0.1 -0.2
## [1] 0.88


## Show the mutator effect of a specific genotype
evalGenotypeMut("a1, c2", fm2, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##
```

```
##  Individual mutator product terms are : 5 3
## [1] 15
```

```
## Fitness and mutator of a specific genotype
evalGenotypeFitAndMut("a1, c2", fe2, fm2, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
##
##  Individual s terms are : 0.1 -0.2
##
##  Individual mutator product terms are : 5 3
## [1]   0.88 15.00
```

You can also use the `evalAll` functions. We do not show the output here to avoid cluttering the vignette:

```
## Show only all the fitness effects
evalAllGenotypes(fe2, order = FALSE)

## Show only all mutator effects
evalAllGenotypesMut(fm2)

## Show all fitness and mutator
evalAllGenotypesFitAndMut(fe2, fm2, order = FALSE)
```

Building upon the above, the next is an example where we have a bunch of no interaction genes that affect fitness, and a small set of genes that affect the mutation rate (but have no fitness effects).

```
set.seed(1) ## for reproducibility
## 17 genes, 7 with no direct fitness effects
ni <- c(rep(0, 7), runif(10, min = -0.01, max = 0.1))
names(ni) <- c("a1", "a2", "b1", "b2", "b3", "c1", "c2",
               paste0("g", 1:10))

fe3 <- allFitnessEffects(noIntGenes = ni)

fm3 <- allMutatorEffects(epistasis = c("A" = 5,
                                       "B" = 10,
                                       "C" = 3,
                                       "A:C" = 70),
                         geneToModule = c("A" = "a1, a2",
                                          "B" = "b1, b2, b3",
                                          "C" = "c1, c2"))
```

Let us check what the effects are of a few genotypes:

```
## These only affect mutation, not fitness
evalGenotypeFitAndMut("a1, a2", fe3, fm3, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
##
##   Individual s terms are : 0 0
##
##   Individual mutator product terms are : 5
## [1] 1 5
```

```
evalGenotypeFitAndMut("a1, b3", fe3, fm3, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
##
##   Individual s terms are : 0 0
##
##   Individual mutator product terms are : 5 10
## [1]   1 50
```

```
## These only affect fitness: the mutator multiplier is 1
evalGenotypeFitAndMut("g1", fe3, fm3, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
##
##   Individual s terms are : 0.019206
## [1] 1.019 1.000
```

```
evalGenotypeFitAndMut("g3, g9", fe3, fm3, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
##
##   Individual s terms are : 0.0530139 0.0592025
## [1] 1.115 1.000
```

```
## These affect both
evalGenotypeFitAndMut("g3, g9, a2, b3", fe3, fm3, verbose = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
##
##   Individual s terms are : 0 0 0.0530139 0.0592025
##
##   Individual mutator product terms are : 5 10
## [1]   1.115 50.000
```

Finally, we will do a simulation with those data

```
set.seed(1) ## so that it is easy to reproduce
mue1 <- oncoSimulIndiv(fe3, muEF = fm3,
                       mu = 1e-6,
                       initSize = 1e5,
                       model = "McFL",
                       detectionSize = 5e6,
                       finalTime = 500,
                       onlyCancer = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
```

```
## We do not show this in the vignette to avoid cluttering it
## with output
mue1
```

Of course, it is up to you to keep things reasonable: mutator effects are multiplicative, so if you specify, say, 20 genes (without modules), or 20 modules, each with a mutator effect of 50, the overall mutation rate can be increased by a factor of 50^{20} and that is unlikely to be what you really want (see also section 18.7).

You can play with the following case (an extension of the example above), where a clone with a mutator phenotype and some fitness enhancing mutations starts giving rise to many other clones, some with additional mutator effects, and thus leading to the number of clones blowing up (as some also accumulate additional fitness-enhancing mutations). Things start getting out of hand shortly after time 250. The code below takes a few minutes to run and is not executed here, but you can run it to get an idea of the increase in the number of clones and their relationships (the usage of `plotClonePhylog` is explained in section 8).

```
set.seed(1) ## for reproducibility
## 17 genes, 7 with no direct fitness effects
ni <- c(rep(0, 7), runif(10, min = -0.01, max = 0.1))
names(ni) <- c("a1", "a2", "b1", "b2", "b3", "c1", "c2",
               paste0("g", 1:10))

## Next is for nicer figure labeling.
## Consider as drivers genes with s >0
gp <- which(ni > 0)

fe3 <- allFitnessEffects(noIntGenes = ni,
                         drvNames = names(ni)[gp])


set.seed(12)
mue1 <- oncoSimulIndiv(fe3, muEF = fm3,
                       mu = 1e-6,
                       initSize = 1e5,
```

```
                            model = "McFL",
                            detectionSize = 5e6,
                            finalTime = 270,
                            keepPhylog = TRUE,
                            onlyCancer = FALSE)
mue1
## If you decrease N even further it gets even more cluttered
op <- par(ask = TRUE)
plotClonePhylog(mue1, N = 10, timeEvents = TRUE)
plot(mue1, plotDrivers = TRUE, addtot = TRUE,
     plotDiversity = TRUE)

## The stacked plot is slow; be patient
## Most clones have tiny population sizes, and their lines
## are piled on top of each other
plot(mue1, addtot = TRUE,
     plotDiversity = TRUE, type = "stacked")
par(op)
```

4 Plotting fitness landscapes

The `evalAllGenotypes` and related functions allow you to obtain tables of the genotype to fitness mappings. It might be more convenient to actually plot that, allowing us to quickly identify local minima and maxima and get an idea of how the fitness landscape looks.

In `plotFitnessLandscape` I have blatantly and shamelessly copied most of the looks of the plots of MAGELLAN (Brouillet et al., 2015) (see also http://wwwabi.snv.jus sieu.fr/public/Magellan/), a very nice web-based tool for fitness landscape plotting and analysis (MAGELLAN provides some other extra functionality and epistasis statistics not provided here).

As an example, let us show the example of Weissman et al. we saw in 5.4:

```r
d1 <- -0.05 ## single mutant fitness 0.95
d2 <- -0.08 ## double mutant fitness 0.92
d3 <- 0.2   ## triple mutant fitness 1.2
s2 <- ((1 + d2)/(1 + d1)^2) - 1
s3 <- ( (1 + d3)/((1 + d1)^3 * (1 + s2)^3) ) - 1

wb <- allFitnessEffects(
    epistasis = c(
        "A" = d1,
        "B" = d1,
        "C" = d1,
        "A:B" = s2,
        "A:C" = s2,
        "B:C" = s2,
        "A:B:C" = s3))
```

```r
plotFitnessLandscape(wb, use_ggrepel = TRUE)
```
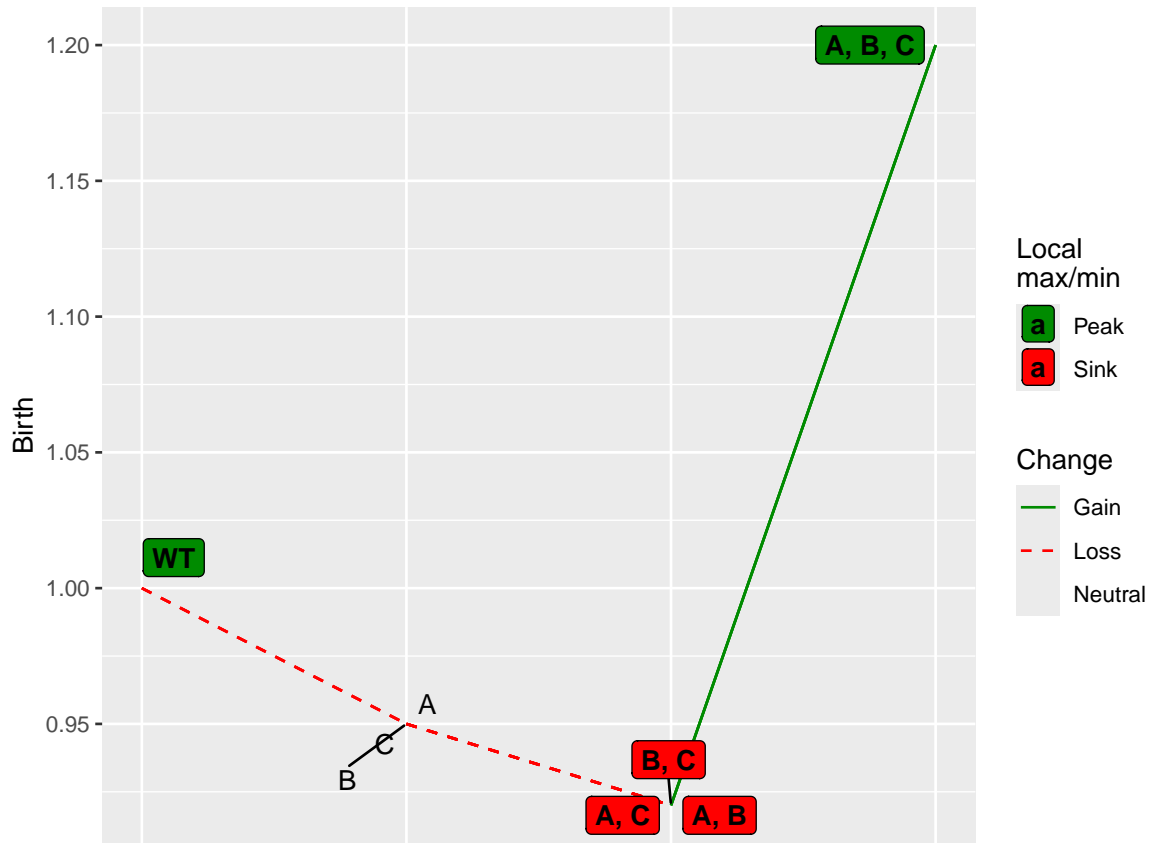
We have set `use_ggrepel = TRUE` to avoid overlap of labels.

For some types of objects, directly invoking `plot` will give you the fitness landscape plot:

```
(ewb <- evalAllGenotypes(wb, order = FALSE))
##   Genotype Birth
## 1        A  0.95
## 2        B  0.95
## 3        C  0.95
## 4     A, B  0.92
## 5     A, C  0.92
## 6     B, C  0.92
## 7  A, B, C  1.20
```

```
plot(ewb, use_ggrepel = TRUE)
```
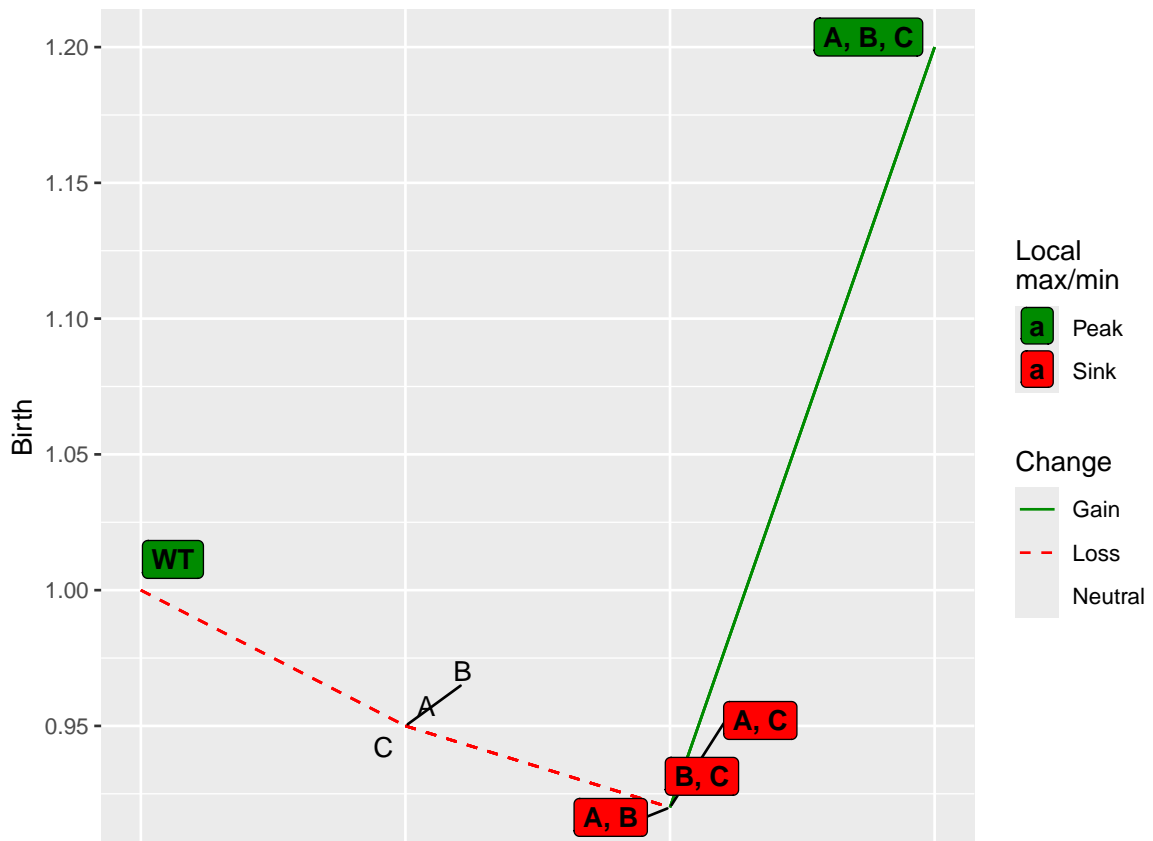
This is example (section 5.5) will give a very busy plot:

```r
par(cex = 0.7)
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 = 0.1,
            sh = -0.9,
            typeDep = "MN"))
plot(evalAllGenotypes(pancr, order = FALSE), use_ggrepel = TRUE)
## Warning: ggrepel: 103 unlabeled data points (too many overlaps).
## Consider increasing max.overlaps
## Warning: ggrepel: 7 unlabeled data points (too many overlaps).
## Consider increasing max.overlaps
```
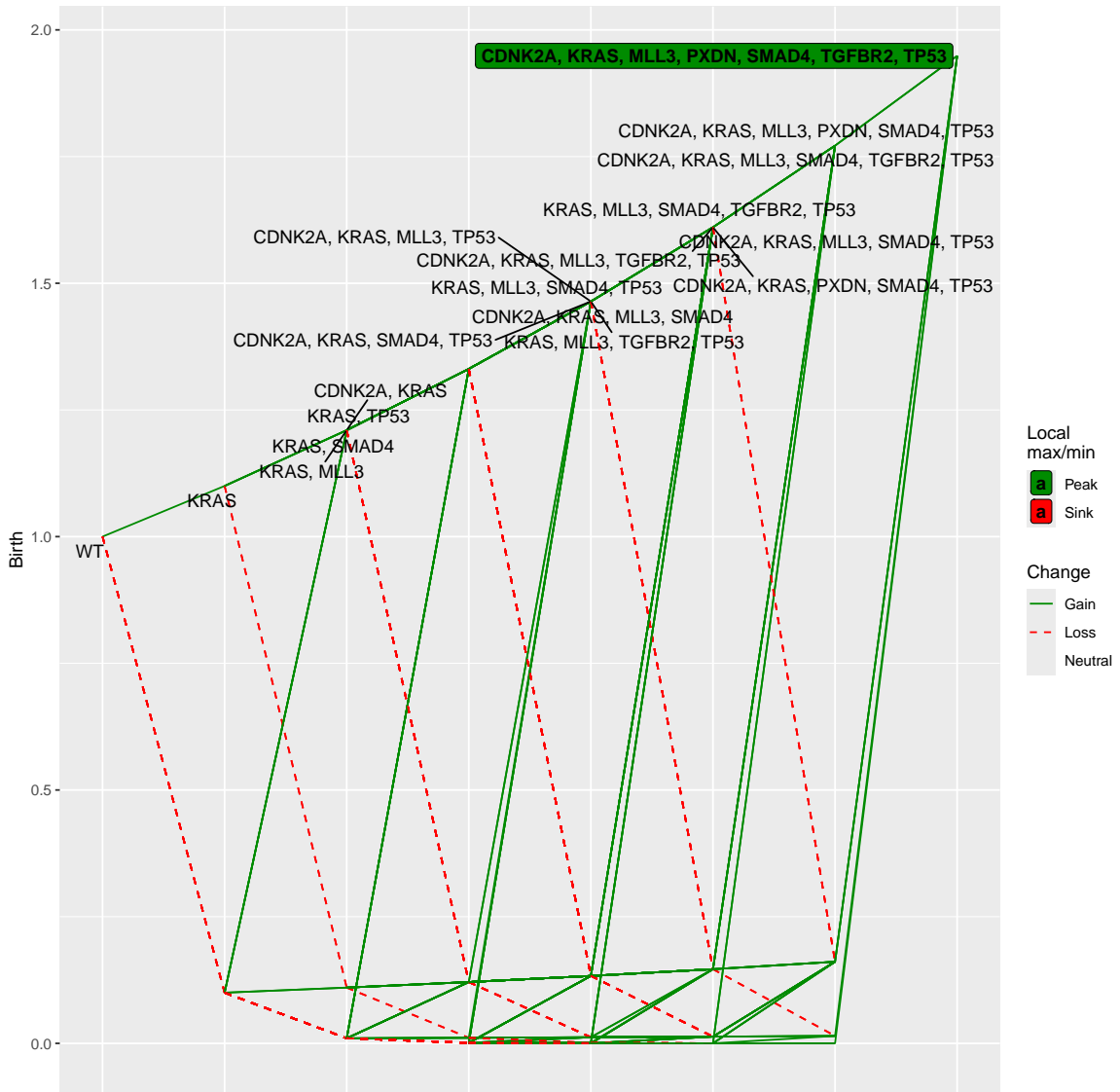
5 Specifying fitness effects: some examples from the literature

5.1 Bauer et al., 2014

In the model of Bauer and collaborators (Bauer et al., 2014, p. 54) we have "For cells without the primary driver mutation, each secondary driver mutation leads to a change in the cell's fitness by s_P. For cells with the primary driver mutation, the fitness advantage obtained with each secondary driver mutation is s_{DP}."

The proliferation probability is given as:

- $\frac{1}{2}(1 + s_p)^k$ when there are k secondary drivers mutated and no primary diver;
- $\frac{1}{2}\frac{1+S_D^+}{1+S_D^-}(1 + S_{DP})^k$ when the primary driver is mutated;

apoptosis is one minus the proliferation rate.

5.1.1 Using a DAG

We cannot find a simple mapping from their expressions to our fitness parameterization, but we can get fairly close by using a DAG; in this one, note the unusual feature of having one of the "s" terms (that for the driver dependency on root) be negative. Using the parameters given in the legend of their Figure 3 for $s_p, S_D^+, S_D^-, S_{DP}$ and obtaining that negative value for the dependency of the driver on root we can do:

```
K <- 4
sp <- 1e-5
sdp <- 0.015
sdplus <- 0.05
sdminus <- 0.1
cnt <- (1 + sdplus)/(1 + sdminus)
prod_cnt <- cnt - 1
bauer <- data.frame(parent = c("Root", rep("D", K)),
                child = c("D", paste0("s", 1:K)),
                s = c(prod_cnt, rep(sdp, K)),
                sh = c(0, rep(sp, K)),
                typeDep = "MN")
fbauer <- allFitnessEffects(bauer)
(b1 <- evalAllGenotypes(fbauer, order = FALSE, addwt = TRUE))
##            Genotype  Birth
## 1                WT 1.0000
## 2                 D 0.9545
## 3                s1 1.0000
## 4                s2 1.0000
## 5                s3 1.0000
## 6                s4 1.0000
## 7             D, s1 0.9689
## 8             D, s2 0.9689
```

```
## 9                D, s3 0.9689
## 10               D, s4 0.9689
## 11              s1, s2 1.0000
## 12              s1, s3 1.0000
## 13              s1, s4 1.0000
## 14              s2, s3 1.0000
## 15              s2, s4 1.0000
## 16              s3, s4 1.0000
## 17          D, s1, s2 0.9834
## 18          D, s1, s3 0.9834
## 19          D, s1, s4 0.9834
## 20          D, s2, s3 0.9834
## 21          D, s2, s4 0.9834
## 22          D, s3, s4 0.9834
## 23          s1, s2, s3 1.0000
## 24          s1, s2, s4 1.0000
## 25          s1, s3, s4 1.0000
## 26          s2, s3, s4 1.0000
## 27      D, s1, s2, s3 0.9981
## 28      D, s1, s2, s4 0.9981
## 29      D, s1, s3, s4 0.9981
## 30      D, s2, s3, s4 0.9981
## 31     s1, s2, s3, s4 1.0000
## 32 D, s1, s2, s3, s4 1.0131
```
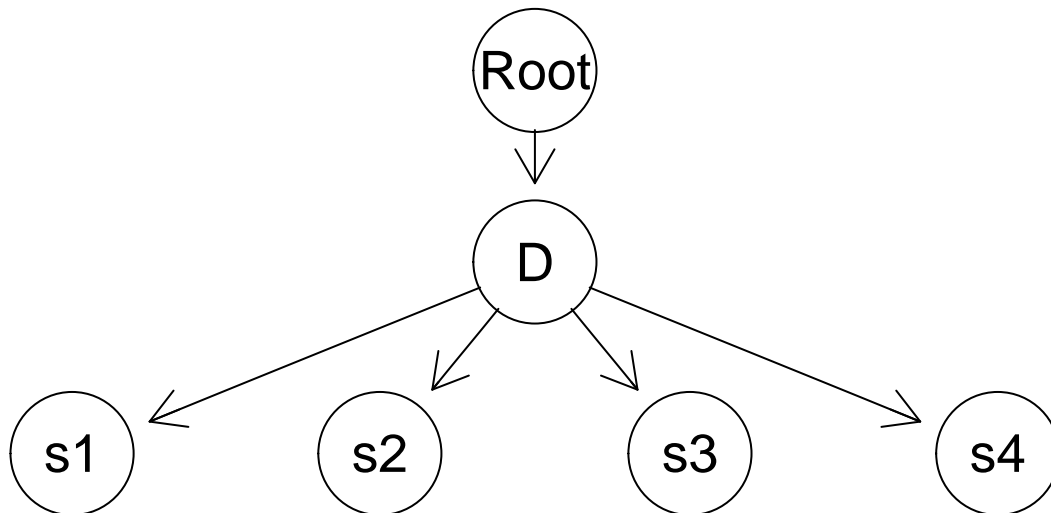
(We use "D" for "driver" or "primary driver", as is it is called in the original paper, and "s" for secondary drivers, somewhat similar to passengers).

Note that what we specify as "typeDep" is irrelevant (MN, SMN, or XMPN make no difference).

This is the DAG:

```
plot(fbauer)
```

Root

D

s1   s2   s3   s4

And if you compare the tabular output of `evalAllGenotypes` you can see that the values of fitness reproduces the fitness landscape that they show in their Figure 1. We can also use our plot for fitness landscapes:

```
plot(b1, use_ggrepel = TRUE)
## Warning: ggrepel: 20 unlabeled data points (too many overlaps).
## Consider increasing max.overlaps
```
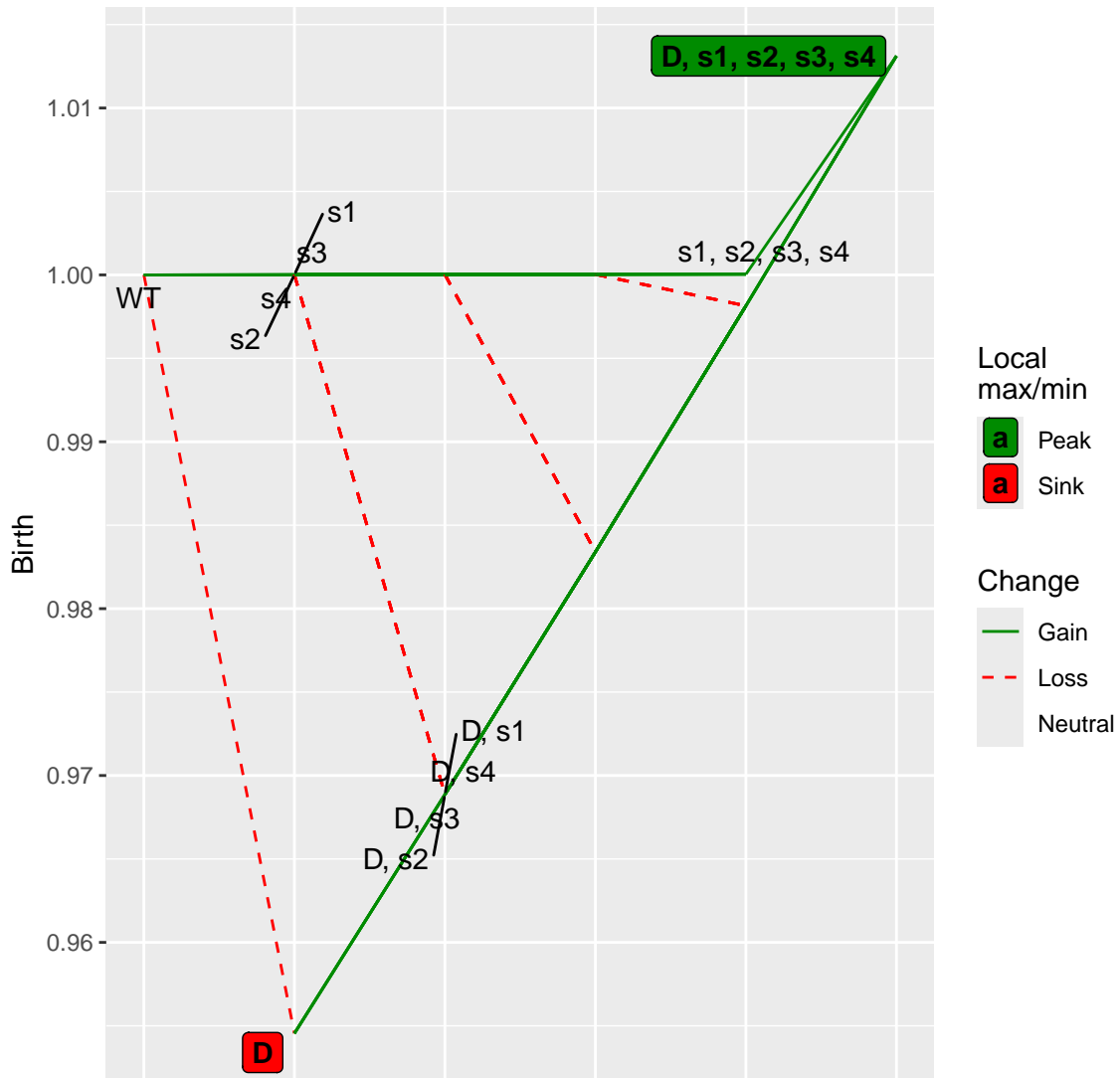
5.1.2 Specifying fitness of genotypes directly

An alternative approach to specify the fitness, if the number of genotypes is reasonably small, is to directly evaluate fitness as given by their expressions. Then, use the `genotFitness` argument to `allFitnessEffects`.

We will create all possible genotypes; then we will write a function that gives the fitness of each genotype according to their expression; finally, we will call this function on the data frame of genotypes, and pass this data frame to `allFitnessEffects`.

```
m1 <- expand.grid(D = c(1, 0), s1 = c(1, 0), s2 = c(1, 0),
                  s3 = c(1, 0), s4 = c(1, 0))

fitness_bauer <- function(D, s1, s2, s3, s4,
                          sp = 1e-5, sdp = 0.015, sdplus = 0.05,
                          sdminus = 0.1) {
    if(!D) {
        b <- 0.5 * ( (1 + sp)^(sum(c(s1, s2, s3, s4)))))
```

```
    } else {
        b <- 0.5 *
            (((1 + sdplus)/(1 + sdminus)  *
             (1 + sdp)^(sum(c(s1, s2, s3, s4)))))
    }
    fitness <- b - (1 - b)
    our_fitness <- 1 + fitness ## prevent negative fitness and
    ## make wt fitness = 1
    return(our_fitness)
}


m1$Fitness <-
    apply(m1, 1, function(x) do.call(fitness_bauer, as.list(x)))

bauer2 <- allFitnessEffects(genotFitness = m1)
```

Now, show the fitness of all genotypes:

```
evalAllGenotypes(bauer2, order = FALSE, addwt = TRUE)
##              Genotype  Birth
## 1                 WT 1.0000
## 2                  D 0.9545
## 3                 s1 1.0000
## 4                 s2 1.0000
## 5                 s3 1.0000
## 6                 s4 1.0000
## 7             D, s1 0.9689
## 8             D, s2 0.9689
## 9             D, s3 0.9689
## 10            D, s4 0.9689
## 11           s1, s2 1.0000
## 12           s1, s3 1.0000
## 13           s1, s4 1.0000
## 14           s2, s3 1.0000
## 15           s2, s4 1.0000
## 16           s3, s4 1.0000
## 17        D, s1, s2 0.9834
## 18        D, s1, s3 0.9834
## 19        D, s1, s4 0.9834
## 20        D, s2, s3 0.9834
## 21        D, s2, s4 0.9834
## 22        D, s3, s4 0.9834
## 23       s1, s2, s3 1.0000
## 24       s1, s2, s4 1.0000
## 25       s1, s3, s4 1.0000
## 26       s2, s3, s4 1.0000
```
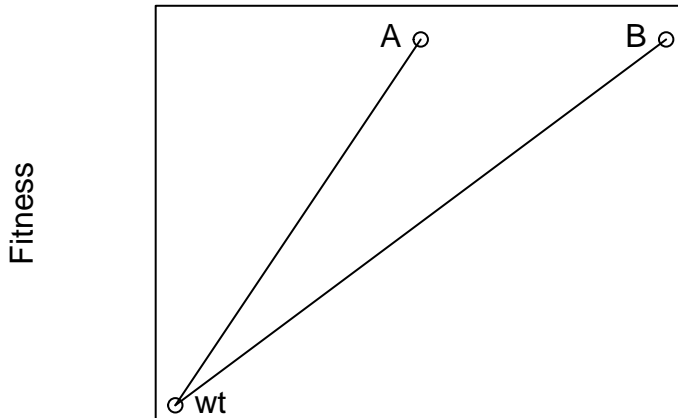
```
## 27     D, s1, s2, s3 0.9981
## 28     D, s1, s2, s4 0.9981
## 29     D, s1, s3, s4 0.9981
## 30     D, s2, s3, s4 0.9981
## 31    s1, s2, s3, s4 1.0000
## 32 D, s1, s2, s3, s4 1.0131
```

Can we use modules in this example, if we use the "lego system"? Sure, as in any other case.

5.2 Misra et al., 2014

Figure 1 of Misra et al. (2014) presents three scenarios which are different types of epistasis.

5.2.1 Example 1.a

In that figure it is evident that the fitness effect of "A" and "B" are the same. There are two different models depending on whether "AB" is just the product of both, or there is epistasis. In the first case probably the simplest is:

```
s <- 0.1 ## or whatever number
m1a1 <- allFitnessEffects(data.frame(parent = c("Root", "Root"),
                                      child = c("A", "B"),
                                      s = s,
                                      sh = 0,
                                      typeDep = "MN"))
evalAllGenotypes(m1a1, order = FALSE, addwt = TRUE)
##   Genotype Birth
## 1       WT  1.00
## 2        A  1.10
## 3        B  1.10
## 4     A, B  1.21
```

If the double mutant shows epistasis, as we saw before (section 3.7.1) we have a range of options. For example:

```
s <- 0.1
sab <- 0.3
m1a2 <- allFitnessEffects(epistasis = c("A:-B" = s,
                                        "-A:B" = s,
                                        "A:B" = sab))
evalAllGenotypes(m1a2, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1       WT   1.0
## 2        A   1.1
## 3        B   1.1
## 4     A, B   1.3
```

But we could also modify the graph dependency structure, and we have to change the value of the coefficient, since that is what multiplies each of the terms for "A" and "B": $(1 + s_{AB}) = (1 + s)^2(1 + s_{AB3})$

```
sab3 <- ((1 + sab)/((1 + s)^2)) - 1
m1a3 <- allFitnessEffects(data.frame(parent = c("Root", "Root"),
                                     child = c("A", "B"),
                                     s = s,
                                     sh = 0,
                                     typeDep = "MN"),
                          epistasis = c("A:B" = sab3))
evalAllGenotypes(m1a3, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1       WT   1.0
## 2        A   1.1
## 3        B   1.1
## 4     A, B   1.3
```
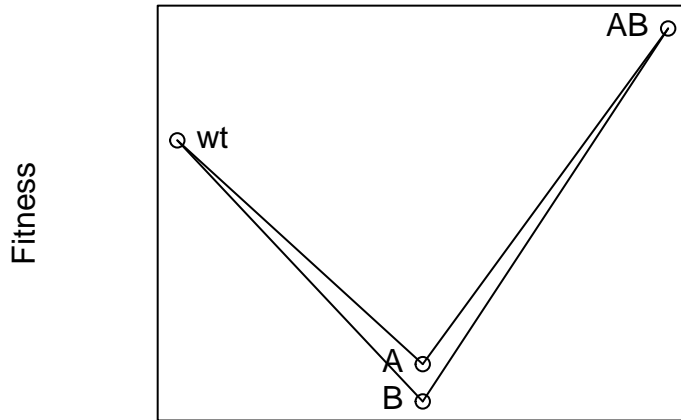
And, obviously

```
all.equal(evalAllGenotypes(m1a2, order = FALSE, addwt = TRUE),
          evalAllGenotypes(m1a3, order = FALSE, addwt = TRUE))
## [1] TRUE
```

5.2.2   Example 1.b

This is a specific case of synthetic viability (see also section 3.9):

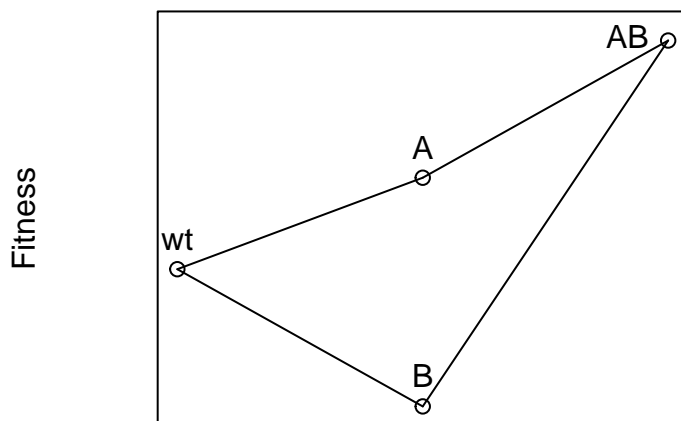Here, $S_A, S_B < 0$, $S_B < 0$, $S_{AB} > 0$ and $(1 + S_{AB})(1 + S_A)(1 + S_B) > 1$.

As before, we can specify this in several different ways. The simplest is to specify all genotypes:

```r
sa <- -0.6
sb <- -0.7
sab <- 0.3
m1b1 <- allFitnessEffects(epistasis = c("A:-B" = sa,
                                        "-A:B" = sb,
                                        "A:B" = sab))
evalAllGenotypes(m1b1, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT   1.0
## 2         A   0.4
## 3         B   0.3
## 4      A, B   1.3
```

We could also use a tree and modify the "sab" for the epistasis, as before (5.2.1).

5.2.3   Example 1.c

The final case, in figure 1.c of Misra et al., is just epistasis, where a mutation in one of the genes is deleterious (possibly only mildly), in the other is beneficial, and the double mutation has fitness larger than any of the other two.

Here we have that $s_A > 0$, $s_B < 0$, $(1 + s_{AB})(1 + s_A)(1 + s_B) > (1 + s_{AB})$ so $s_{AB} > \frac{-s_B}{1+s_B}$

As before, we can specify this in several different ways. The simplest is to specify all genotypes:

```
sa <- 0.2
sb <- -0.3
sab <- 0.5
m1c1 <- allFitnessEffects(epistasis = c("A:-B" = sa,
                                        "-A:B" = sb,
                                        "A:B" = sab))
evalAllGenotypes(m1c1, order = FALSE, addwt = TRUE)
##   Genotype Birth
## 1       WT   1.0
## 2        A   1.2
## 3        B   0.7
## 4     A, B   1.5
```

We could also use a tree and modify the "sab" for the epistasis, as before (5.2.1).

5.3 Ochs and Desai, 2015

In Ochs & Desai (2015) the authors present a model shown graphically as (the actual numerical values are arbitrarily set by me):
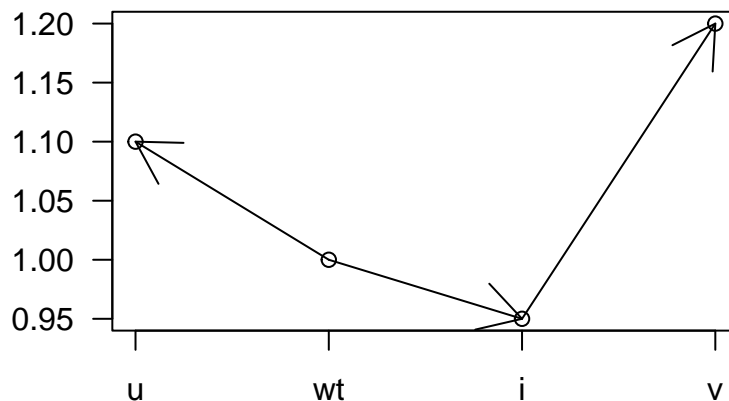
In their model, $s_u > 0$, $s_v > s_u$, $s_i < 0$, we can only arrive at v from i, and the mutants "ui" and "uv" can never appear as their fitness is 0, or $-\infty$, so $s_{ui} = s_{uv} = -1$ (or $-\infty$).
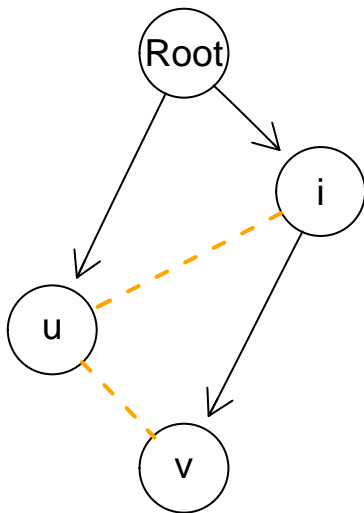
We can specify this combining a graph and epistasis specifications:

```
su <- 0.1
si <- -0.05
fvi <- 1.2 ## the fitness of the vi mutant
sv <- (fvi/(1 + si)) - 1
sui <- suv <- -1
od <- allFitnessEffects(
```

```
    data.frame(parent = c("Root", "Root", "i"),
               child = c("u", "i", "v"),
               s = c(su, si, sv),
               sh = -1,
               typeDep = "MN"),
    epistasis = c(
        "u:i" = sui,
        "u:v" = suv))
```

A figure showing that model is

```
plot(od)
```


And the fitness of all genotype is

```
evalAllGenotypes(od, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT  1.00
## 2         i  0.95
## 3         u  1.10
## 4         v  0.00
## 5      i, u  0.00
## 6      i, v  1.20
## 7      u, v  0.00
## 8   i, u, v  0.00
```

We could alternatively have specified fitness either directly specifying the fitness of each genotype or specifying epistatic effects. Let us use the second approach:

%% this was wrong %% u <- 0.2; i <- -0.02; vi <- 0.6; ui <- uv <- -Inf

```
u <- 0.1; i <- -0.05; vi <- (1.2/0.95) - 1; ui <- uv <- -Inf
od2 <- allFitnessEffects(
    epistasis = c("u" = u,   "u:i" = ui,
                  "u:v" = uv, "i" = i,
```

```
                       "v:-i" = -Inf, "v:i" = vi))
evalAllGenotypes(od2, addwt = TRUE)
##   Genotype Birth
## 1       WT  1.00
## 2        i  0.95
## 3        u  1.10
## 4        v  0.00
## 5     i, u  0.00
## 6     i, v  1.20
## 7     u, v  0.00
## 8  i, u, v  0.00
```
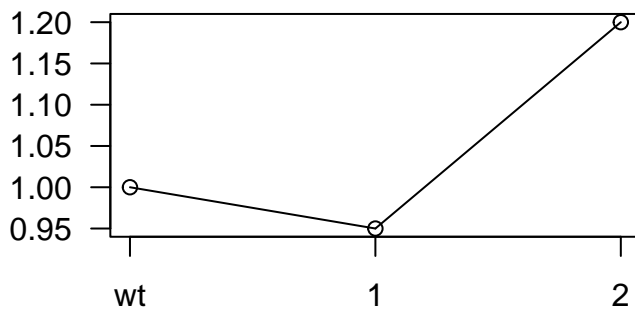
We will return to this model when we explain the usage of `fixation` for stopping the simulations (see 6.3.3 and 6.3.4).

5.4   Weissman et al., 2009

In their figure 1a, Weissman et al. (2009) present this model (actual numeric values are set arbitrarily)
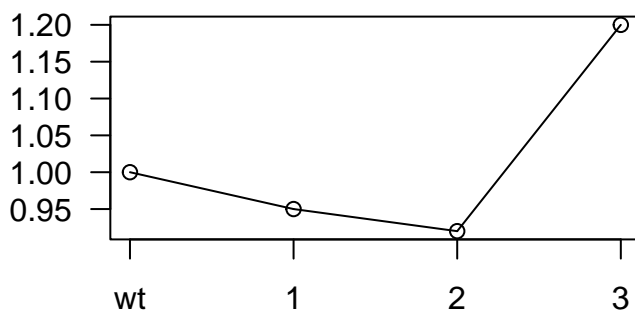
5.4.1   Figure 1.a

where the "1" and "2" in the figure refer to the total number of mutations in two different loci. This is, therefore, very similar to the example in section 5.2.2. Here we have, in their notation, $\delta_1 < 0$, fitness of single "A" or single "B" $= 1 + \delta_1$, $S_{AB} > 0$, $(1 + S_{AB})(1 + \delta_1)^2 > 1$.

5.4.2   Figure 1.b

In their figure 1b they show

Where, as before, 1, 2, 3, denote the total number of mutations over three different loci and $\delta_1 < 0$, $\delta_2 < 0$, fitness of single mutant is $(1 + \delta_1)$, of double mutant is $(1 + \delta_2)$ so that $(1 + \delta_2) = (1 + \delta_1)^2(1 + s_2)$ and of triple mutant is $(1 + \delta_3)$, so that $(1 + \delta_3) = (1 + \delta_1)^3(1 + s_2)^3(1 + s_3)$.

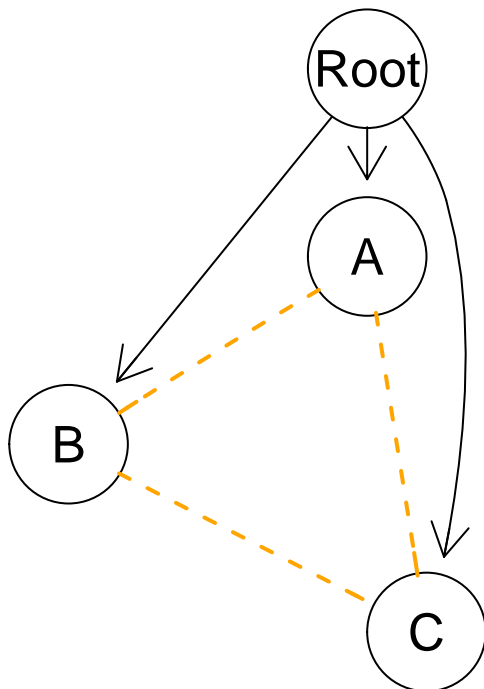We can specify this combining a graph with epistasis:

```
d1 <- -0.05 ## single mutant fitness 0.95
d2 <- -0.08 ## double mutant fitness 0.92
d3 <- 0.2   ## triple mutant fitness 1.2

s2 <- ((1 + d2)/(1 + d1)^2) - 1
s3 <- ( (1 + d3)/((1 + d1)^3 * (1 + s2)^3) ) - 1

w <- allFitnessEffects(
    data.frame(parent = c("Root", "Root", "Root"),
               child = c("A", "B", "C"),
               s = d1,
               sh = -1,
               typeDep = "MN"),
    epistasis = c(
        "A:B" = s2,
        "A:C" = s2,
        "B:C" = s2,
        "A:B:C" = s3))
```

The model can be shown graphically as:

```
plot(w)
```

And fitness of all genotypes is:

```
evalAllGenotypes(w, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT  1.00
## 2         A  0.95
## 3         B  0.95
## 4         C  0.95
## 5      A, B  0.92
## 6      A, C  0.92
## 7      B, C  0.92
## 8   A, B, C  1.20
```

Alternatively, we can directly specify what each genotype adds to the fitness, given the included genotype. This is basically replacing the graph by giving each of "A", "B", and "C" directly:

```
wb <- allFitnessEffects(
    epistasis = c(
        "A" = d1,
        "B" = d1,
        "C" = d1,
        "A:B" = s2,
        "A:C" = s2,
        "B:C" = s2,
        "A:B:C" = s3))

evalAllGenotypes(wb, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT  1.00
## 2         A  0.95
## 3         B  0.95
## 4         C  0.95
## 5      A, B  0.92
## 6      A, C  0.92
## 7      B, C  0.92
## 8   A, B, C  1.20
```

The plot, of course, is not very revealing and we cannot show that there is a three-way interaction (only all three two-way interactions):

```
plot(wb)
```
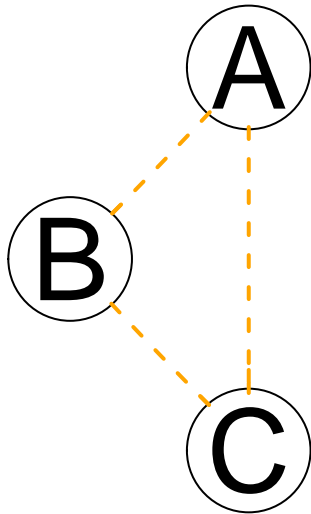
As we have seen several times already (sections 3.7.1, 3.7.2, 3.7.3) we can also give the genotypes directly and, consequently, the fitness of each genotype (not the added contribution):

```
wc <- allFitnessEffects(
    epistasis = c(
        "A:-B:-C" = d1,
        "B:-C:-A" = d1,
        "C:-A:-B" = d1,
        "A:B:-C" = d2,
        "A:C:-B" = d2,
        "B:C:-A" = d2,
        "A:B:C" = d3))
evalAllGenotypes(wc, order = FALSE, addwt = TRUE)
##    Genotype Birth
## 1        WT  1.00
## 2         A  0.95
## 3         B  0.95
## 4         C  0.95
## 5      A, B  0.92
## 6      A, C  0.92
## 7      B, C  0.92
## 8   A, B, C  1.20
```

5.5   Gerstung et al., 2011, pancreatic cancer poset

We can specify the pancreatic cancer poset in Gerstung, Eriksson, et al. (2011) (their figure 2B, left). We use directly the names of the genes, since that is immediately supported by the new version.

```
pancr <- allFitnessEffects(
    data.frame(parent = c("Root", rep("KRAS", 4),
                    "SMAD4", "CDNK2A",
```
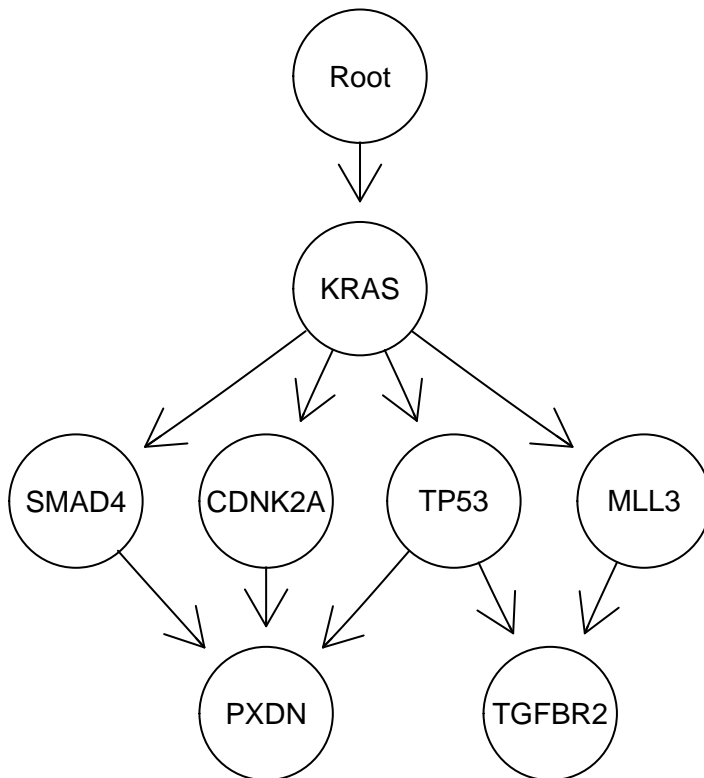
```
                "TP53", "TP53", "MLL3"),
            child = c("KRAS","SMAD4", "CDNK2A",
                "TP53", "MLL3",
                rep("PXDN", 3), rep("TGFBR2", 2)),
            s = 0.1,
            sh = -0.9,
            typeDep = "MN"))

plot(pancr)
```


Of course the "s" and "sh" are set arbitrarily here.

5.6   Raphael and Vandin's 2014 modules

In Raphael & Vandin (2015), the authors show several progression models in terms
of modules. We can code the extended poset for the colorectal cancer model in their
Figure 4.a is (s and sh are arbitrary):

```
rv1 <- allFitnessEffects(data.frame(parent = c("Root", "A", "KRAS"),
                            child = c("A", "KRAS", "FBXW7"),
                            s = 0.1,
                            sh = -0.01,
                            typeDep = "MN"),
                    geneToModule = c("Root" = "Root",
                        "A" = "EVC2, PIK3CA, TP53",
                        "KRAS" = "KRAS",
```
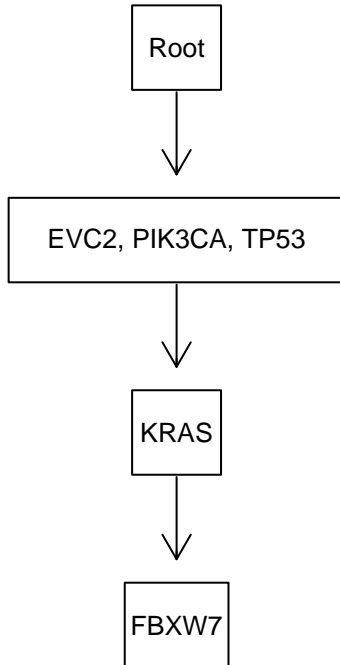
```
                          "FBXW7" = "FBXW7"))

plot(rv1, expandModules = TRUE, autofit = TRUE)
```


We have used the (experimental) `autofit` option to fit the labels to the edges. Note how we can use the same name for genes and modules, but we need to specify all the modules.

Their Figure 5b is

```
rv2 <- allFitnessEffects(
    data.frame(parent = c("Root", "1", "2", "3", "4"),
               child = c("1", "2", "3", "4", "ELF3"),
               s = 0.1,
               sh = -0.01,
               typeDep = "MN"),
    geneToModule = c("Root" = "Root",
                     "1" = "APC, FBXW7",
                     "2" = "ATM, FAM123B, PIK3CA, TP53",
                     "3" = "BRAF, KRAS, NRAS",
                     "4" = "SMAD2, SMAD4, SOX9",
                     "ELF3" = "ELF3"))

plot(rv2, expandModules = TRUE,   autofit = TRUE)
```

```
┌──────┐
│ Root │
└──────┘
    │
    ▼
┌──────────────┐
│ APC, FBXW7   │
└──────────────┘
    │
    ▼
┌────────────────────────────┐
│ ATM, FAM123B, PIK3CA, TP53 │
└────────────────────────────┘
    │
    ▼
┌────────────────────┐
│ BRAF, KRAS, NRAS   │
└────────────────────┘
    │
    ▼
┌────────────────────────┐
│ SMAD2, SMAD4, SOX9     │
└────────────────────────┘
    │
    ▼
┌──────┐
│ ELF3 │
└──────┘
```
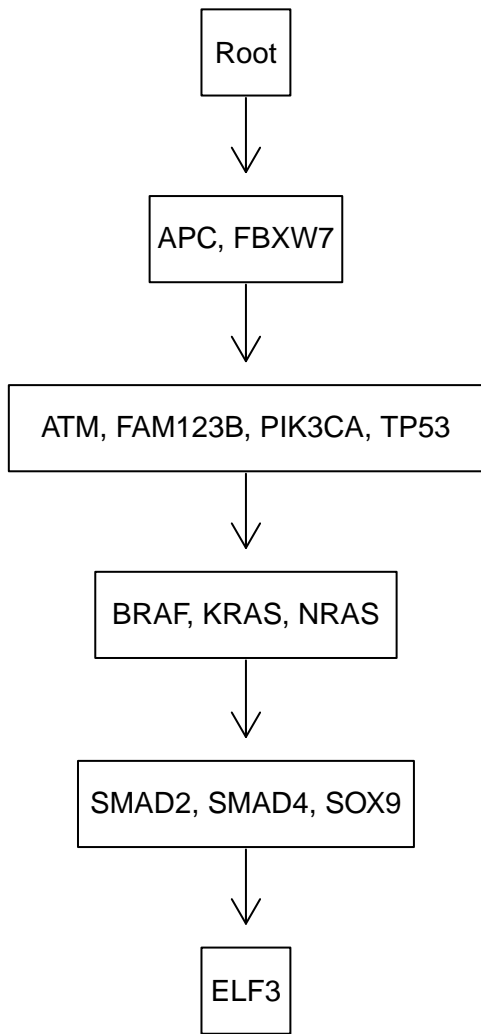
6 Running and plotting the simulations: starting, ending, and examples

6.1 Starting and ending

After you have decided the specifics of the fitness effects and the model, you need to decide:

- Where will you start your simulation from. This involves deciding the initial population size (argument `initSize`) and, possibly, the genotype of the initial population; the later is covered in section 6.2.

- When will you stop it: how long to run it, and whether or not to require simulations to reach cancer (under some definition of what it means to reach cancer). This is covered in 6.3.

## 6.2 Can I start the simulation from a specific mutant?

You bet. In version 2 you can specify the genotype for the initial mutant with the same flexibility as in `evalGenotype`. Here we show a couple of examples (we use the representation of the parent-child relationships —discussed in section 8— of the clones so that you can see which clones appear, and from which, and check that we are not making mistakes).

```r
o3init <- allFitnessEffects(orderEffects = c(
                    "M > D > F" = 0.99,
                    "D > M > F" = 0.2,
                    "D > M"     = 0.1,
                    "M > D"     = 0.9),
                noIntGenes = c("u" = 0.01,
                               "v" = 0.01,
                               "w" = 0.001,
                               "x" = 0.0001,
                               "y" = -0.0001,
                               "z" = -0.001),
                geneToModule =
                    c("M" = "m",
                      "F" = "f",
                      "D" = "d") )

oneI <- oncoSimulIndiv(o3init, model = "McFL",
                    mu = 5e-5, finalTime = 200,
                    detectionDrivers = 3,
                    onlyCancer = FALSE,
                    initSize = 1000,
                    keepPhylog = TRUE,
                    initMutant = c("m > u > d")
```

```
                            )
plotClonePhylog(oneI, N = 0)
```
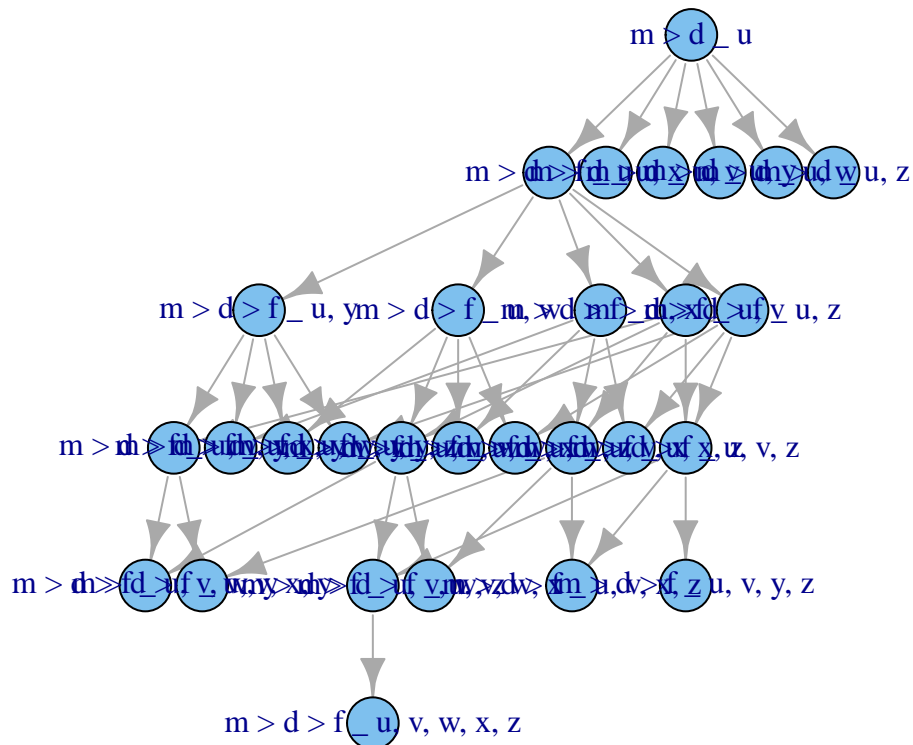


```
## Note we also disable the stopping stochastically as a function of size
## to allow the population to grow large and generate may different
## clones.

## For speed, we set a small finalTime and we fix the seed
## for reproducilibity. Beware: since finalTime is short, sometimes
## we do not reach cancer
set.seed(1)
RNGkind("L'Ecuyer-CMRG")

ospI <- oncoSimulPop(2,
                  o3init, model = "Exp",
                  mu = 5e-5, finalTime = 200,
                  detectionDrivers = 3,
                  onlyCancer = TRUE,
                  initSize = 10,
                  keepPhylog = TRUE,
                  initMutant = c("d > m > z"),
                  mc.cores = 2,
                  seed = NULL
                  )

## Show just one example
```
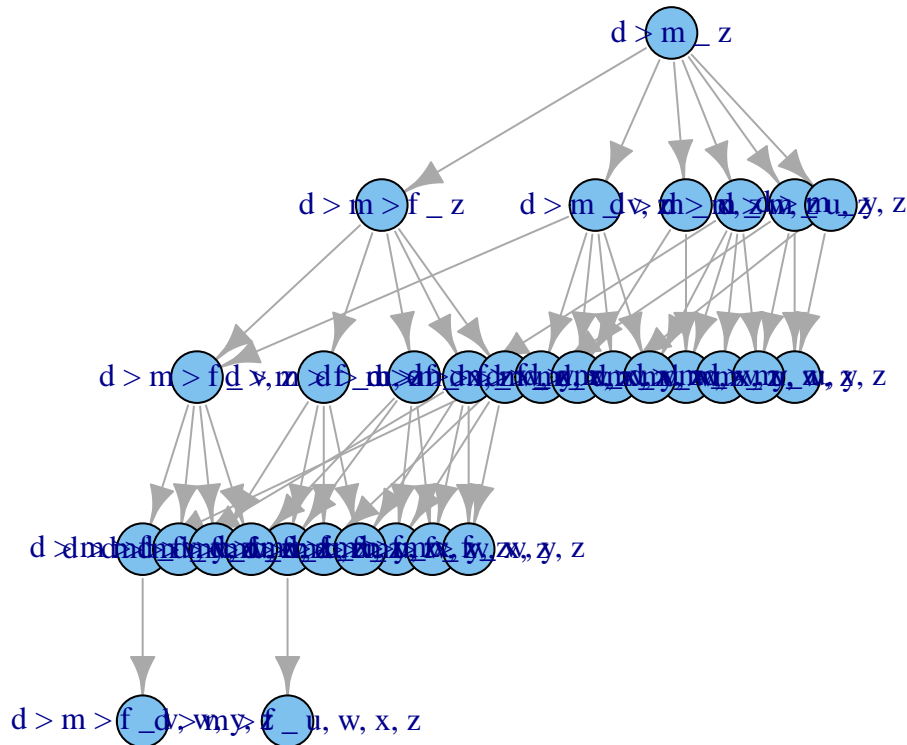
```
## op <- par(mar = rep(0, 4), mfrow = c(1, 2))
plotClonePhylog(ospI[[1]])
```



```
## plotClonePhylog(ospI[[2]])
## par(op)

set.seed(1)
RNGkind("L'Ecuyer-CMRG")
ossI <- oncoSimulSample(2,
                        o3init, model = "Exp",
                        mu = 5e-5, finalTime = 200,
                        detectionDrivers = 2,
                        onlyCancer = TRUE,
                        initSize = 10,
                        initMutant = c("z > d"),
                        ## check presence of initMutant:
                        thresholdWhole = 1,
                        seed = NULL
                        )
## Successfully sampled 2 individuals
##
##  Subjects by Genes matrix of 2 subjects and 9 genes.


## No phylogeny is kept with oncoSimulSample, but look at the
## OcurringDrivers and the sample
ossI$popSample
```

```
##       d f m u v w x y z
## [1,] 1 0 1 0 0 0 0 0 1
## [2,] 1 0 1 0 0 0 0 0 1
```

```
ossI$popSummary[, "OccurringDrivers", drop = FALSE]
##   OccurringDrivers
## 1
## 2
```

Since version 2.21.994, it is possible to start the simulations from arbitrary initial configurations: this uses multiple initial mutants (see section 6.7) and allows for multispecies simulations (section 6.8).

6.3 Ending the simulations

OncoSimulR provides very flexible ways to decide when to stop a simulation. Here we focus on a single simulation; see further options with multiple simulations in 7.

6.3.1 Ending the simulations: conditions

- **onlyCancer = TRUE**. A simulation will be repeated until any one of the "reach cancer" conditions is met, if this happens before the simulation reaches `finalTime`[11]. These conditions are:

  – Total population size becomes larger than `detectionSize`.
  – The number of drivers in any one genotype or clone becomes equal to, or larger than, `detectionDrivers`; note that this allows you to stop the simulation as soon as a **specific genotype** is found, by using exactly and only the genes that make that genotype as the drivers. This is not allowed by the moment in frequency-dependent fitness simulations.
  – A gene or gene combination among those listed in `fixation` becomes fixed in the population (i.e., has a frequency is 1) (see details in (6.3.3 and 6.3.4).
  – The tumor is detected according to a stochastic detection mechanism, where the probability of "detecting the tumor" increases with population size; this is explained below (6.3.2) and is controlled by argument `detectionProb`.

  As we exit as soon as any of the exiting conditions is reached, if you only care about one condition, set the other to `NA` (see also section 6.3.2.1).

- **onlyCancer = FALSE**. A simulation will run only once, and will exit as soon as any of the above conditions are met or as soon as the total population size becomes zero or we reach `finalTime`.

As an example of `onlyCancer = TRUE`, focusing on the first two mechanisms, suppose you give `detectionSize = 1e4` and `detectionDrivers =3` (and you have

---

[11]Of course, the "reach cancer" idea and the `onlyCancer` argument are generic names; this could have been labeled "reach whatever interests me".

`detectionProb = NA`). A simulation will exit as soon as it reaches a total population size of 10^4 or any clone has four drivers, whichever comes first (if any of these happen before `finalTime`).

In the `onlyCancer = TRUE` case, what happens if we reach `finalTime` (or the population size becomes zero) before any of the "reach cancer" conditions have been fulfilled? The simulation will be repeated again, within the following limits:

- `max.wall.time`: the total wall time we allow an individual simulation to run;
- `max.num.tries`: the maximum number of times we allow a simulation to be repeated to reach cancer;
- `max.wall.time.total` and `max.num.tries.total`, similar to the above but over a set of simulations in function `oncoSimulSample`.

Incidentally, we keep track of the number of attempts used (the component `other$attemptsUsed$`) before we reach cancer, so you can estimate (as from a negative binomial sampling) the probability of reaching your desired end point under different scenarios.

The `onlyCancer = FALSE` case might be what you want to do when you examine general population genetics scenarios without focusing on possible sampling issues. To do this, set `finalTime` to the value you want and set `onlyCancer = FALSE`; in addition, set `detectionProb` to "NA" and `detectionDrivers` and `detectionSize` to "NA" or to huge numbers[12]. In this scenario you simply collect the simulation output at the end of the run, regardless of what happened with the population (it became extinct, it did not reach a large size, it did not accumulate drivers, etc).

6.3.2   Stochastic detection mechanism: "detectionProb"

This is the process that is controlled by the argument `detectionProb`. Here the probability of tumor detection increases with the total population size. This is biologically a reasonable assumption: the larger the tumor, the more likely it is it will be detected.

At regularly spaced times during the simulation, we compute the probability of detection as a function of size and determine (by comparing against a random uniform number) if the simulation should finish. For simplicity, and to make sure the probability is bounded between 0 and 1, we use the function

$$P(N) = \begin{cases} 1 - e^{-c((N-B)/B)} & \text{if } N > B \\ 0 & \text{if } N \leq B \end{cases} \tag{1}$$

where $P(N)$ is the probability that a tumor with a population size N will be detected, and c (argument $cPDetect$ in the `oncoSimul*` functions) controls how fast $P(N)$ increases with increasing population size relative to a baseline, B ($PDBaseline$ in the `oncoSimul*` functions); with B we both control the minimal population size at

---

[12]Setting `detectionDrivers` and `detectionSize` to "NA" is in fact equivalent to setting them to the largest possible numbers for these variables: $2^{32} - 1$ and ∞, respectively.

which this mechanism stats operating (because we will rarely want detection unless there is some meaningful increase of population size over `initSize`) and we model the increase in $P(N)$ as a function of relative differences with respect to B. (Note that this is **a major change** in version 2.9.9. Before version 2.9.9, the expression used was $P(N) = 1 - e^{-c(N-B)}$, so we did not make the increase relative to B; of course, you can choose an appropriate c to make different models comparable, but the expression used before 2.9.9 made it much harder to compare simulations with very different initial population sizes, as baselines are often naturall a function of initial population sizes.)

The $P(N)$ refers to the probability of detection at each one of the occasions when we assess the probability of exiting. When, or how often, do we do that? When we assess probability of exiting is controlled by `checkSizePEvery`, which will often be much larger than `sampleEvery`[13]. Biologically, a way to think of `checkSizePEvery` is "time between doctor appointments".

An important **warning**, though: for populations that are growing very, very fast or where some genes might have very large effects on fitness even a moderate `checkSizePEvery` of, say, 10, might be inappropriate, since populations could have increased by several orders of magnitude between successive checks. This issue is also discussed in section 2.1.1 and 2.2.

Finally, you can specify c (*cPDetect*) directly (you will need to set `n2` and `p2` to NA). However, it might be more intuitive to specify the pair `n2`, `p2`, such that $P(n2) = p2$ (and from that pair we solve for the value of *cPDetect*).

You can get a feeling for the effects of these arguments by playing with the following code, that we do not execute here for the sake of speed. Here no mutation has any effect, but there is a non-zero probability of exiting as soon as the total population size becomes larger than the initial population size. So, eventually, all simulations will exit and, as we are using the McFarland model, population size will vary slightly around the initial population size.

```
gi2 <- rep(0, 5)
names(gi2) <- letters[1:5]
oi2 <- allFitnessEffects(noIntGenes = gi2)
s5 <- oncoSimulPop(200,
                   oi2,
                   model = "McFL",
                   initSize = 1000,
                   onlyCancer = TRUE,
                   detectionProb = c(p2 = 0.1,
                                     n2 = 2000,
```

[13]We assess probability of exiting at every sampling time, as given by `sampleEvery`, that is the smallest possible sampling time that is separated from the previous time of assessment by at least `checkSizePEvery`. In other words, the interval between successive assessments will be the smallest multiple integer of `sampleEvery` that is larger than `checkSizePEvery`. For example, suppose `sampleEvery = 2` and `checkSizePEvery = 3`: we will assess exiting at times $4, 8, 12, 16, \ldots$. If `sampleEvery = 3` and `checkSizePEvery = 3`: we will assess exiting at times $6, 12, 18, \ldots$.

```
                                    PDBaseline = 1000,
                                    checkSizePEvery = 2),
                  detectionSize = NA,
                  finalTime = NA,
                  keepEvery = NA,
                  detectionDrivers = NA)
s5
hist(unlist(lapply(s5, function(x) x$FinalTime)))
```

As you decrease `checkSizePEvery` the distribution of "FinalTime" will resemble more and more an exponential distribution.

In this vignette, there are some further examples of using this mechanism in 6.5.4 and 6.5.2, with the default arguments.

6.3.2.1 Stochastic detection mechanism and minimum number of drivers

We said above that we exit as soon as any of the conditions is reached (i.e., we use an OR operation over the exit conditions). There is a special exception to this procedure: if you set `AND_DrvProbExit = TRUE`, both the number of drivers and the `detectionProb` mechanism condition must fulfilled. This means that the `detectionProb` mechanism not assessed unless the `detectionDrivers` condition is. Using `AND_DrvProbExit = TRUE` allows to run simulations and ensure that all of the returned simulations will have at least some cells with the number of drivers as specified by `detectionDrivers`. Note, though, that this does not guarantee that when you sample the population, all those drivers will be detected (as this depends on the actual proportion of cells with the drivers and the settings of `samplePop`).

6.3.3 Fixation of genes/gene combinations

In some cases we might be interested in running simulations until a particular set of genes, or gene combinations, reaches fixation. This exit condition might be more relevant than some of the above in many non-cancer-related evolutionary genetics scenarios.

Simulations will stop as soon as any of the genes or gene combinations in the vector (or list) `fixation` reaches a frequency of 1. These gene combinations might have non-zero intersection (i.e., they might share genes), and those genes need not be drivers. If we want simulations to only stop when fixation of those genes/gene combinations is reached, we will set all other stopping conditions to `NA`. It is, of course, up to you to ensure that those stopping conditions are reasonable (that they can be reached) and to use, or not, `finalTime`; otherwise, simulations will eventually abort (e.g., when `max.wall.time` or `max.num.tries` are reached). Since we are asking for fixation, the `Exp` or `Bozic` models will often not be appropriate here; instead, models with competition such as `McFL` are more appropriate.

We return here to the example from section 5.3.

```
u <- 0.2; i <- -0.02; vi <- 0.6; ui <- uv <- -Inf
od2 <- allFitnessEffects(
    epistasis = c("u" = u,  "u:i" = ui,
                  "u:v" = uv, "i" = i,
                  "v:-i" = -Inf, "v:i" = vi))
```

Ochs and Desai explain that "Each simulated population was evolved until either the uphill genotype or valley-crossing genotype fixed." (see Ochs & Desai (2015), p.2, section "Simulations"). We will do the same here. We specify that we want to end the simulation when either the "u" or the "v, i" genotypes have reached fixation, by passing those genotype combinations as the `fixation` argument (in this example using `fixation = c("u", "v")` would have been equivalent, since the "v" genotype by itself has fitness of 0).

We want to be explicit that fixation will be the one and only condition for ending the simulations, and thus we set arguments `detectionDrivers`, `finalTime`, `detectionSize` and `detectionProb` explicitly to `NA`. (We set the number of repetitions only to 10 for the sake of speed when creating the vignette).

```
initS <- 20
## We use only a small number of repetitions for the sake
## of speed. Even fewer in Windows, since we run on a single
## core

if(.Platform$OS.type == "windows") {
    nruns <- 4
} else {
    nruns <- 10
}

od100 <- oncoSimulPop(nruns, od2,
                      fixation = c("u", "v, i"),
                      model = "McFL",
                      mu = 1e-4,
                      detectionDrivers = NA,
                      finalTime = NA,
                      detectionSize = NA,
                      detectionProb = NA,
                      onlyCancer = TRUE,
                      initSize = initS,
                      mc.cores = 2)
```

What is the frequency of each genotype among the simulations? (or, what is the frequency of fixation of each genotype?)

```
sampledGenotypes(samplePop(od100))
##
##  Subjects by Genes matrix of 10 subjects and 3 genes.
```

```
##    Genotype Freq
## 1    i, v    1
## 2       u    9
##
##  Shannon's diversity (entropy) of sampled genotypes: 0.3251
```

Note the very large variability in reaching fixation

```
head(summary(od100)[, c(1:3, 8:9)])
##   NumClones TotalPopSize LargestClone FinalTime NumIter
## 1         3           31           31    3386.7  135482
## 2         3           23           23     820.3   32821
## 3         3           20           20     391.8   15675
## 4         3           18           18    2417.9   96726
## 5         4           44           44    5571.6  222895
## 6         3           31           31    2980.9  119250
```

6.3.4 Fixation of genotypes

Section 6.3.3 deals with the fixation of gene/gene combinations. What if you want fixation on specific genotypes? To give an example, suppose we have a five loci genotype and suppose that you want to stop the simulations only if genotypes "A", "B, E", or "A, B, C, D, E" reach fixation. You do not want to stop it if, say, genotype "A, B, E" reaches fixation. To specify genotypes, you prepend the genotype combinations with a "_,", and that tells OncoSimulR that you want fixation of **genotypes**, not just gene combinations.

An example of the differences between the mechanisms can be seen from this code:

```
## Create a simple fitness landscape
rl1 <- matrix(0, ncol = 6, nrow = 9)
colnames(rl1) <- c(LETTERS[1:5], "Fitness")
rl1[1, 6] <- 1
rl1[cbind((2:4), c(1:3))] <- 1
rl1[2, 6] <- 1.4
rl1[3, 6] <- 1.32
rl1[4, 6] <- 1.32
rl1[5, ] <- c(0, 1, 0, 0, 1, 1.5)
rl1[6, ] <- c(0, 0, 1, 1, 0, 1.54)
rl1[7, ] <- c(1, 0, 1, 1, 0, 1.65)
rl1[8, ] <- c(1, 1, 1, 1, 0, 1.75)
rl1[9, ] <- c(1, 1, 1, 1, 1, 1.85)
class(rl1) <- c("matrix", "genotype_fitness_matrix")
## plot(rl1) ## to see the fitness landscape

## Gene combinations
local_max_g <- c("A", "B, E", "A, B, C, D, E")
## Specify the genotypes
```

168

```
local_max <- paste0("_,", local_max_g)

fr1 <- allFitnessEffects(genotFitness = rl1, drvNames = LETTERS[1:5])
initS <- 2000


######## Stop on gene combinations   #####
r1 <- oncoSimulPop(10,
                      fp = fr1,
                      model = "McFL",
                      initSize = initS,
                      mu = 1e-4,
                      detectionSize = NA,
                      sampleEvery = .03,
                      keepEvery = 1,
                      finalTime = 50000,
                      fixation = local_max_g,
                      detectionDrivers = NA,
                      detectionProb = NA,
                      onlyCancer = TRUE,
                      max.num.tries = 500,
                      max.wall.time = 20,
                      errorHitMaxTries = TRUE,
                      keepPhylog = FALSE,
                      mc.cores = 2)
sp1 <- samplePop(r1, "last", "singleCell")
##
##  Subjects by Genes matrix of 10 subjects and 5 genes.
```

```
sgsp1 <- sampledGenotypes(sp1)
## often you will stop on gene combinations that
## are not local maxima in the fitness landscape
sgsp1
##    Genotype Freq
## 1         A    6
## 2   A, C, D    2
## 3      B, E    2
##
##  Shannon's diversity (entropy) of sampled genotypes: 0.9503
```

```
sgsp1$Genotype %in% local_max_g
## [1]  TRUE FALSE  TRUE
```

```
####### Stop on genotypes    ####

r2 <- oncoSimulPop(10,
```

```
                    fp = fr1,
                    model = "McFL",
                    initSize = initS,
                    mu = 1e-4,
                    detectionSize = NA,
                    sampleEvery = .03,
                    keepEvery = 1,
                    finalTime = 50000,
                    fixation = local_max,
                    detectionDrivers = NA,
                    detectionProb = NA,
                    onlyCancer = TRUE,
                    max.num.tries = 500,
                    max.wall.time = 20,
                    errorHitMaxTries = TRUE,
                    keepPhylog = FALSE,
                    mc.cores = 2)
## All final genotypes should be local maxima
sp2 <- samplePop(r2, "last", "singleCell")
##
##  Subjects by Genes matrix of 10 subjects and 5 genes.
```

```
sgsp2 <- sampledGenotypes(sp2)
sgsp2$Genotype %in% local_max_g
## [1] TRUE TRUE TRUE
```

6.3.5 Fixation: tolerance, number of periods, minimal size

In particular if you specify stopping on genotypes, you might want to think about three additional parameters: `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).

170

An example of using those options:

```
## Create a simple fitness landscape
rl1 <- matrix(0, ncol = 6, nrow = 9)
colnames(rl1) <- c(LETTERS[1:5], "Fitness")
rl1[1, 6] <- 1
rl1[cbind((2:4), c(1:3))] <- 1
rl1[2, 6] <- 1.4
rl1[3, 6] <- 1.32
rl1[4, 6] <- 1.32
rl1[5, ] <- c(0, 1, 0, 0, 1, 1.5)
rl1[6, ] <- c(0, 0, 1, 1, 0, 1.54)
rl1[7, ] <- c(1, 0, 1, 1, 0, 1.65)
rl1[8, ] <- c(1, 1, 1, 1, 0, 1.75)
rl1[9, ] <- c(1, 1, 1, 1, 1, 1.85)
class(rl1) <- c("matrix", "genotype_fitness_matrix")
## plot(rl1) ## to see the fitness landscape

## The local fitness maxima are
## c("A", "B, E", "A, B, C, D, E")

fr1 <- allFitnessEffects(genotFitness = rl1, drvNames = LETTERS[1:5])
initS <- 2000


## Stop on genotypes

r3 <- oncoSimulPop(10,
                   fp = fr1,
                   model = "McFL",
                   initSize = initS,
                   mu = 1e-4,
                   detectionSize = NA,
                   sampleEvery = .03,
                   keepEvery = 1,
                   finalTime = 50000,
                   fixation = c(paste0("_,",
                               c("A", "B, E", "A, B, C, D, E")),
                            fixation_tolerance = 0.1,
                            min_successive_fixation = 200,
                            fixation_min_size = 3000),
                   detectionDrivers = NA,
                   detectionProb = NA,
                   onlyCancer = TRUE,
                   max.num.tries = 500,
                   max.wall.time = 20,
                   errorHitMaxTries = TRUE,
                   keepPhylog = FALSE,
```

```
              mc.cores = 2)
```

6.3.6 Mixing stopping on gene combinations and genotypes

This would probably be awfully confusing and is not tested formally (though it should work). Let me know if you think this is an important feature. (Pull requests with tests welcome.)

6.4 Plotting genotype/driver abundance over time; plotting the simulated trajectories

We have seen many of these plots already, starting with Figure 2 and Figure 4 and we will see many more below, in the examples, starting with section 6.5.1 such as in figures 5 and 6. In a nutshell, what we are plotting is the information contained in the `pops.by.time` matrix, the matrix that contains the abundances of all the clones (or genotypes) at each of the sampling periods.

The functions that do the work are called `plot` and these are actually methods for objects of class "oncosimul" and "oncosimulpop". You can access the help by doing `?plot.oncosimul`, for example.

What entities are shown in the plot? You can show the trajectories of:

- numbers of drivers (e.g., 5);
- genotypes or clones (e.g., 6).

(Of course, showing "drivers" requires that you have specified certain genes as drivers.)

What types of plots are available?

- line plots;
- stacked plots;
- stream plots.

All those three are shown in both of Figure 5 and Figure 6.

If you run multiple simulations using `oncoSimulPop` you can plot the trajectories of all of the simulations.

6.5 Several examples of simulations and plotting simulation trajectories

6.5.1 Bauer's example again

We will use the model of Bauer et al. (2014) that we saw in section 5.1.

```
K <- 5
sd <- 0.1
sdp <- 0.15
```

```
sp <- 0.05
bauer <- data.frame(parent = c("Root", rep("p", K)),
                    child = c("p", paste0("s", 1:K)),
                    s = c(sd, rep(sdp, K)),
                    sh = c(0, rep(sp, K)),
                    typeDep = "MN")
fbauer <- allFitnessEffects(bauer, drvNames = "p")
set.seed(1)
## Use fairly large mutation rate
b1 <- oncoSimulIndiv(fbauer, mu = 5e-5, initSize = 1000,
                     finalTime = NA,
                     onlyCancer = TRUE,
                     detectionProb = "default")
```

We will now use a variety of plots

```
par(mfrow = c(3, 1))
## First, drivers
plot(b1, type = "line", addtot = TRUE)
plot(b1, type = "stacked")
plot(b1, type = "stream")
```

```
par(mfrow = c(3, 1))
## Next, genotypes
plot(b1, show = "genotypes", type = "line")
plot(b1, show = "genotypes", type = "stacked")
plot(b1, show = "genotypes", type = "stream")
```

In this case, probably the stream plots are most helpful. Note, however, that (in contrast to some figures in the literature showing models of clonal expansion) the stream plot (or the stacked plot) does not try to explicitly show parent-descendant relationships, which would hardly be realistically possible in these plots (although the plots of phylogenies in section 8 could be of help).

6.5.2 McFarland model with 5000 passengers and 70 drivers

```
set.seed(678)
nd <- 70
np <- 5000
s <- 0.1
sp <- 1e-3
spp <- -sp/(1 + sp)
mcf1 <- allFitnessEffects(noIntGenes = c(rep(s, nd), rep(spp, np)),
                          drvNames = seq.int(nd))
mcf1s <-  oncoSimulIndiv(mcf1,
                         model = "McFL",
```
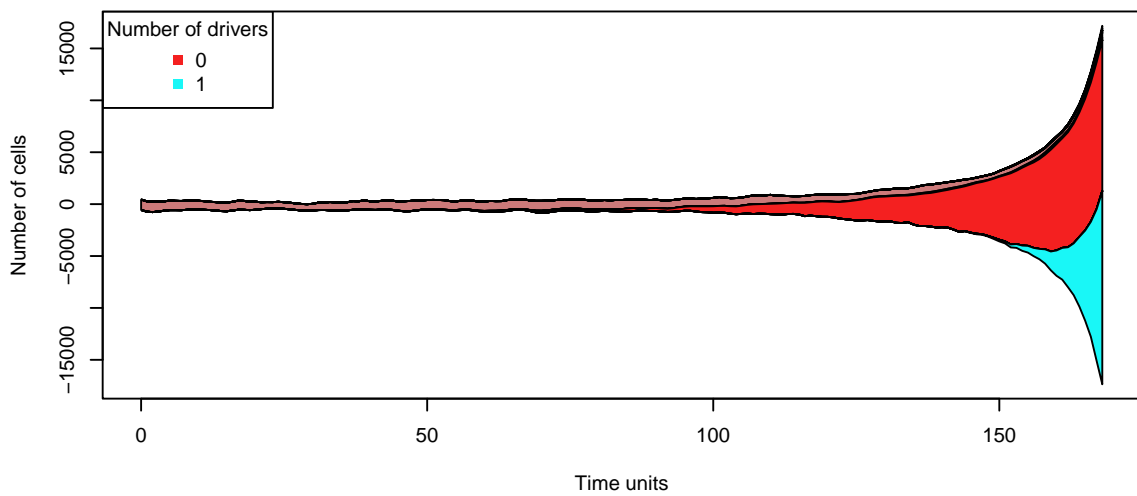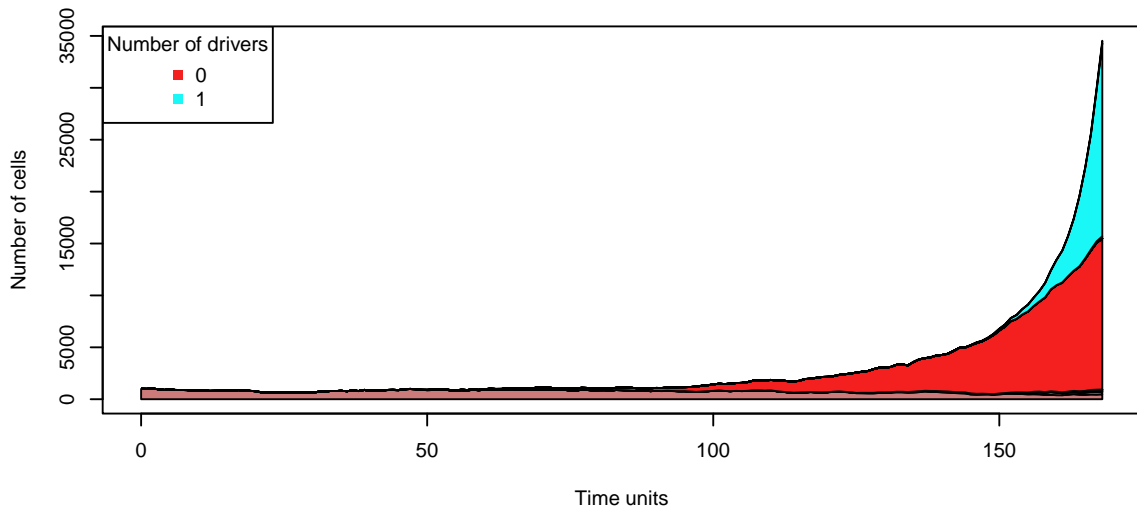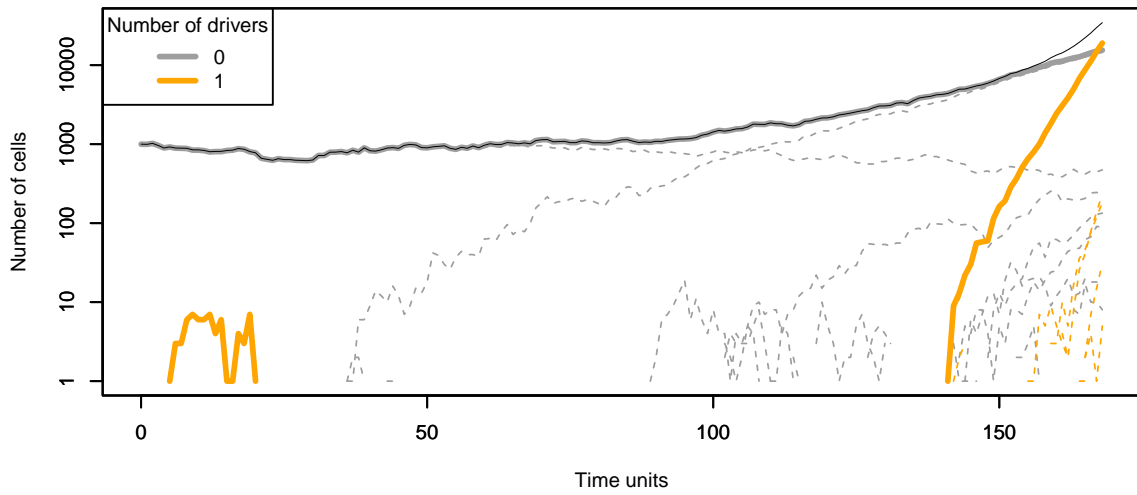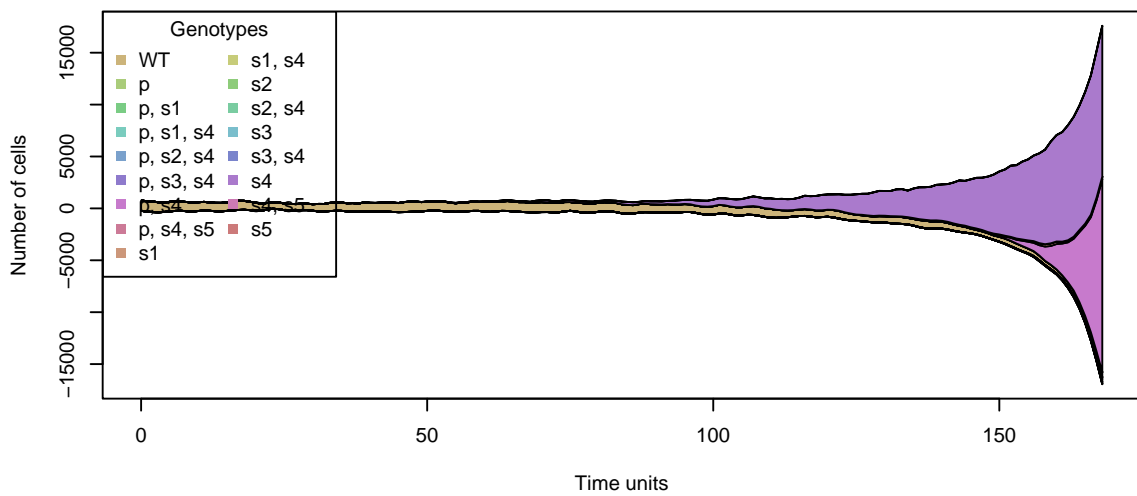
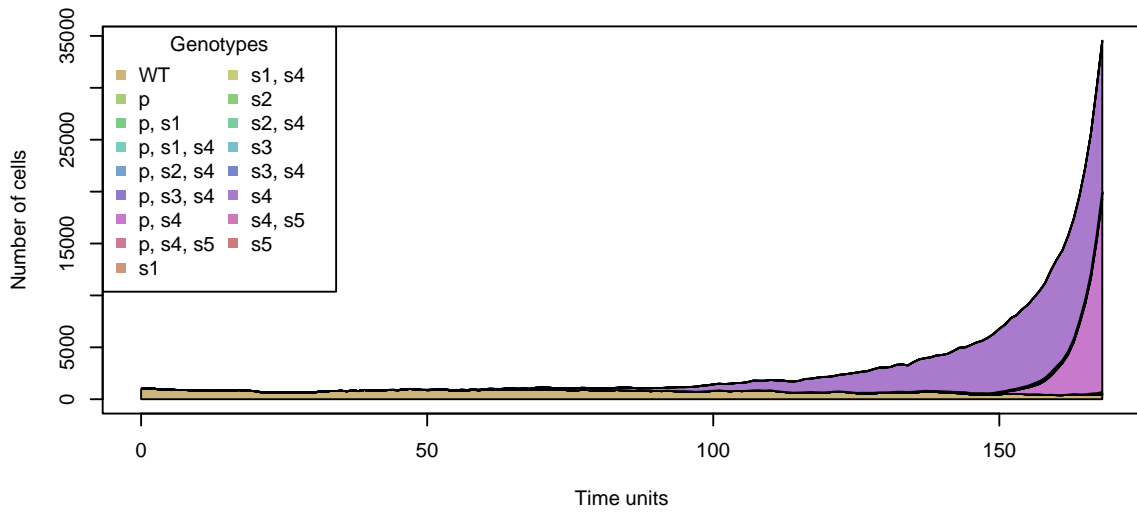Figure 5: Three drivers' plots of a simulation of Bauer's model

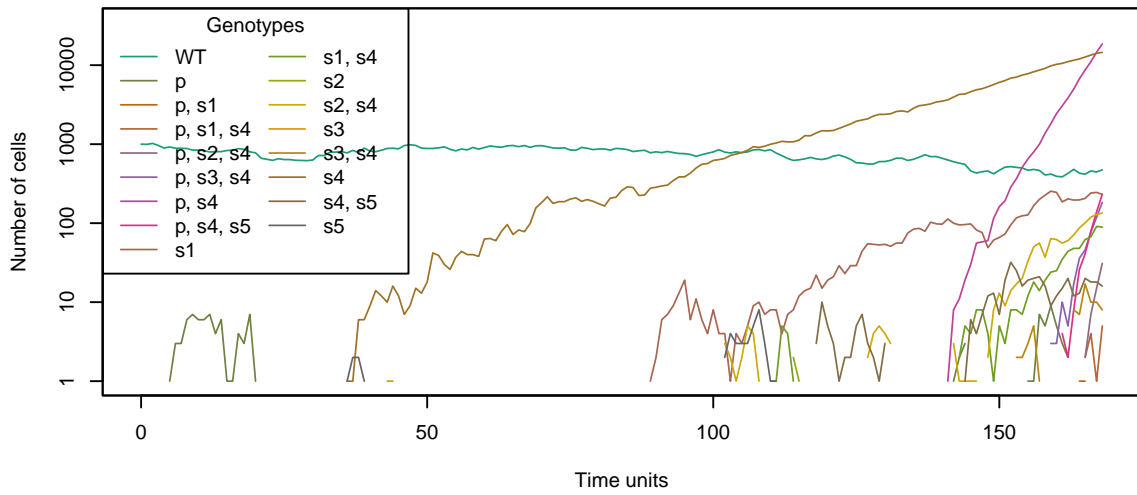Figure 6: Three genotypes' plots of a simulation of Bauer's model

```
                              mu = 1e-7,
                              detectionProb = "default",
                              detectionSize = NA,
                              detectionDrivers = NA,
                              sampleEvery = 0.025,
                              keepEvery = 8,
                              initSize = 2000,
                              finalTime = 4000,
                              onlyCancer = FALSE)
summary(mcf1s)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1       973         2855         2412             3              2
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    2                  11      2841  117409
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.01309       1695       1972
##                               OccurringDrivers
## 1 7, 9, 10, 11, 16, 20, 22, 23, 33, 45, 59
```
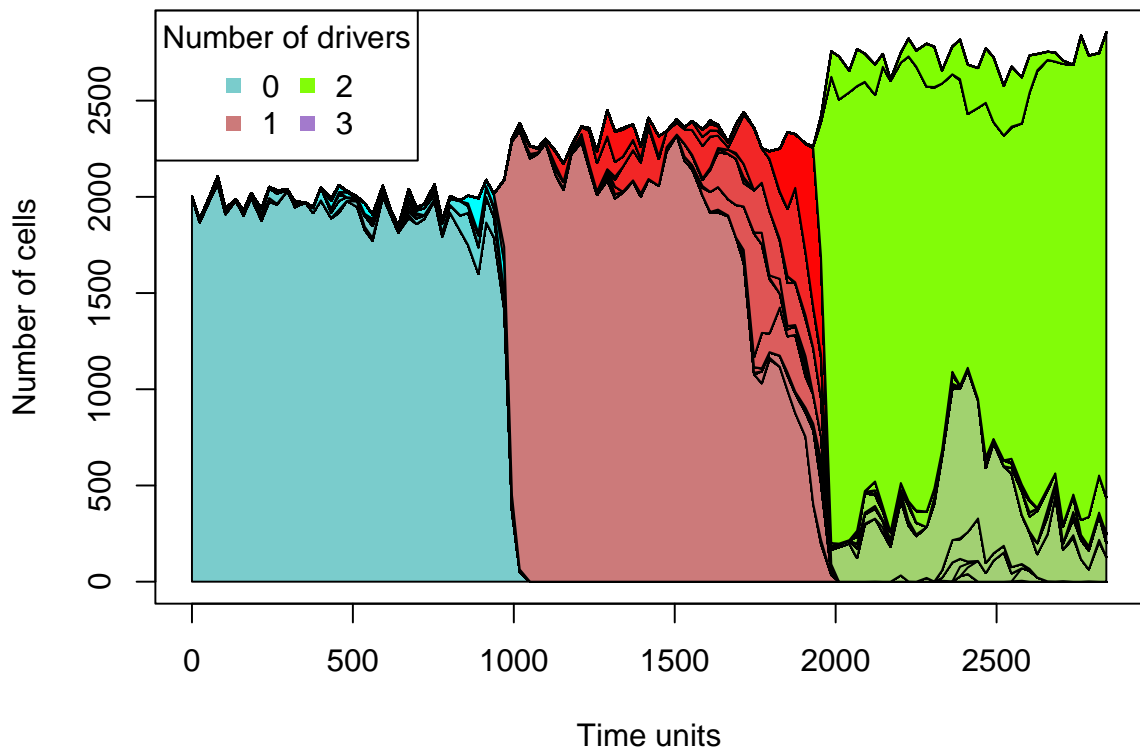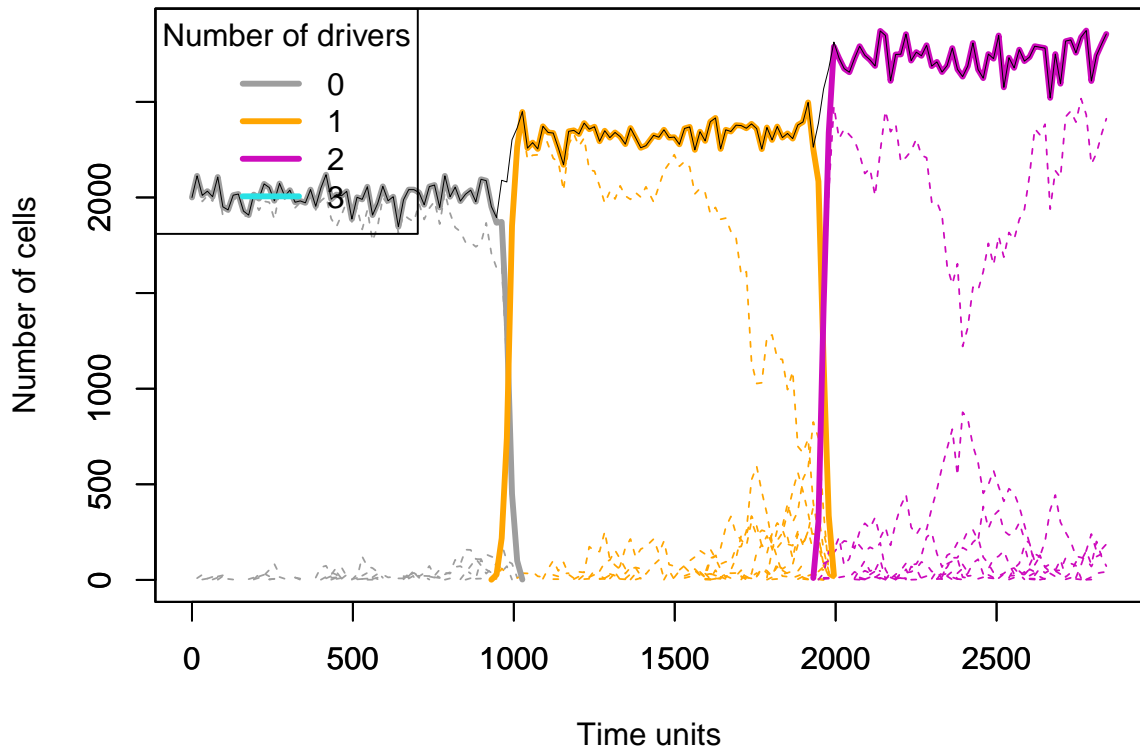
```
par(mfrow  = c(2, 1))
## I use thinData to make figures smaller and faster
plot(mcf1s, addtot = TRUE, lwdClone = 0.9, log = "",
     thinData = TRUE, thinData.keep = 0.5)
plot(mcf1s, show = "drivers", type = "stacked",
     thinData = TRUE, thinData.keep = 0.3,
     legend.ncols = 2)
```

With the above output (where we see there are over 500 different genotypes) trying to represent the genotypes makes no sense.

6.5.3 McFarland model with 50,000 passengers and 70 drivers: clonal competition

The next is too slow (takes a couple of minutes in an i5 laptop) and too big to run in a vignette, because we keep track of over 4000 different clones (which leads to a result object of over 800 MB):

```
set.seed(123)
nd <- 70
np <- 50000
s <- 0.1
sp <- 1e-4 ## as we have many more passengers
spp <- -sp/(1 + sp)
mcfL <- allFitnessEffects(noIntGenes = c(rep(s, nd), rep(spp, np)),
                          drvNames = seq.int(nd))
mcfLs <-  oncoSimulIndiv(mcfL,
                         model = "McFL",
                         mu = 1e-7,
                         detectionSize = 1e8,
                         detectionDrivers = 100,
                         sampleEvery = 0.02,
                         keepEvery = 2,
                         initSize = 1000,
                         finalTime = 2000,
                         onlyCancer = FALSE)
```

But you can access the pre-stored results and plot them (beware: this object has been trimmed by removing empty passenger rows in the Genotype matrix)

```
data(mcfLs)
plot(mcfLs, addtot = TRUE, lwdClone = 0.9, log = "",
     thinData = TRUE, thinData.keep = 0.3,
     plotDiversity = TRUE)
```
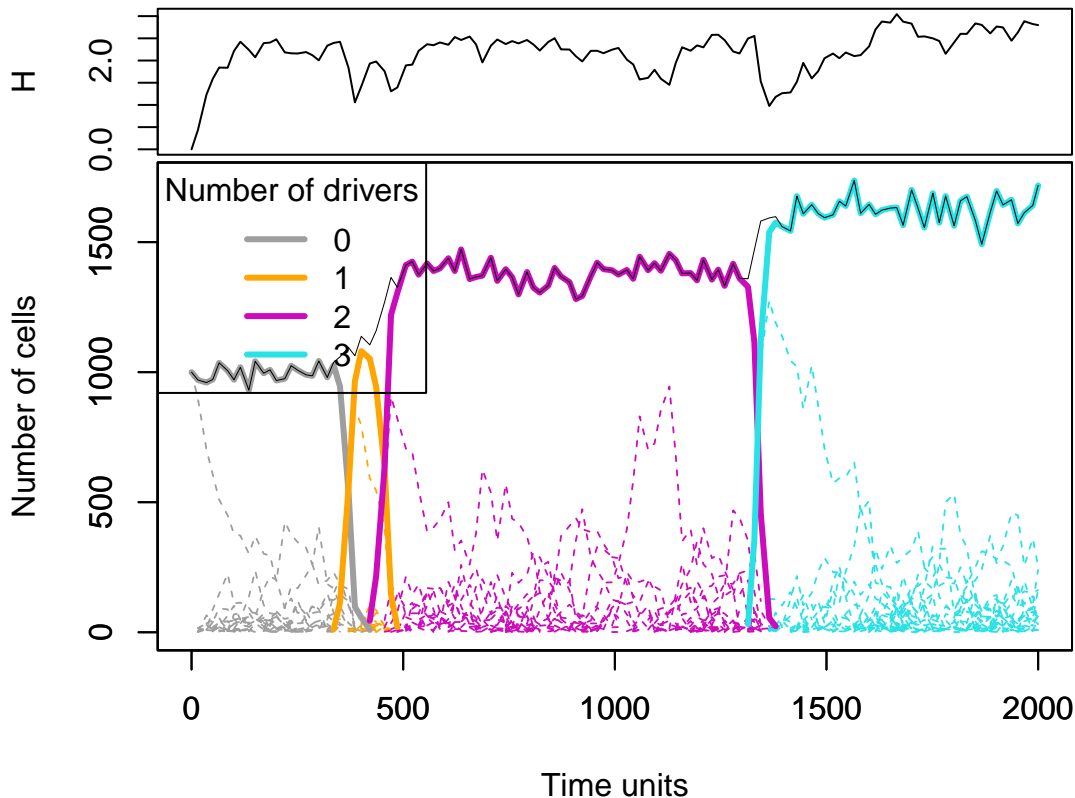
The argument `plotDiversity = TRUE` asks to show a small plot on top with Shannon's diversity index.

```
summary(mcfLs)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1      4458         1718          253             3              3
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    3                  70      2000  113759
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.01922      184.1      199.6
##   OccurringDrivers
## 1   13, 38, 40, 69
```

```
## number of passengers per clone
summary(colSums(mcfLs$Genotypes[-(1:70), ]))
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    4.00    6.00    5.67    7.75   13.00
```

Note that we see clonal competition between clones with the same number of drivers (and with different drivers, of course). We will return to this (section 6.5.5).

A stacked plot might be better to show the extent of clonal competition (plotting takes some time —a stream plot reveals similar patterns and is also slower than the line plot). I will aggressively thin the data for this plot so it is faster and smaller (but we miss some of the fine grain, of course):

```
plot(mcfLs, type = "stacked", thinData = TRUE,
     thinData.keep = 0.1,
     plotDiversity = TRUE,
     xlim = c(0, 1000))
```
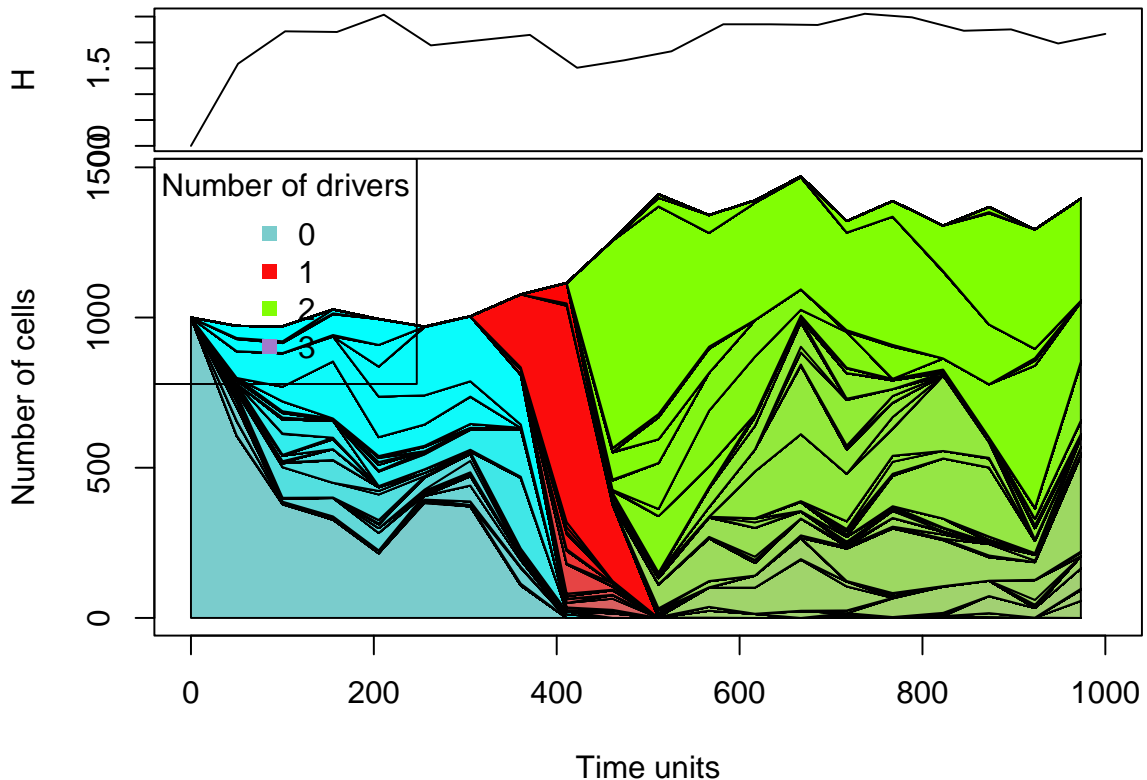

6.5.4 Simulation with a conjunction example

We will use several of the previous examples. Most of them are in file
examplesFitnessEffects, where they are stored inside a list, with named
components (names the same as in the examples above):

```
data(examplesFitnessEffects)
names(examplesFitnessEffects)
##  [1] "cbn1"    "cbn2"    "smn1"    "xor1"    "fp3"     "fp4m"    "o3"
##  [8] "ofe1"    "ofe2"    "foi1"    "sv"      "svB"     "svB1"    "sv2"
## [15] "sm1"     "e2"      "E3A"     "em"      "fea"     "fbauer" "w"
## [22] "pancr"
```

```
set.seed(1)
```

We will simulate using the simple CBN-like restrictions of section 3.4.4 with two
different models.

```
data(examplesFitnessEffects)
evalAllGenotypes(examplesFitnessEffects$cbn1, order = FALSE)[1:10, ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##    Genotype Fitness
## 1        a    1.10
## 2        b    1.10
## 3        c    0.10
## 4        d    1.10
## 5        e    1.10
## 6        g    0.10
## 7     a, b    1.21
## 8     a, c    0.11
## 9     a, d    1.21
## 10    a, e    1.21
```

```
sm <- oncoSimulIndiv(examplesFitnessEffects$cbn1,
                     model = "McFL",
                     mu = 5e-7,
                     detectionSize = 1e8,
                     detectionDrivers = 2,
                     detectionProb = "default",
                     sampleEvery = 0.025,
                     keepEvery = 5,
                     initSize = 2000,
                     onlyCancer = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
```

```
summary(sm)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         4         2635         2014             2              2
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    2                   3     372.5   14905
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.01266     314478     333333
##   OccurringDrivers
## 1          a, b, e
```

```
set.seed(1234)
evalAllGenotypes(examplesFitnessEffects$cbn1, order = FALSE,
             model = "Bozic")[1:10, ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##    Genotype Death_rate
```

```
## 1          a        0.90
## 2          b        0.90
## 3          c        1.90
## 4          d        0.90
## 5          e        0.90
## 6          g        1.90
## 7       a, b        0.81
## 8       a, c        1.71
## 9       a, d        0.81
## 10      a, e        0.81
```

```
sb <-  oncoSimulIndiv(examplesFitnessEffects$cbn1,
                      model = "Bozic",
                      mu = 5e-6,
                     detectionProb = "default",
                      detectionSize = 1e8,
                      detectionDrivers = 4,
                      sampleEvery = 2,
                      initSize = 2000,
                      onlyCancer = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
summary(sb)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1        12        26655        25030             2              2
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    1                   6       550     310
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA      33333      33333
##   OccurringDrivers
## 1 a, b, c, d, e, g
```

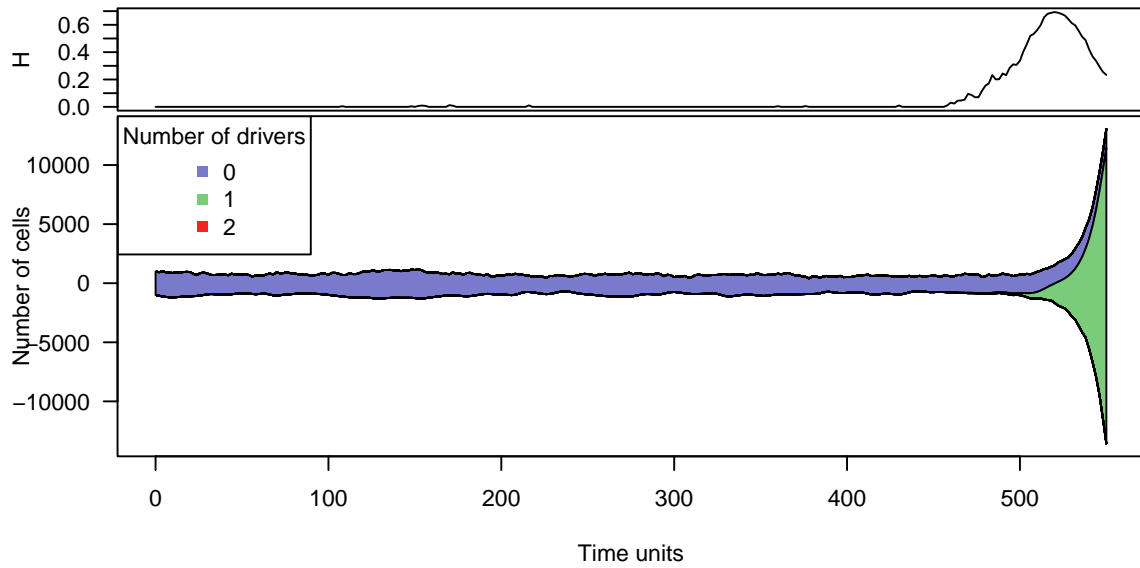As usual, we will use several plots here.

```
## Show drivers, line plot
par(cex = 0.75, las = 1)
plot(sb,show = "drivers", type = "line", addtot = TRUE,
     plotDiversity = TRUE)
```
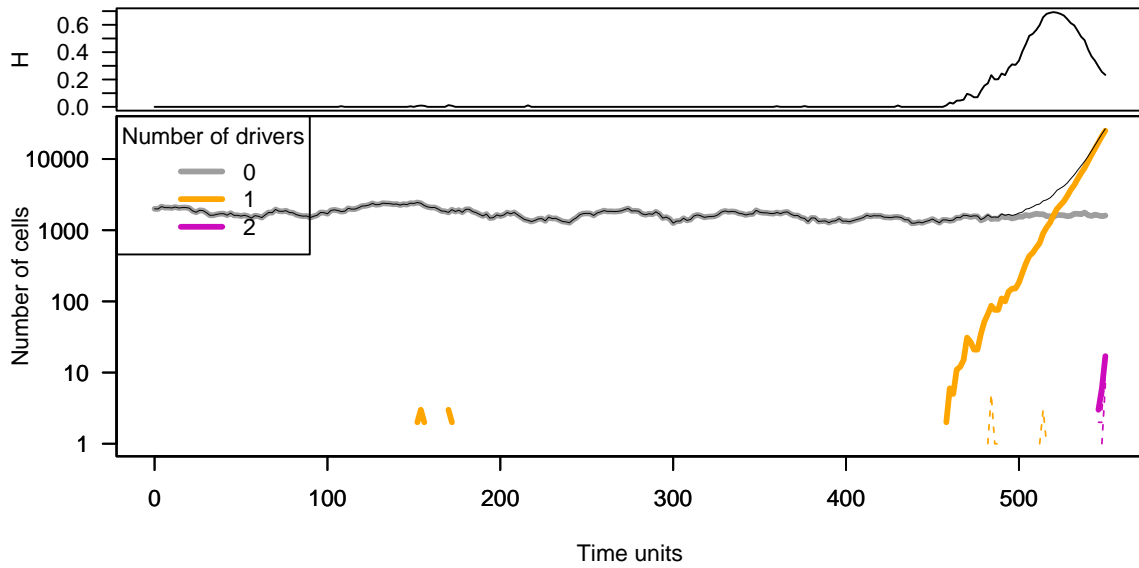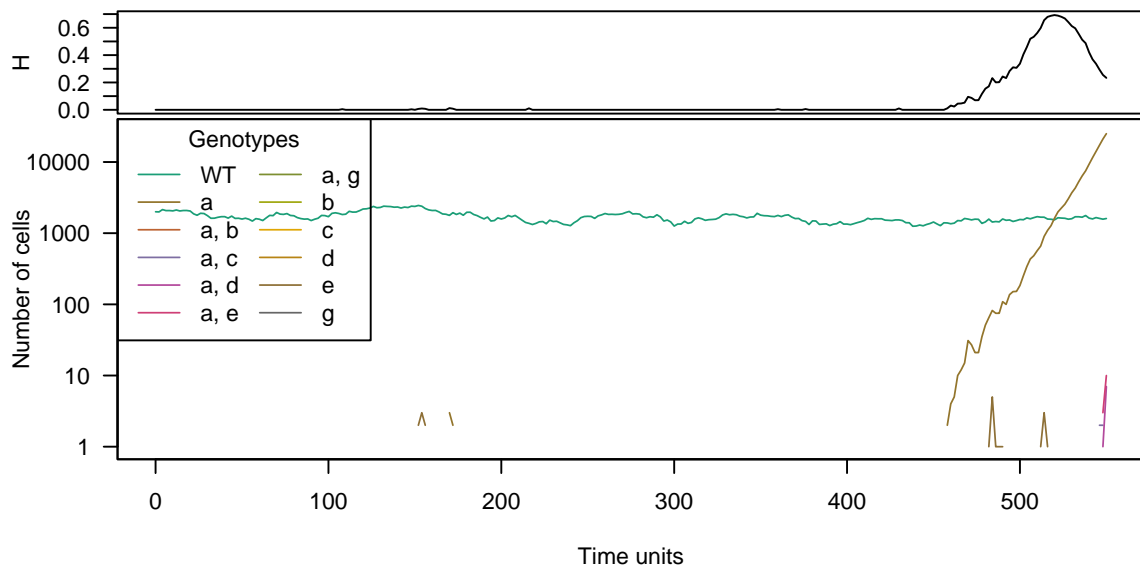
```
## Drivers, stacked
par(cex = 0.75, las = 1)
plot(sb,show = "drivers", type = "stacked", plotDiversity = TRUE)
```
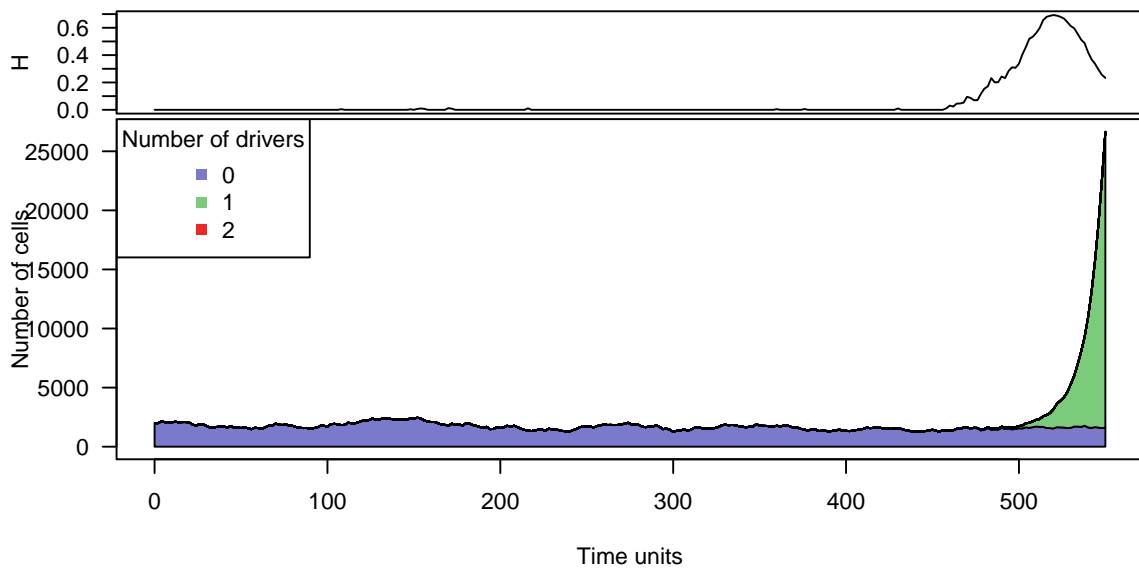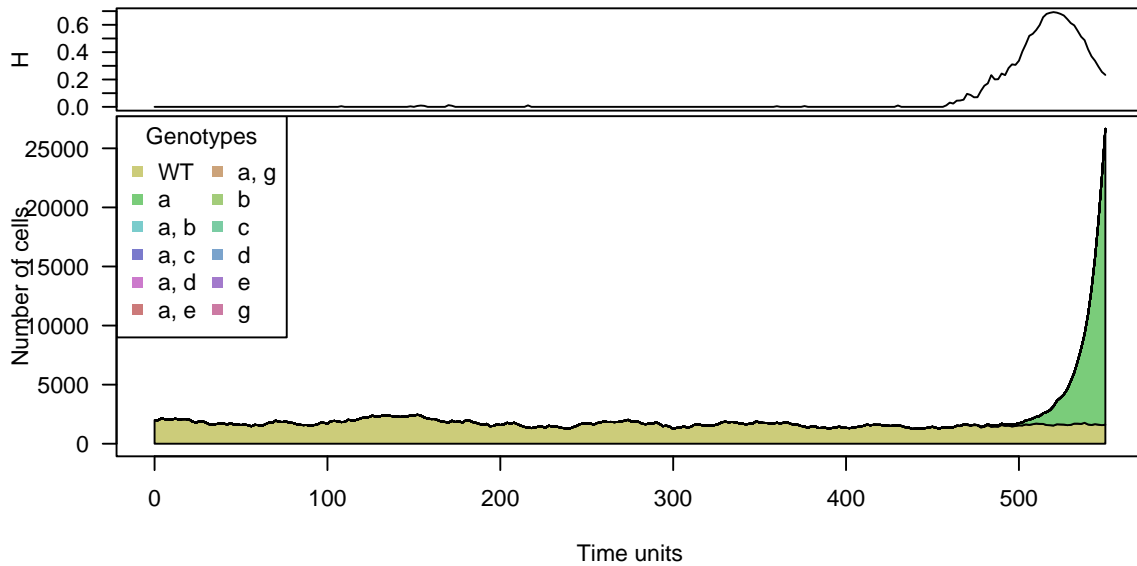


```
## Drivers, stream
par(cex = 0.75, las = 1)
plot(sb,show = "drivers", type = "stream", plotDiversity = TRUE)
```
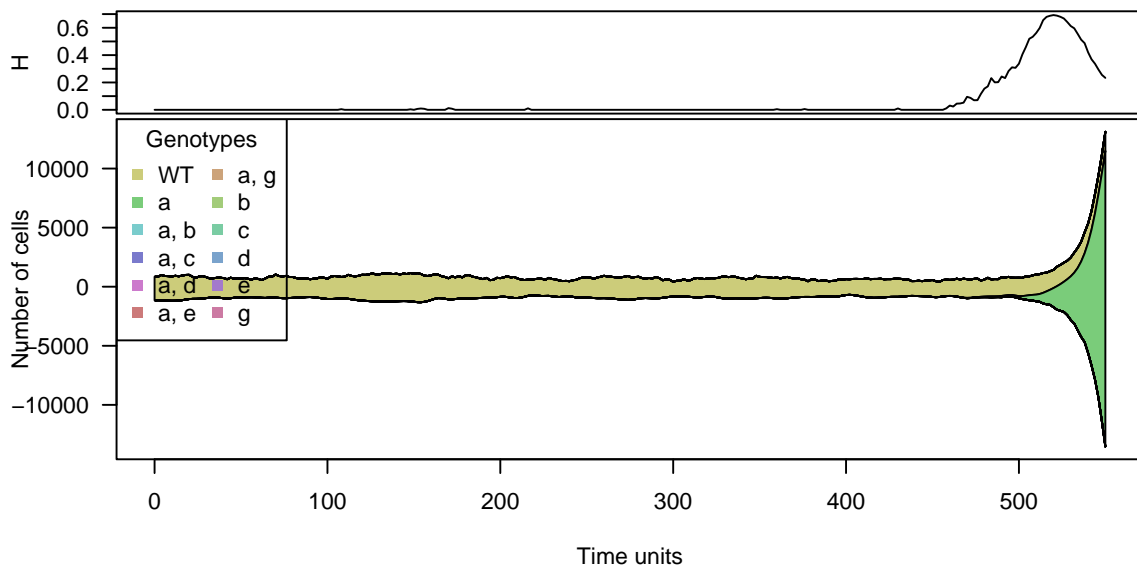
```
## Genotypes, line plot
par(cex = 0.75, las = 1)
plot(sb,show = "genotypes", type = "line", plotDiversity = TRUE)
```



```
## Genotypes, stacked
par(cex = 0.75, las = 1)
plot(sb,show = "genotypes", type = "stacked", plotDiversity = TRUE)
```

```
## Genotypes, stream
par(cex = 0.75, las = 1)
plot(sb,show = "genotypes", type = "stream", plotDiversity = TRUE)
```


The above illustrates again that different types of plots can be useful to reveal different patterns in the data. For instance, here, because of the huge relative frequency of one of the clones/genotypes, the stacked and stream plots do not reveal the other clones/genotypes as we cannot use a log-transformed y-axis, even if there are other clones/genotypes present.

6.5.5 Simulation with order effects and McFL model

(We use a somewhat large mutation rate than usual, so that the simulation runs quickly.)

```
set.seed(4321)
tmp <- oncoSimulIndiv(examplesFitnessEffects[["o3"]],
```

```
                           model = "McFL",
                           mu = 5e-5,
                           detectionSize = 1e8,
                           detectionDrivers = 3,
                           sampleEvery = 0.025,
                           max.num.tries = 10,
                           keepEvery = 5,
                           initSize = 2000,
                           finalTime = 6000,
                           onlyCancer = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```
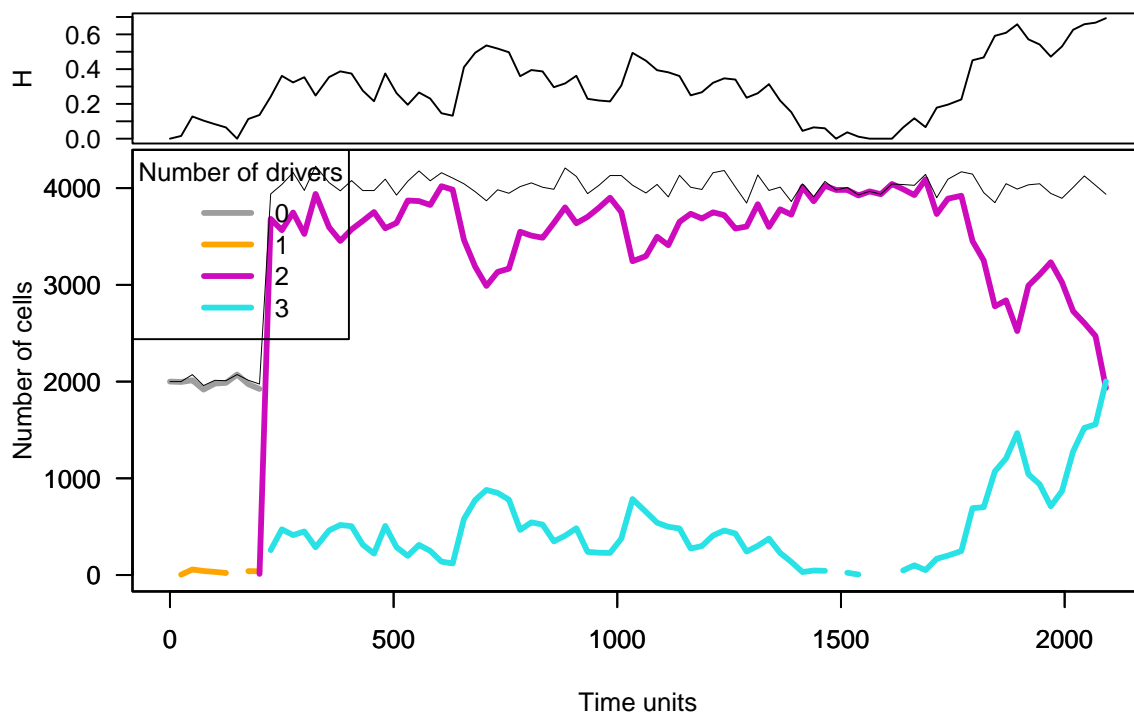
We show a stacked and a line plot of the drivers:

```
par(las = 1, cex = 0.85)
plot(tmp, addtot = TRUE, log = "", plotDiversity = TRUE,
     thinData = TRUE, thinData.keep = 0.2)
```
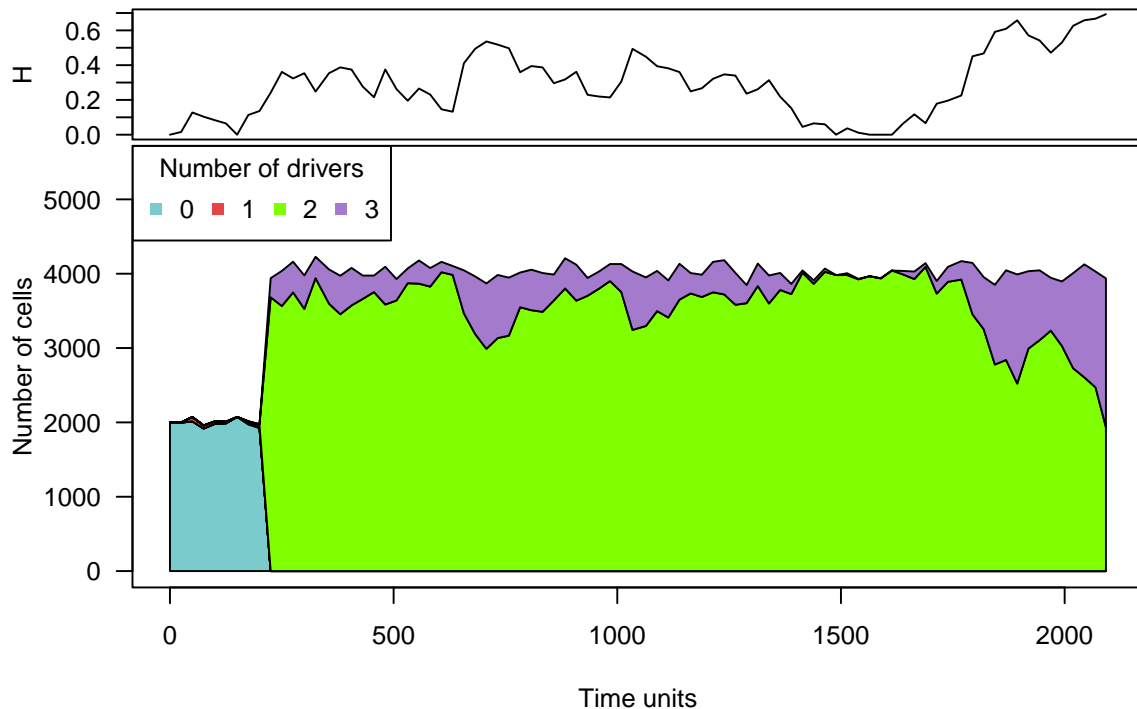


```
par(las = 1, cex = 0.85)
plot(tmp, type = "stacked", plotDiversity = TRUE,
     ylim = c(0, 5500), legend.ncols = 4,
     thinData = TRUE, thinData.keep = 0.2)
```

In this example (and at least under Linux, with both GCC and clang —random number streams in C++, and thus simulations, can differ between combinations of operating system and compiler), we can see that the mutants with three drivers do not get established when we stop the simulation at time 6000. This is one case where the summary statistics about number of drivers says little of value, as fitness is very different for genotypes with the same number of mutations, and does not increase in a simple way with drivers:

```
evalAllGenotypes(examplesFitnessEffects[["o3"]], addwt = TRUE,
                 order = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##     Genotype Fitness
## 1         WT    1.00
## 2          d    1.00
## 3          f    1.00
## 4          m    1.00
## 5      d > f    1.00
## 6      d > m    1.10
## 7      f > d    1.00
## 8      f > m    1.00
## 9      m > d    1.50
## 10     m > f    1.00
## 11 d > f > m    1.54
## 12 d > m > f    1.32
## 13 f > d > m    0.77
```

```
## 14 f > m > d    1.50
## 15 m > d > f    1.50
## 16 m > f > d    1.50
```

A few figures could help:

```
plot(tmp, show = "genotypes", ylim = c(0, 5500), legend.ncols = 3,
     thinData = TRUE, thinData.keep = 0.5)
```
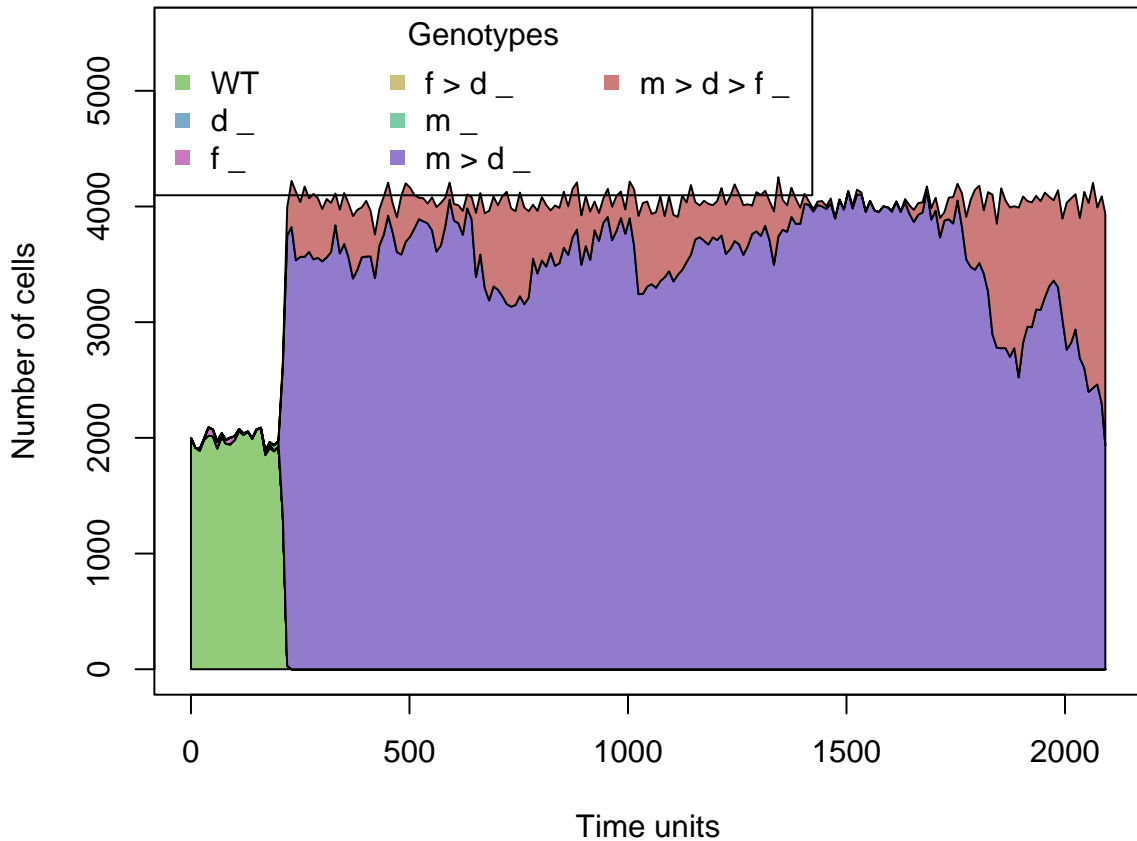

(When reading the figure legends, recall that genotype $x > y \_ z$ is one where a mutation in "x" happened before a mutation in "y", and there is also a mutation in "z" for which order does not matter. Here, there are no genes for which order does not matter and thus there is nothing after the "_").

In this case, the clones with three drivers end up displacing those with two by the time we stop; moreover, notice how those with one driver never really grow to a large population size, so we basically go from a population with clones with zero drivers to a population made of clones with two or three drivers:

```
set.seed(15)
tmp <- oncoSimulIndiv(examplesFitnessEffects[["o3"]],
                      model = "McFL",
                      mu = 5e-5,
                      detectionSize = 1e8,
                      detectionDrivers = 3,
                      sampleEvery = 0.025,
```

188

```
                        max.num.tries = 10,
                        keepEvery = 5,
                        initSize = 2000,
                        finalTime = 20000,
                        onlyCancer = FALSE,
                        extraTime = 1500)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
tmp
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = examplesFitnessEffects[["o3"]], model = "McFL",
##      mu = 5e-05, detectionSize = 1e+08, detectionDrivers = 3,
##      sampleEvery = 0.025, initSize = 2000, keepEvery = 5, extraTime = 1500,
##      finalTime = 20000, onlyCancer = FALSE, max.num.tries = 10)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         7         3984         3984             3              3
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    3                   3      7178  288895
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.01343       6254       6667
##   OccurringDrivers
## 1          d, f, m
##
## Final population composition:
##       Genotype    N
## 1            _    0
## 2          d _    0
## 3      d > m _    0
## 4          f _    0
## 5          m _    0
## 6      m > d _    0
## 7 m > d > f _ 3984
```
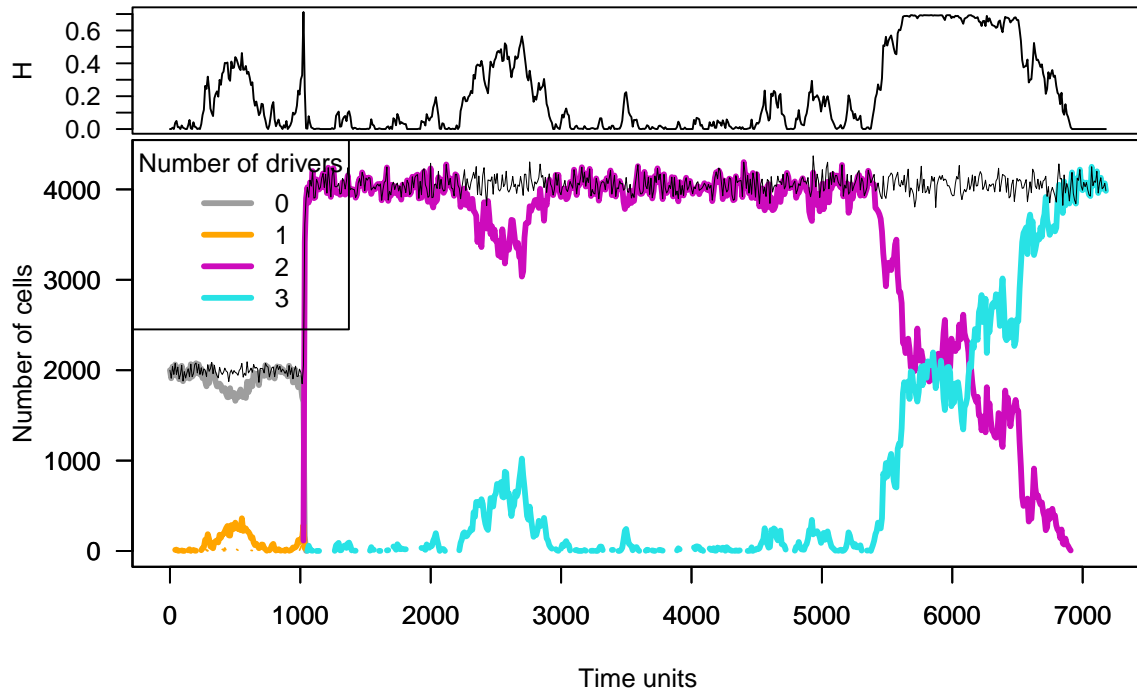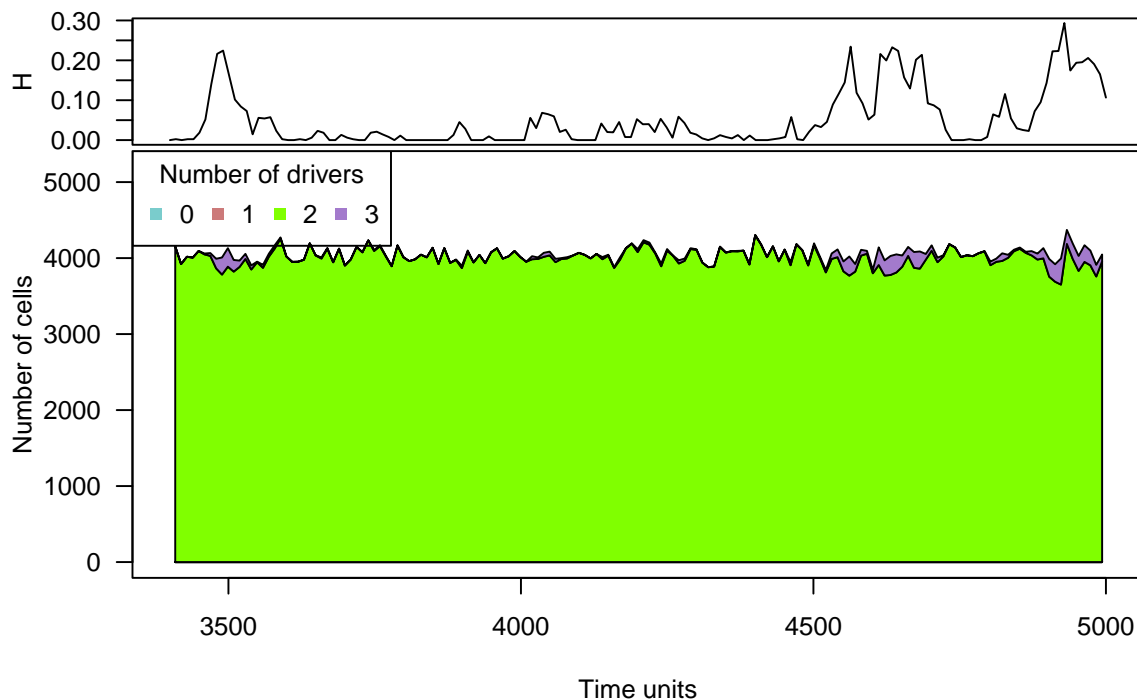
use a drivers plot:

```
par(las = 1, cex = 0.85)
plot(tmp, addtot = TRUE, log = "", plotDiversity = TRUE,
     thinData = TRUE, thinData.keep = 0.5)
```

```
par(las = 1, cex = 0.85)
plot(tmp, type = "stacked", plotDiversity = TRUE,
     legend.ncols = 4, ylim = c(0, 5200), xlim = c(3400, 5000),
     thinData = TRUE, thinData.keep = 0.5)
```


Now show the genotypes explicitly:

```
## Improve telling apart the most abundant
## genotypes by sorting colors
## differently via breakSortColors
```

```
## Modify ncols of legend, so it is legible by not overlapping
## with plot
par(las = 1, cex = 0.85)
plot(tmp, show = "genotypes", breakSortColors = "distave",
     plotDiversity = TRUE, legend.ncols = 4,
     ylim = c(0, 5300), xlim = c(3400, 5000),
     thinData = TRUE, thinData.keep = 0.5)
```
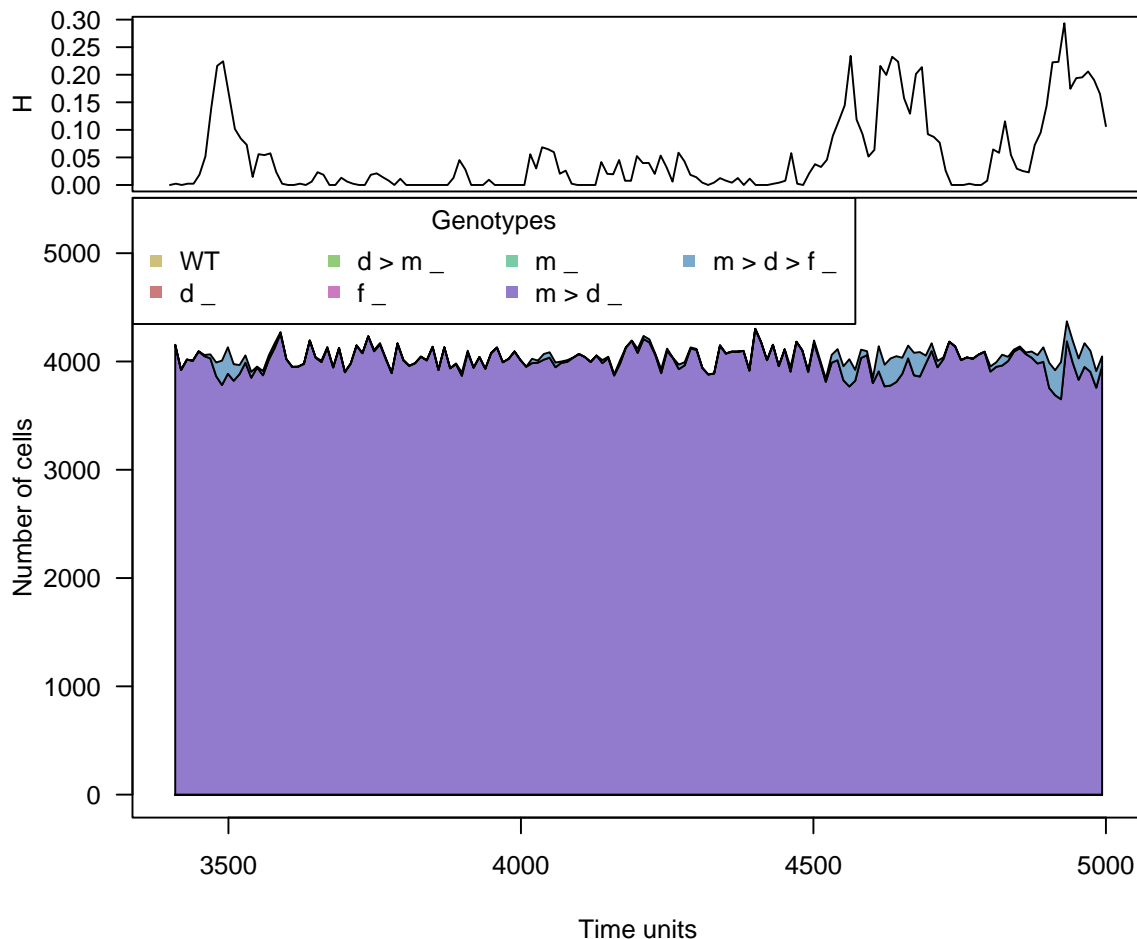

As before, the argument `plotDiversity = TRUE` asks to show a small plot on top with Shannon's diversity index. Here, as before, the quick clonal expansion of the clone with two drivers leads to a sudden drop in diversity (for a while, the population is made virtually of a single clone). Note, however, that compared to section 6.5.3, we are modeling here a scenario with very few genes, and correspondingly very few possible genotypes, and thus it is not strange that we observe very little diversity.

(We have used `extraTime` to continue the simulation well past the point of detection, here specified as three drivers. Instead of specifying `extraTime` we can set the `detectionDrivers` value to a number larger than the number of existing possible drivers, and the simulation will run until `finalTime` if `onlyCancer = FALSE`.)

191

6.6 Interactive graphics

It is possible to create interactive stacked area and stream plots using the *streamgraph* package, available from https://github.com/hrbrmstr/streamgraph. However, that package is not available as a CRAN or BioConductor package, and thus we cannot depend on it for this vignette (or this package). You can, however, paste the code below and make it run locally.

Before calling the `streamgraph` function, though, we need to convert the data from the original format in which it is stored into "long format". A simple convenience function is provided as `OncoSimulWide2Long` in *OncoSimulR*.

As an example, we will use the data we generated above for section 6.5.1.

```
## Convert the data
lb1 <- OncoSimulWide2Long(b1)

## Install the streamgraph package from GitHub and load
library(devtools)
devtools::install_github("hrbrmstr/streamgraph")
library(streamgraph)

## Stream plot for Genotypes
sg_legend(streamgraph(lb1, Genotype, Y, Time, scale = "continuous"),
            show=TRUE, label="Genotype: ")

## Staked area plot and we use the pipe
streamgraph(lb1, Genotype, Y, Time, scale = "continuous",
           offset = "zero") %>%
    sg_legend(show=TRUE, label="Genotype: ")
```

6.7 Multiple initial mutants: starting the simulation from arbitrary configurations

You can specify the population composition when you start the simulation: in other words, you can use multiple initial mutants. Simply pass a vector to `initMutant` and a vector of the same length to `initSize`: the first are the genotypes/clones, the second the population sizes of the corresponding genotypes/clones.

(It often makes no sense to start the simulation with genotypes with birth rate of 0: you can try it, but you will be told about it.)

Two examples.

```
r2 <- rfitness(6)
## Make sure these always viable for interesting stuff
r2[2, 7] <- 1 + runif(1) # A
r2[4, 7] <- 1 + runif(1) # C
r2[8, 7] <- 1 + runif(1) # A, B
```

```
o2 <- allFitnessEffects(genotFitness = r2)
ag <- evalAllGenotypes(o2)

out1 <- oncoSimulIndiv(o2, initMutant = c("A", "C"),
                        initSize = c(100, 200),
                        onlyCancer = FALSE,
                        finalTime = 200)
```

No WT, nor any other genotypes with a single mutation (except "A" and "C") would thus be possible either (it is impossible to obtain, say, a "B" if there are no WT).

We can do something similar with the frequency-dependent functionality (section 10):

```
gffd0 <- data.frame(
    Genotype = c(
        "A", "A, B",
        "C", "C, D", "C, E"),
    Fitness = c(
        "1.3",
        "1.4",
        "1.4",
        "1.1 + 0.7*((f_A + f_A_B) > 0.3)",
        "1.2 + sqrt(f_A + f_C + f_C_D)"))

afd0 <- allFitnessEffects(genotFitness = gffd0,
                           frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = gffd0,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
```

```
sp <- 1:5
names(sp) <- c("A", "C", "A, B", "C, D", "C, E")
eag0 <- evalAllGenotypes(afd0, spPopSizes = sp)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
os0 <- oncoSimulIndiv(afd0,
                       initMutant = c("A", "C"),
                       finalTime = 20, initSize = c(1e4, 1e5),
                       onlyCancer = FALSE, model = "McFLD")
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
```

6.8 Multispecies simulations

Since we can use arbitrary initial populations to start the simulation (section 6.7) and we can use arbitrary fitness specifications, you can run multi-species simulations using a simple trick.

Suppose you want to use a two species simulation, where the first species has two loci and the second three loci. This is a possible procedure:

- Create a genotype of total length $(1 + 2) + (1 + 3)$.
- The first locus in each set we will use as the "species indicator".
- Thus, Species A will use loci 1, 2, 3 and Species B loci 4 to 7.
- Make all genotype combinations where loci from different species are mutated lethal (e.g., a genotype with loci 2 and 5 mutated is not viable).
- Now you can start the simulation from the "WT of each species": the initMutants will have genotypes with only loci 1 or loci 4 mutated.

This trick is really only an approximation: mutation to the other species is actually death. So there is "leakage" from mutation to death, as in example 10.4: both species are leaking a small number of children via mutation to non-viable "hybrids". Factor this into your equations for death rate, but this should be negligible if death rate $\gg$ mutation rate. (You can ameliorate this problem slightly by making mutation to the "species indicator" locus very small, say 10^{-10} —do not set it to 0, as you will get an error).

Of course, you can extend the scheme above to arbitrary numbers of species.

Let's give several examples.

We use a capital letter for the "species indicator locus" and name each of the species-specific loci with the lower case and a number. We then ameliorate the leakage issue by making mutation to "A" or "B" tiny (though there is still leakage from, say, "A" to "A, b1").

```
mspec <- data.frame(
    Genotype = c("A",
                "A, a1", "A, a2", "A, a1, a2",
                "B",
                "B, b1", "B, b2", "B, b3",
                "B, b1, b2", "B, b1, b3", "B, b1, b2, b3"),
    Fitness = 1 + runif(11)
)
fmspec <- allFitnessEffects(genotFitness = mspec)
## Column names of object not Genotype and Birth Renaming them assuming that is w
## Warning in allGenotypes_to_matrix(x, frequencyDependentBirth,
## frequencyDependentDeath, : No WT genotype. Setting its birth to 1.
```

```
afmspec <- evalAllGenotypes(fmspec)


## Show only viable ones
afmspec[afmspec$Fitness >= 1, ]
## [1] Genotype Birth
## <0 rows> (or 0-length row.names)
```

```
muv <- c(1e-10, rep(1e-5, 2), 1e-10, rep(1e-5, 3))
names(muv) <- c("A", paste0("a", 1:2), "B", paste0("b", 1:3))

out1 <- oncoSimulIndiv(fmspec, initMutant = c("A", "B"),
                       initSize = c(100, 200),
                       mu = muv,
                       onlyCancer = FALSE,
                       finalTime = 200)
```

We can do something similar with the frequency-dependent-fitness functionality. (We use a somewhat silly specification, so that checking equations is easy)

```
mspecF <- data.frame(
    Genotype = c("A",
                 "A, a1", "A, a2", "A, a1, a2",
                 "B",
                 "B, b1", "B, b2", "B, b3",
                 "B, b1, b2", "B, b1, b3", "B, b1, b2, b3"),
    Fitness = c("1 + f_A_a1",
                "1 + f_A_a2",
                "1 + f_A_a1_a2",
                "1 + f_B",
                "1 + f_B_b1",
                "1 + f_B_b2",
                "1 + f_B_b3",
                "1 + f_B_b1_b2",
                "1 + f_B_b1_b3",
                "1 + f_B_b1_b2_b3",
                "1 + f_A")
)
fmspecF <- allFitnessEffects(genotFitness = mspecF,
                             frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = mspecF,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'

## Remeber, spPopSizes correspond to the genotypes
## shown in
```

```
fmspecF$full_FDF_spec
##      A a1 a2 B b1 b2 b3 Genotype_as_numbers Genotype_as_letters
## 1  1  0  0 0  0  0  0                     1                   A
## 2  0  0  0 1  0  0  0                     4                   B
## 3  1  1  0 0  0  0  0                  1, 2               A, a1
## 4  1  0  1 0  0  0  0                  1, 3               A, a2
## 5  0  0  0 1  1  0  0                  4, 5               B, b1
## 6  0  0  0 1  0  1  0                  4, 6               B, b2
## 7  0  0  0 1  0  0  1                  4, 7               B, b3
## 8  1  1  1 0  0  0  0               1, 2, 3           A, a1, a2
## 9  0  0  0 1  1  1  0               4, 5, 6           B, b1, b2
## 10 0  0  0 1  1  0  1               4, 5, 7           B, b1, b3
## 11 0  0  0 1  1  1  1            4, 5, 6, 7       B, b1, b2, b3
##     Genotype_as_fvarsb Fitness_as_fvars Fitness_as_letters
## 1                  f_1          1 + f_1_2          1 + f_A_a1
## 2                  f_4          1 + f_4_5          1 + f_B_b1
## 3                f_1_2          1 + f_1_3          1 + f_A_a2
## 4                f_1_3        1 + f_1_2_3       1 + f_A_a1_a2
## 5                f_4_5          1 + f_4_6          1 + f_B_b2
## 6                f_4_6          1 + f_4_7          1 + f_B_b3
## 7                f_4_7        1 + f_4_5_6       1 + f_B_b1_b2
## 8              f_1_2_3            1 + f_4             1 + f_B
## 9              f_4_5_6        1 + f_4_5_7       1 + f_B_b1_b3
## 10             f_4_5_7      1 + f_4_5_6_7    1 + f_B_b1_b2_b3
## 11           f_4_5_6_7            1 + f_1             1 + f_A
```

```
## in exactly that order if it is unnamed.

afmspecF <- evalAllGenotypes(fmspecF,
                             spPopSizes = 1:11)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
## Alternatively, pass a named vector, which is the recommended approach

spp <- 1:11
names(spp) <- c("A","B",
                "A, a1", "A, a2",
                "B, b1", "B, b2", "B, b3",
                "A, a1, a2",
                "B, b1, b2", "B, b1, b3", "B, b1, b2, b3")
```

```
afmspecF <- evalAllGenotypes(fmspecF,
                             spPopSizes = spp)
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.


## Show only viable ones
afmspecF[afmspecF$Fitness >= 1, ]
##        Genotype Fitness
## 2             A   1.045
## 5             B   1.076
## 9         A, a1   1.061
## 10        A, a2   1.121
## 24        B, b1   1.091
## 25        B, b2   1.106
## 26        B, b3   1.136
## 30    A, a1, a2   1.030
## 61    B, b1, b2   1.152
## 62    B, b1, b3   1.167
## 99 B, b1, b2, b3   1.015


## Expected values of fitness
exv <- 1 + c(3, 5, 4, 8, 6, 7, 9, 2, 10, 11, 1)/sum(1:11)
stopifnot(isTRUE(all.equal(exv, afmspecF[afmspecF$Fitness >= 1, ]$Fitness)))

muv <- c(1e-10, rep(1e-5, 2), 1e-10, rep(1e-5, 3))
names(muv) <- c("A", paste0("a", 1:2), "B", paste0("b", 1:3))

out1 <- oncoSimulIndiv(fmspecF, initMutant = c("A", "B"),
                       initSize = c(1e4, 1e5),
                       mu = muv,
                       finalTime = 20,
                       model = "McFLD",
                       onlyCancer = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Init Mutant with birth == 1.0
## Init Mutant with birth == 1.0
```

Some further examples are given below, as in 10.4.2.

7 Sampling multiple simulations

Often, you will want to simulate multiple runs of the same scenario, and then obtain the matrix of runs by mutations (a matrix of individuals/samples by genes or, equivalently, a vector of "genotypes"), and do something with them. OncoSimulR offers several ways of doing this.

The key function here is `samplePop`, either called explicitly after `oncoSimulPop` (or `oncoSimulIndiv`), or implicitly as part of a call to `oncoSimulSample`. With `samplePop` you can use **single cell** or **whole tumor** sampling (for details see the help of `samplePop`). Depending on how the simulations were conducted, you might also sample at different times, or as a function of population sizes. A major difference between procedures has to do with whether or not you want to keep the complete history of the simulations.

You want to keep the complete history of population sizes of clones during the simulations. You will simulate using:

- `oncoSimulIndiv` repeatedly (maybe within `mclapply`, to parallelize the run).

- `oncoSimulPop`. `oncoSimulPop` is basically a thin wrapper around `oncoSimulIndiv` that uses `mclapply`.

In both cases, you specify the conditions for ending the simulations (as explained in 6.3). Then, you use function `samplePop` to obtain the matrix of samples by mutations.

You do not want to keep the complete history of population sizes of clones during the simulations. You will simulate using:

- `oncoSimulIndiv` repeatedly, with argument `keepEvery = NA`.

- `oncoSimulPop`, with argument `keepEvery = NA`.

  In both cases you specify the conditions for ending the simulations (as explained in 6.3). Then, you use function `samplePop`.

- `oncoSimulSample`, specifying the conditions for ending the simulations (as explained in 6.3). In this case, you will not use `samplePop`, as that is implicitly called by `oncoSimulSample`. The output is directly the matrix (and a little bit of summary from each run), and during the simulation it only stores one time point.

Why the difference between the above cases? If you keep the complete history of population sizes, you can take samples at any of the times between the beginning and the end of the simulations. If you do not keep the history, you can only sample at the time the simulation exited (see section 18.2). Why would you want to use the second route? If we are only interested in the final matrix of individuals by mutations, keeping the complete history above is wasteful because we store fully all of the simulations (for example in the call to `oncoSimulPop`) and then sample (in the call to `samplePop`). Further criteria to use when choosing between sampling procedures is whether you need `detectionSize` and `detectionDrivers` do differ

between simulations: if you use `oncoSimulPop` the arguments for `detectionSize` and `detectionDrivers` must be the same for all simulations but this is not the case for `oncoSimulSample`. See further comments in 7.2. Finally, parallelized execution is available for `oncoSimulPop` but, by design, not for `oncoSimulSample`.

The following are a few examples. First we run `oncoSimulPop` to obtain 4 simulations and in the last line we sample from them:

```
pancrPop <- oncoSimulPop(4, pancr,
                         onlyCancer = TRUE,
                         detectionSize = 1e7,
                         keepEvery = 10,
                         mc.cores = 2)
```

```
summary(pancrPop)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1        11     10512126     10460815             0              0
## 2        14     10565402      7416572             0              0
## 3        11     10838249     10791939             0              0
## 4        14     11021534     10893508             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0      1269    1975
## 2                    0                   0       462    1014
## 3                    0                   0      2047    2784
## 4                    0                   0      1848    2605
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA     142857     142857
## 2          FALSE          FALSE      NA     142857     142857
## 3          FALSE          FALSE      NA     142857     142857
## 4          FALSE          FALSE      NA     142857     142857
##   OccurringDrivers
## 1
## 2
## 3
## 4
```

```
samplePop(pancrPop)
##
##  Subjects by Genes matrix of 4 subjects and 7 genes.
##      CDNK2A KRAS MLL3 PXDN SMAD4 TGFBR2 TP53
## [1,]      0    1    0    0     0      0    0
## [2,]      0    1    0    0     0      0    0
## [3,]      0    1    0    0     0      0    0
## [4,]      0    1    0    0     0      0    0
```

Now a simple multiple call to `oncoSimulIndiv` wrapped inside `mclapply`; this is basically the same we just did above. We set the class of the object to allow direct

usage of `samplePop`. (Note: in Windows `mc.cores > 1` is not supported, so for the vignette to run in Windows, Linux, and Mac we explicitly set it here in the call to `mclapply`. For regular usage, you will not need to do this; just use whatever is appropriate for your operating system and number of cores. As well, we do not need any of this with `oncoSimulPop` because the code inside `oncoSimulPop` already takes care of setting `mc.cores` to 1 in Windows).

```r
library(parallel)

if(.Platform$OS.type == "windows") {
    mc.cores <- 1
} else {
    mc.cores <- 2
}

p2 <- mclapply(1:4, function(x) oncoSimulIndiv(pancr,
                                               onlyCancer = TRUE,
                                               detectionSize = 1e7,
                                               keepEvery = 10),
               mc.cores = mc.cores)
class(p2) <- "oncosimulpop"
samplePop(p2)
##
##  Subjects by Genes matrix of 4 subjects and 7 genes.
##      CDNK2A KRAS MLL3 PXDN SMAD4 TGFBR2 TP53
## [1,]      0    1    0    0     0      0    0
## [2,]      0    1    0    0     0      0    0
## [3,]      0    1    0    0     0      0    0
## [4,]      0    1    0    0     0      0    0
```

Above, we have kept the complete history of the simulations as you can check by doing, for instance

```r
tail(pancrPop[[1]]$pops.by.time)
##          [,1] [,2] [,3] [,4]      [,5] [,6] [,7]  [,8] [,9] [,10]
## [123,] 1220 2088    0    0     77052    3    0     0    0     0
## [124,] 1230 2070    0    0    211703    4    0     0    0     0
## [125,] 1240 2570    0    0    580354   27    0    47    2    17
## [126,] 1250 2521   16    0   1566900   73    0   475    4   222
## [127,] 1260 2638  220    0   4254588  794    4  4777    2  1545
## [128,] 1269 2760 1726   18 10460815 5401   10 30651   14 10731
##          [,11] [,12]
## [123,]      0     0
## [124,]      0     0
## [125,]      0     0
## [126,]      0     0
## [127,]      0     0
```

```
## [128,]      0      0
```

If we were not interested in the complete history of simulations we could have done
instead (note the argument `keepEvery = NA`)

```
pancrPopNH <- oncoSimulPop(4, pancr,
                           onlyCancer = TRUE,
                           detectionSize = 1e7,
                           keepEvery = NA,
                           mc.cores = 2)
```

```
summary(pancrPopNH)
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         9     10317397     10291675             0              0
## 2        13     10528981      7009735             0              0
## 3         9     10407445     10243670             0              0
## 4         8     10422244     10397392             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0      2220    2988
## 2                    0                   0      1815    2410
## 3                    0                   0      1948    2688
## 4                    0                   0      1467    2148
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA     142857     142857
## 2          FALSE          FALSE      NA     142857     142857
## 3          FALSE          FALSE      NA     142857     142857
## 4          FALSE          FALSE      NA     142857     142857
##   OccurringDrivers
## 1
## 2
## 3
## 4
```

```
samplePop(pancrPopNH)
##
##  Subjects by Genes matrix of 4 subjects and 7 genes.
##      CDNK2A KRAS MLL3 PXDN SMAD4 TGFBR2 TP53
## [1,]      0    1    0    0     0      0    0
## [2,]      0    1    0    0     0      0    0
## [3,]      0    1    0    0     0      0    0
## [4,]      0    1    0    0     0      0    0
```

which only keeps the very last sample:

```
pancrPopNH[[1]]$pops.by.time
##      [,1]  [,2] [,3]     [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] 2220 14751 6772 10291675 2814    7  825   12  540     1
```

Or we could have used `oncoSimulSample`:

```
pancrSamp <- oncoSimulSample(4, pancr, onlyCancer = TRUE)
## Successfully sampled 4 individuals
##
##  Subjects by Genes matrix of 4 subjects and 7 genes.
```

```
pancrSamp$popSamp
##       CDNK2A KRAS MLL3 PXDN SMAD4 TGFBR2 TP53
## [1,]      0    1    0    0     1      0    0
## [2,]      0    1    0    0     0      0    0
## [3,]      0    1    0    0     0      0    0
## [4,]      0    1    0    0     0      0    0
```

Again, why the above differences? If we are only interested in the final matrix of populations by mutations, keeping the complete history the above is wasteful, because we store fully all of the simulations (in the call to `oncoSimulPop`) and then sample (in the call to `samplePop`).

7.1 Whole-tumor and single-cell sampling, and do we always want to sample?

`samplePop` is designed to emulate the process of obtaining a sample from a (set of) "patient(s)". But there is no need to sample. The history of the population, with a granularity that is controlled by argument `keepEvery`, is kept in the matrix `pops.by.time` which contains the number of cells of every clone at every sampling point (see further details in 18.2). This is the information used in the plots that show the trajectory of a simulation: the plots that show the change in genotype or driver abundance over time (see section 6.4 and examples mentioned there).

Regardless of whether and how you plot the information in `pops.by.time`, you can also sample one or multiple simulations using `samplePop`. In **whole-tumor** sampling the resolution is the whole tumor (or the whole population). Thus, a key argument is `thresholdWhole`, the threshold for detecting a mutation: a gene is considered mutated if it is altered in at least "thresholdWhole" proportion of the cells in that simulation (at a particular time point). This of course means that your "sampled genotype" might not correspond to any existing genotype because we are summing over all cells in the population. For instance, suppose that at the time we take the sample there are only two clones in the population, one clone with a frequency of 0.4 that has gene A mutated, and a second clone one with a frequency of 0.6 that has gene B mutated. If you set `thresholdWhole` to values ≤ 0.4 the sampled genotype will show both A and B mutated. **Single-cell** sampling is provided as an option in contrast to whole-tumor sampling. Here any sampled genotype will correspond to an existing genotype as you are sampling with single-cell resolution.

When `samplePop` is run on a set of simulated data of, say, 100 simulated trajectories (100 "subjects"), it will produce a matrix with 100 rows (100 "subjects"). But if it makes sense in the context of your problem (e.g., multiple samples per patient?) you

can of course run `samplePop` repeatedly.

7.2 Differences between "samplePop" and "oncoSimulSample"

`samplePop` provides two sampling times: "last" and "uniform". It also allows you to sample at the first sample time(s) at which the population(s) reaches a given size, which can be either the same or different for each simulation (with argument `popSizeSample`). "last" means to sample each individual in the very last time period of the simulation. "uniform" means sampling each individual at a time chosen uniformly from all the times recorded in the simulation between the time when the first driver appeared and the final time period. "unif" means that it is almost sure that different individuals will be sampled at different times. "last" does not guarantee that different individuals will be sampled at the same time unit, only that all will be sampled in the last time unit of their simulation.

With `oncoSimulSample` we obtain samples that correspond to `timeSample = "last"` in `samplePop` by specifying a unique value for `detectionSize` and `detectionDrivers`. The data from each simulation will correspond to the time point at which those are reached (analogous to `timeSample = "last"`). How about uniform sampling? We pass a vector of `detectionSize` and `detectionDrivers`, where each value of the vector comes from a uniform distribution. This is not identical to the "uniform" sampling of `oncoSimulSample`, as we are not sampling uniformly over all time periods, but are stopping at uniformly distributed values over the stopping conditions. Arguably, however, the procedure in `samplePop` might be closer to what we mean with "uniformly sampled over the course of the disease" if that course is measured in terms of drivers or size of tumor.

An advantage of `oncoSimulSample` is that we can specify arbitrary sampling schemes, just by passing the appropriate vector `detectionSize` and `detectionDrivers`. A disadvantage is that if we change the stopping conditions we can not just resample the data, but we need to run it again.

There is no difference between `oncoSimulSample` and `oncoSimulPop` + `samplePop` in terms of the `typeSample` argument (whole tumor or single cell).

Finally, there are some additional differences between the two functions. `oncoSimulPop` can run parallelized (it uses `mclapply`). This is not done with `oncoSimulSample` because this function is designed for simulation experiments where you want to examine many different scenarios simultaneously. Thus, we provide additional stopping criteria (`max.wall.time.total` and `max.num.tries.total`) to determine whether to continue running the simulations, that bounds the total running time of all the simulations in a call to `oncoSimulSample`. And, if you are running multiple different scenarios, you might want to make multiple, separate, independent calls (e.g., from different R processes) to `oncoSimulSample`, instead of relying in `mclapply`, since this is likely to lead to better usage of multiple cores/CPUs if you are examining a large number of different scenarios.

8 Showing the genealogical relationships of clones

If you run simulations with `keepPhylog = TRUE`, the simulations keep track of when every clone is generated, and that will allow us to see the parent-child relationships between clones. (This is disabled by default).

Let us re-run a previous example:

```
set.seed(15)
tmp <- oncoSimulIndiv(examplesFitnessEffects[["o3"]],
                      model = "McFL",
                      mu = 5e-5,
                      detectionSize = 1e8,
                      detectionDrivers = 3,
                      sampleEvery = 0.025,
                      max.num.tries = 10,
                      keepEvery = 5,
                      initSize = 2000,
                      finalTime = 20000,
                      onlyCancer = FALSE,
                      extraTime = 1500,
                      keepPhylog = TRUE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
tmp
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = examplesFitnessEffects[["o3"]], model = "McFL",
##     mu = 5e-05, detectionSize = 1e+08, detectionDrivers = 3,
##     sampleEvery = 0.025, initSize = 2000, keepEvery = 5, extraTime = 1500,
##     finalTime = 20000, onlyCancer = FALSE, keepPhylog = TRUE,
##     max.num.tries = 10)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         7         3984         3984             3              3
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    3                   3      7178  288895
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.01343       6254       6667
##   OccurringDrivers
## 1          d, f, m
##
## Final population composition:
##       Genotype    N
```

```
## 1              _       0
## 2          d _       0
## 3      d > m _       0
## 4          f _       0
## 5          m _       0
## 6      m > d _       0
## 7 m > d > f _    3984
```

We can plot the parent-child relationships[14] of every clone ever created (with fitness larger than 0 —clones without viability are never shown):

```
plotClonePhylog(tmp, N = 0)
```
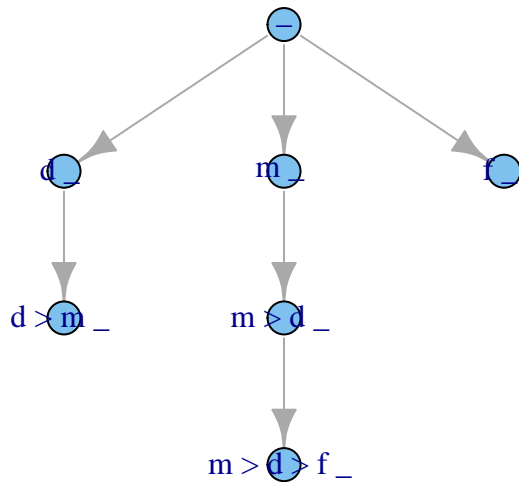

However, we often only want to show clones that exist (have number of cells > 0) at a certain time (while of course showing all of their ancestors, even if those are now extinct —i.e., regardless of their current numbers).

```
plotClonePhylog(tmp, N = 1)
```

---

[14]There are several packages in R devoted to phylogenetic inference and related issues. For instance, *ape*. I have not used that infrastructure because of our very specific needs and circumstances; for instance, internal nodes are observed, we can have networks instead of trees, and we have no uncertainty about when events occurred.
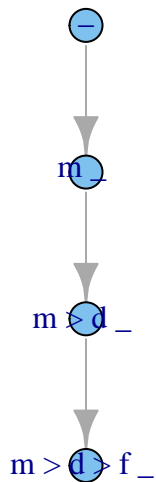
If we set `keepEvents = TRUE` the arrows show how many times each clone appeared: (The next can take a while)

```
plotClonePhylog(tmp, N = 1, keepEvents = TRUE)
```
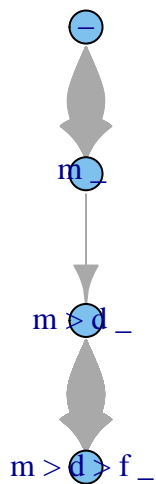

And we can show the plot so that the vertical axis is proportional to time (though you might see overlap of nodes if a child node appeared shortly after the parent):

```
plotClonePhylog(tmp, N = 1, timeEvents = TRUE)
```

We can obtain the adjacency matrix doing

```
get.adjacency(plotClonePhylog(tmp, N = 1, returnGraph = TRUE))
## Warning: `get.adjacency()` was deprecated in igraph 2.0.0.
## i Please use `as_adjacency_matrix()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was gene
## 4 x 4 sparse Matrix of class "dgCMatrix"
##                 _   m _   m > d _   m > d > f _
## _               .   1       .             .
## m _             .   .       1             .
## m > d _         .   .       .             1
## m > d > f _     .   .       .             .
```

We can see another example here:

```
set.seed(456)
mcf1s <-  oncoSimulIndiv(mcf1,
                         model = "McFL",
                         mu = 1e-7,
                         detectionSize = 1e8,
                         detectionDrivers = 100,
                         sampleEvery = 0.025,
                         keepEvery = 2,
                         initSize = 2000,
                         finalTime = 1000,
                         onlyCancer = FALSE,
                         keepPhylog = TRUE)
```

Showing only clones that exist at the end of the simulation (and all their parents):

```
plotClonePhylog(mcf1s, N = 1)
```
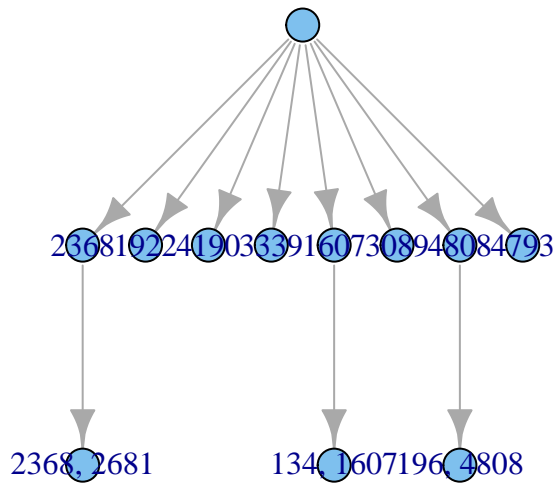

Notice that the labels here do not have a "_", since there were no order effects in fitness. However, the labels show the genes that are mutated, just as before.
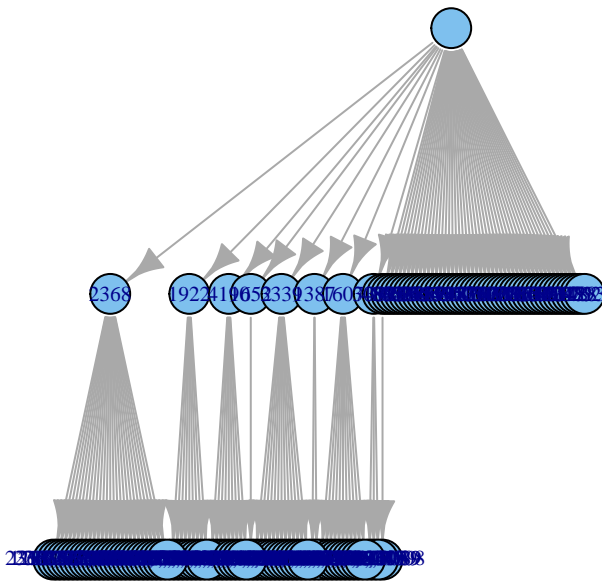
Similar, but with vertical axis proportional to time:

```
par(cex = 0.7)
plotClonePhylog(mcf1s, N = 1, timeEvents = TRUE)
```
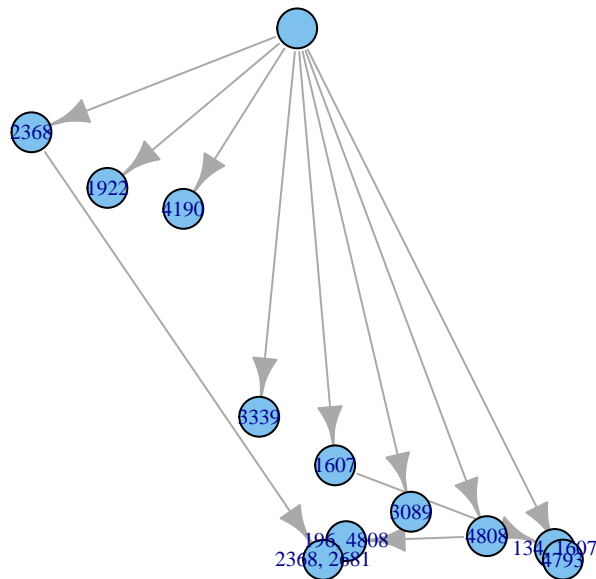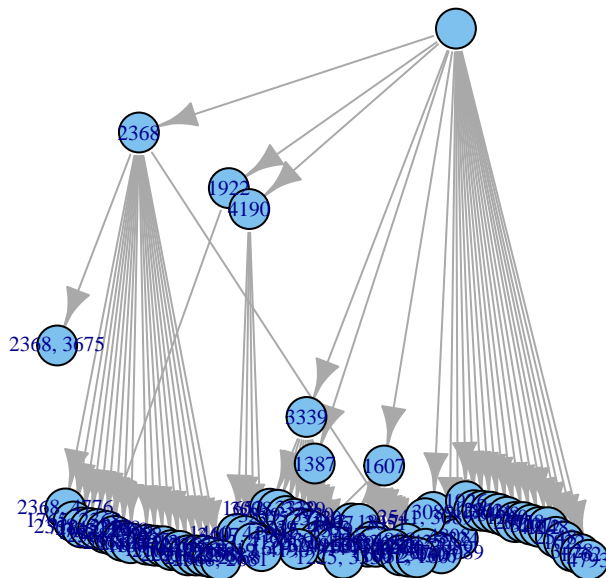

What about those that existed in the last 200 time units?

```
par(cex = 0.7)
plotClonePhylog(mcf1s, N = 1, t = c(800, 1000))
```

208

And try now to show also when the clones appeared (we restrict the time to between 900 and 1000, to avoid too much clutter):

```
par(cex = 0.7)
plotClonePhylog(mcf1s, N = 1, t = c(900, 1000), timeEvents = TRUE)
```
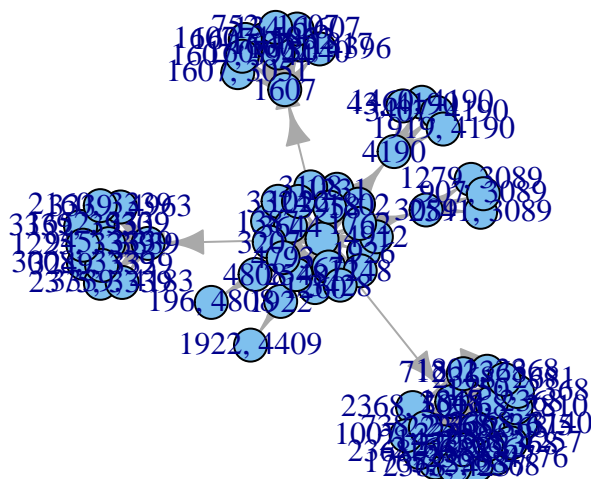

(By playing with t, it should be possible to obtain animations of the phylogeny. We will not pursue it here.)

If the previous graph seems cluttered, we can represent it in a different way by calling *igraph* directly after storing the graph and using the default layout:

```
g1 <- plotClonePhylog(mcf1s, N = 1, t = c(900, 1000),
                      returnGraph = TRUE)
```

```
plot(g1)
```

which might be easier to show complex relationships or identify central or key clones.

It is of course quite possible that, especially if we consider few genes, the parent-child relationships will form a network, not a tree, as the same child node can have multiple parents. You can play with this example, modified from one we saw before (section 3.4.6):

```
op <- par(ask = TRUE)
while(TRUE) {
    tmp <- oncoSimulIndiv(smn1, model = "McFL",
                          mu = 5e-5, finalTime = 500,
                          detectionDrivers = 3,
                          onlyCancer = FALSE,
                          initSize = 1000, keepPhylog = TRUE)
    plotClonePhylog(tmp, N = 0)
}
par(op)
```

8.1   Parent-child relationships from multiple runs

If you use oncoSimulPop you can store and plot the "phylogenies" of the different runs:

```
oi <- allFitnessEffects(orderEffects =
              c("F > D" = -0.3, "D > F" = 0.4),
              noIntGenes = rexp(5, 10),
                        geneToModule =
                              c("F" = "f1, f2, f3",
                                "D" = "d1, d2") )
oiI1 <- oncoSimulIndiv(oi, model = "Exp", onlyCancer = TRUE)
oiP1 <- oncoSimulPop(4, oi,
                    keepEvery = 10,
                    mc.cores = 2,
                    keepPhylog = TRUE, onlyCancer = TRUE)
```

We will plot the first two:

```
op <- par(mar = rep(0, 4), mfrow = c(2, 1))
plotClonePhylog(oiP1[[1]])
plotClonePhylog(oiP1[[2]])
```
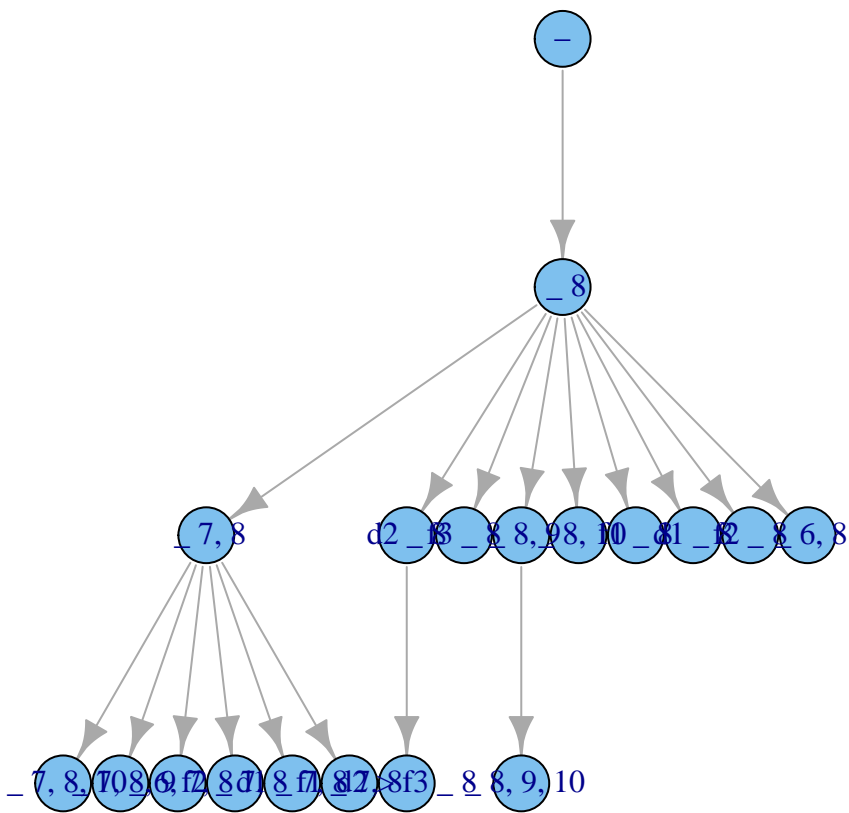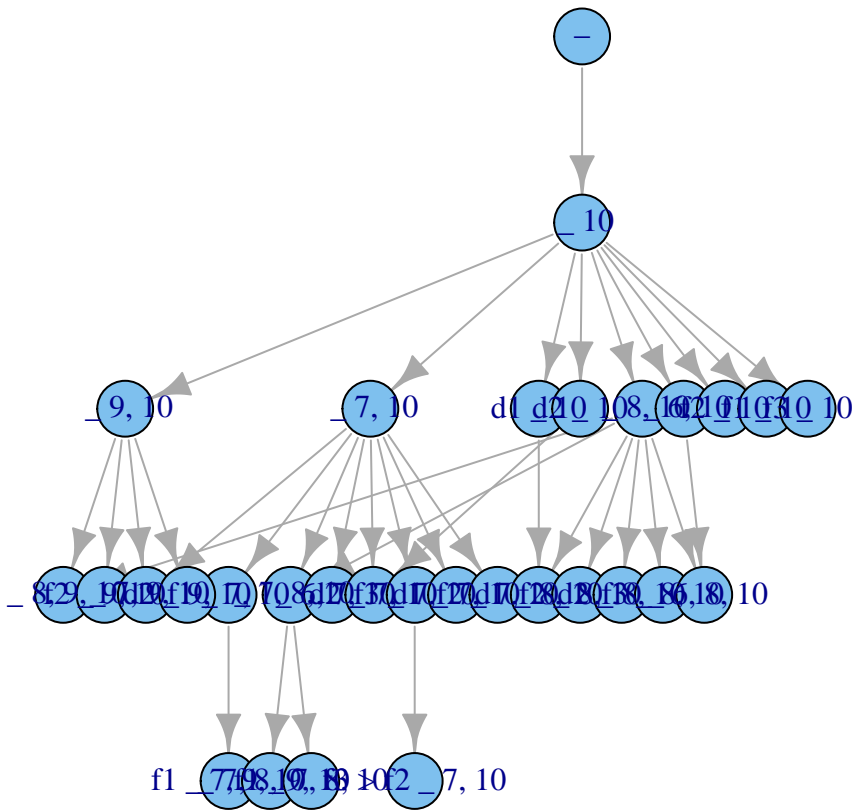
`par`(op)

This is so far disabled in function `oncoSimulSample`, since that function is optimized for other uses. This might change in the future.

9 Generating random fitness landscapes

9.1 Random fitness landscapes from a Rough Mount Fuji model

In most of the examples seen above, we have fully specified the fitness of the different genotypes (either by providing directly the full mapping genotypes to fitness, or by providing that mapping by specifying the effects of the different gene combinations). In some cases, however, we might want to specify a particular model that generates the fitness landscape, and then have fitnesses be random variables obtained under this model. In other words, in this random fitness landscape the fitness of the genotypes is a random variable generated under some specific model. Random fitness landscapes are used extensively, for instance, to understand the evolutionary consequences of different types of epistatic interactions (Franke et al., 2011; e.g., Szendro, Schenk, et al., 2013) and there are especially developed tools for plotting and analyzing random fitness landscapes (e.g., Brouillet et al., 2015).

With OncoSimulR it is possible to generate mappings of genotype to fitness using the function `rfitness` that allows you to use from a pure House of Cards model to a purely additive model (see 9.2 for NK model). I have followed Szendro, Schenk, et al. (2013) and Franke et al. (2011) and model fitness as

$$f_i = -c \, d(i, reference) + x_i \tag{2}$$

where $d(i, j)$ is the Hamming distance between genotypes i and j (the number of positions that differ), c is the decrease in fitness of a genotype per each unit increase in Hamming distance from the reference genotype, and x_i is a random variable (in this case, a normal deviate of mean 0 and standard deviation sd). You can change the reference genotype to any of the genotypes: for the deterministic part, you make the fittest genotype be the one with all positions mutated by setting `reference = "max"`, or use the wildtype by using a string of 0s, or randomly select a genotype as a reference by using `reference = "random"` or `reference = "random2"`. And by changing c and sd you can flexibly modify the relative weight of the purely House of Cards vs. additive component. The expression used above is also very similar to the one on Greene & Crona (2014) if you use `rfitness` with the argument `reference = "max"`.

What can you do with these genotype to fitness mappings? You could plot them, you could use them as input for `oncoSimulIndiv` and related functions, or you could export them (`to_Magellan`) and plot them externally (e.g., in MAGELLAN: http://wwwabi.snv.jussieu.fr/public/Magellan/, Brouillet et al. (2015)).

```
## A small example
rfitness(3)
##      A B C  Birth
## [1,] 0 0 0 1.0000
## [2,] 1 0 0 1.3485
## [3,] 0 1 0 1.0170
```

```
## [4,] 0 0 1 1.0572
## [5,] 1 1 0 2.3799
## [6,] 1 0 1 0.9260
## [7,] 0 1 1 2.1455
## [8,] 1 1 1 0.9875
## attr(,"class")
## [1] "matrix"                "array"
## [3] "genotype_fitness_matrix"
```

```
## A 5-gene example, where the reference genotype is the
## one with all positions mutated, similar to Greene and Crona,
## 2014.  We will plot the landscape and use it for simulations
## We downplay the random component with a sd = 0.5

r1 <- rfitness(5, reference = rep(1, 5), sd = 0.6)
plot(r1)
```
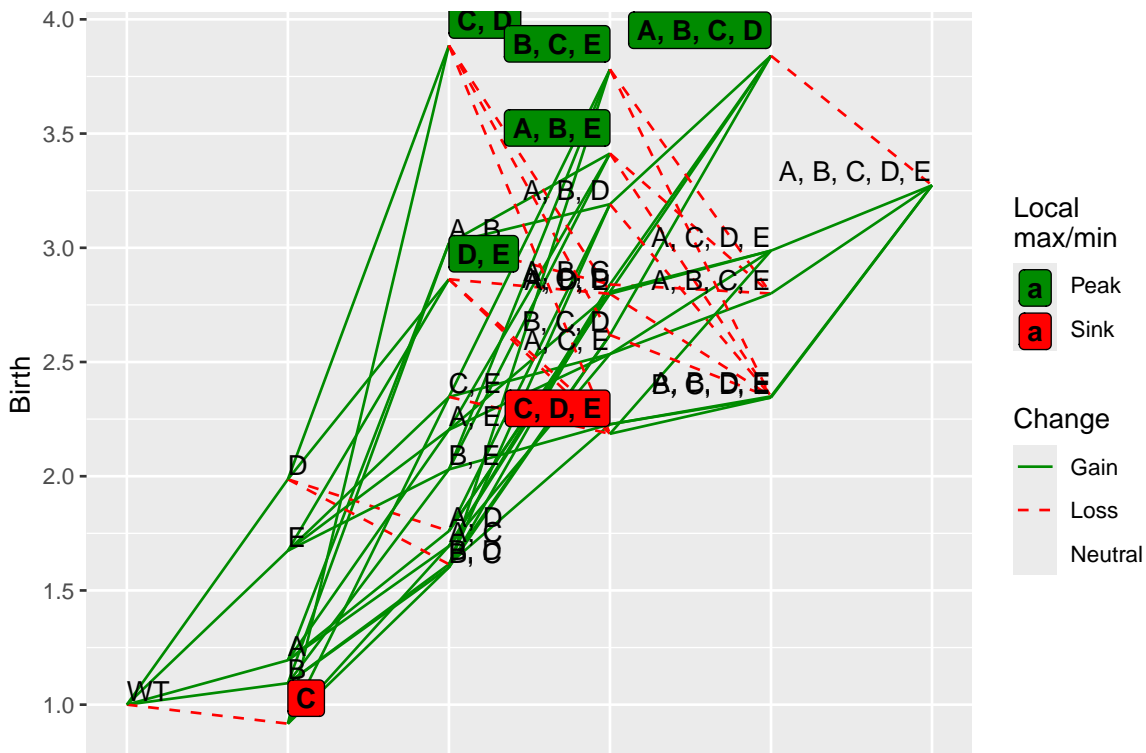


```
oncoSimulIndiv(allFitnessEffects(genotFitness = r1),
            onlyCancer = TRUE)
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = allFitnessEffects(genotFitness = r1), onlyCancer = TRUE)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1        10    212432090    154439733             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
```

```
## 1                      0                    0      333      1394
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA      85791      2e+05
##   OccurringDrivers
## 1
##
## Final population composition:
##    Genotype          N
## 1                  2070
## 2   A, C, D        473
## 3      A, D        240
## 4         B          0
## 5   B, C, D        553
## 6      B, D        238
## 7      C, D 154439733
## 8   C, D, E        326
## 9         D  57856440
## 10     D, E    132017
```
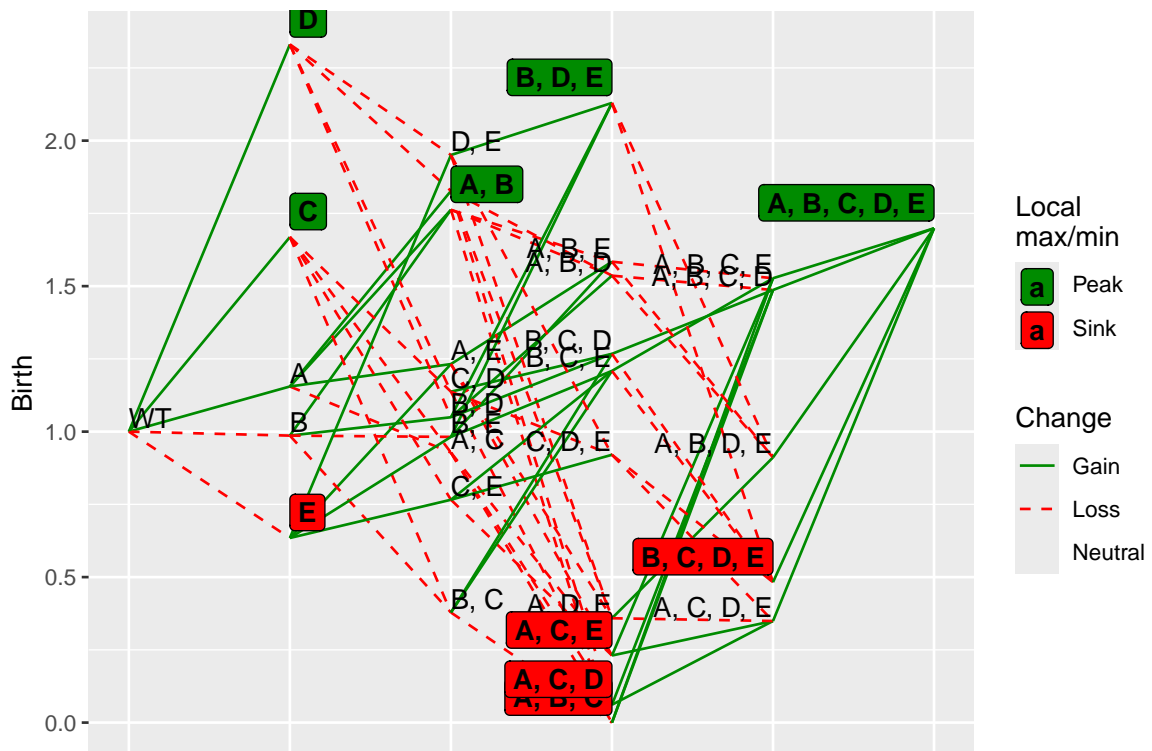
9.2   Random fitness landscapes from Kauffman's NK model

You can also use Kauffman's NK model (Brouillet et al., 2015; e.g., Ferretti et al., 2016). We call the function `fl_generate` from MAGELLAN (Brouillet et al., 2015).

```
rnk <- rfitness(5, K = 3, model = "NK")
plot(rnk)
```

```
oncoSimulIndiv(allFitnessEffects(genotFitness = rnk),
               onlyCancer = TRUE)
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = allFitnessEffects(genotFitness = rnk), onlyCancer = TRUE)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         7     1.26e+08     1.26e+08             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0       401    1635
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA     149820      2e+05
##   OccurringDrivers
## 1
##
## Final population composition:
##   Genotype        N
## 1            8.86e+02
## 2        A 0.00e+00
## 3     A, C 2.66e+02
## 4     B, C 1.26e+02
## 5        C 1.26e+08
## 6     C, D 3.68e+02
## 7     C, E 2.46e+02
```

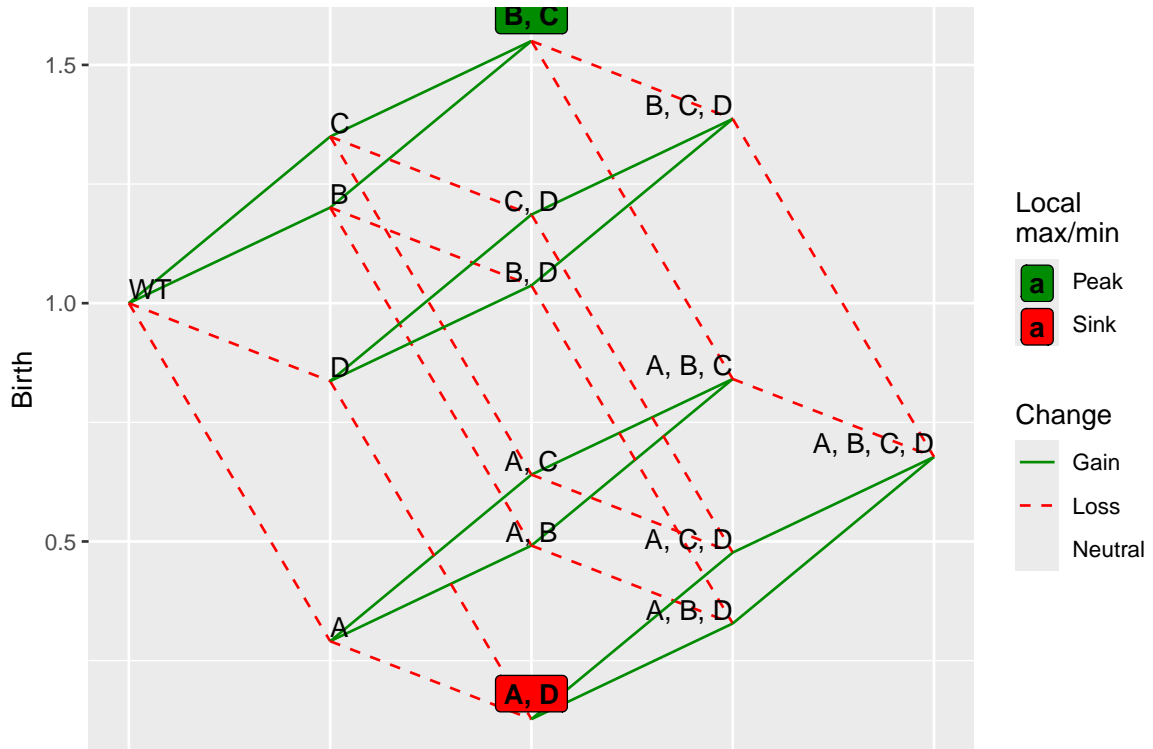## 9.3   Random fitness landscapes from an additive model

This model evaluates fitness with different contributions of each allele, which will be randomly generated.

Given a number a genes by the user, the code uses rnorm to generate random contribution for the mutated allele in each locus. Later, this constributions will be used in the generation of the matrix that gives the value of fitness for each combination of wild type/mutated alleles by addition of the values for each locus and combination.

```
radd <- rfitness(4, model = "Additive", mu = 0, sd = 0.5)
plot(radd)
```

9.4   Random fitness landscapes from Eggbox model

You can also use Eggbox model (Brouillet et al., 2015; e.g., Ferretti et al., 2016), where each locus is either high or low fitness (depending on the "e" parameter value), with a systematic change between each neighbor. We call the function `fl_generate` from MAGELLAN (Brouillet et al., 2015) to generate these landscapes.

```
regg <- rfitness(4, model = "Eggbox", e = 1, E = 0.5)
plot(regg)
```
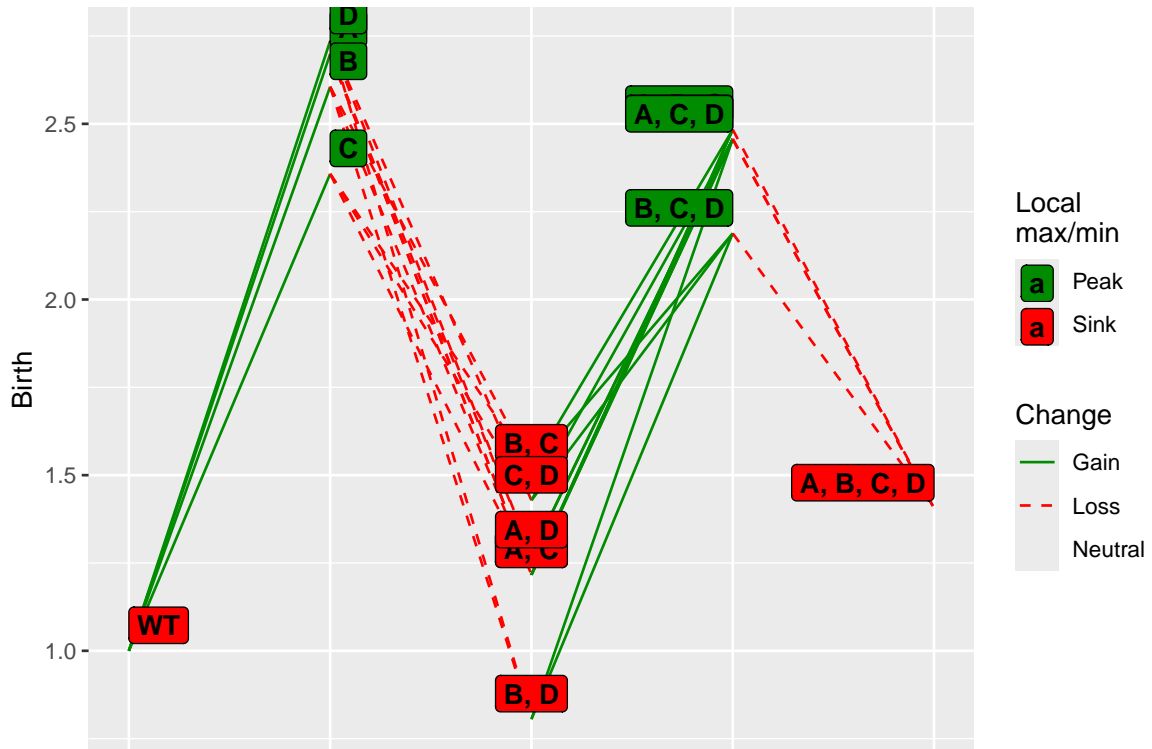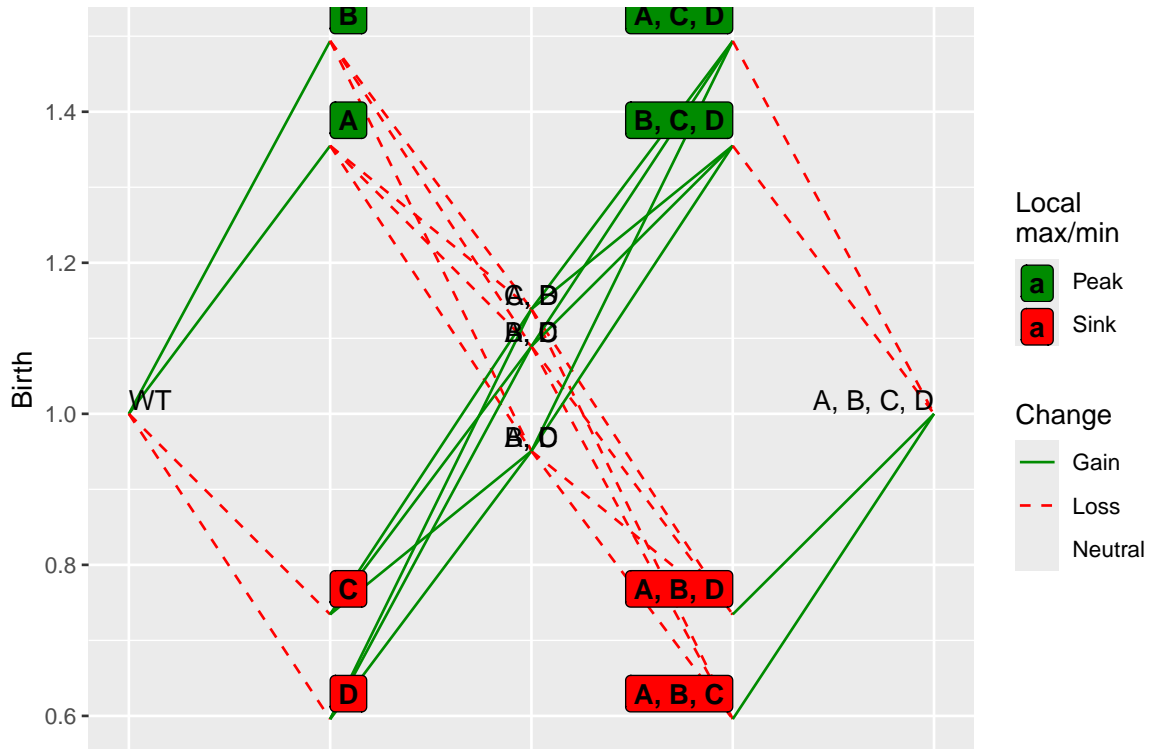
9.5 Random fitness landscapes from Ising model

In the Ising model (Brouillet et al., 2015; e.g., Ferretti et al., 2016), loci are arranged sequentially and each locus interacts with its physical neighbors. For each pair of interacting loci, there is a cost to (log)fitness if both alleles are not identical (and therefore 'compatible'); in this case, the cost for incompatibility i is applied. The last and the first loci will interact only if 'circular' is set. The implementation of this model is decribedin (Brouillet et al., 2015), and we use a call to MAGELLAN code to generate the landscape.

```
ris <- rfitness(g = 4, model = "Ising", i = 0.002, I = 0.45)
plot(ris)
```

9.6 Random fitness landscapes from Full models

MAGELLAN also offers the possibility to combine different models with their own parameters in order to generate a Full model. The models combined are:

- House of cards: `H` is the number of interacting genes.
- Multiplicative (what we call additive): `s` and `S` mean and SD for generating random fitnesses. `d` is a diminishing (negative) or increasing (positive) return as you approach the peak.
- Kauffman NK: each locus interacts with any other K loci that can be chosen randomly pasing "r = TRUE" or among its neigbors.
- RMF: explained at 9.1.
- Ising: `i` and `I` mean and SD for incompatibility. If `circular` option is provided, the last and first alleles can interact (circular arrangement).
- Eggbox: In this model, each locus is considered as high or low fitness. From one locus to another the fitness changes its sign so that one is benefficial and its neigbor is detrimental. `e` and `E` are fitness and noise for fitness.
- Optimum: there is an optimum fitness contribution defined by `o` *mu* and `O` *sigma* and every locus has a production `p` and `P` (also mean and sd respectively).

All models can be taken into account for the fitness calculation. With default parameters, neither of Ising, Eggbox or Optimum contribute to fitness lanadcape generation as all `i`, `e`, `o` and `p` all `== 0`. Also, as all parameters refering to standard deviations have value `== -1`, those are also have no effect unless changed. Further details can be found in MAGELLAN's webpage http://wwwabi.snv.jussieu.fr/public /Magellan/ and (Brouillet et al., 2015).

```
rnk <- rfitness(4, model = "Full", i = 0.5, I = 0.1,
                e = 2, s = 0.3, S = 0.08)
plot(rnk)
```
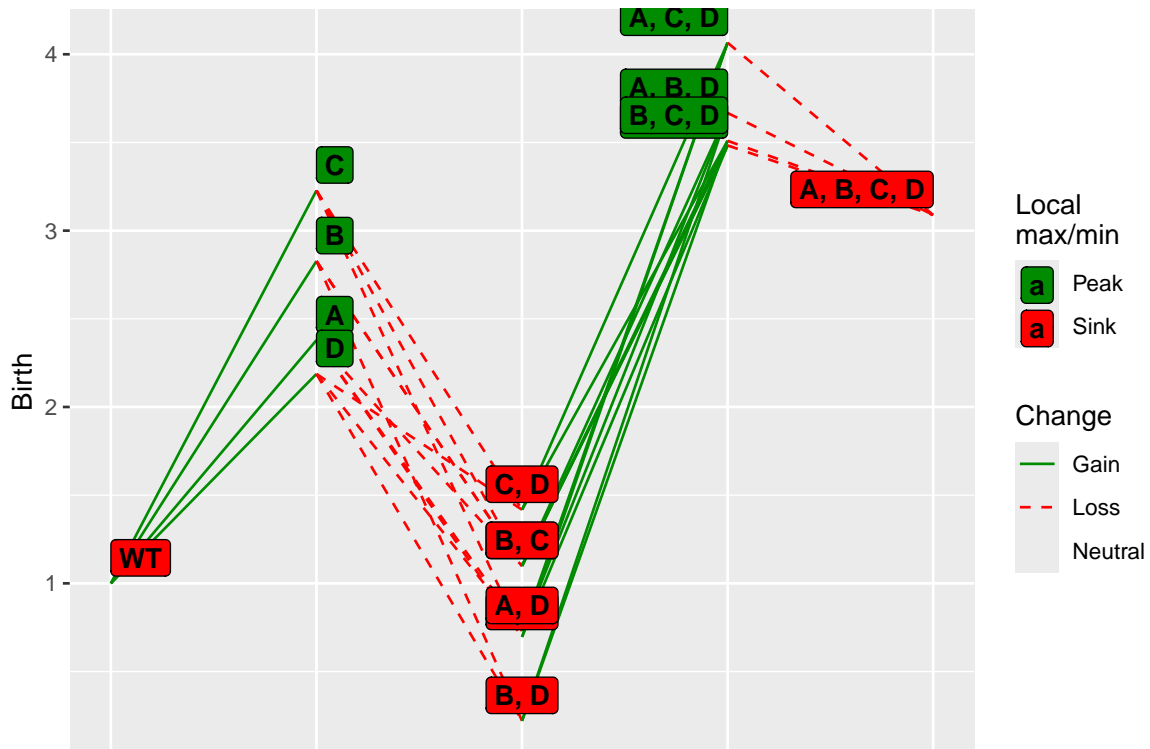

9.7 Epistasis and fitness landscape statistics

We can call MAGELLAN's (Brouillet et al., 2015) `fl_statistics` to obtain fitness landscape statistics, including measures of sign and reciprocal sign epistasis. See the help of `Magellan_stats` for further details on output format. For example:

```
rnk1 <- rfitness(6, K = 1, model = "NK")
Magellan_stats(rnk1)
##      ngeno      npeaks      nsinks       gamma      gamma.         r.s
##     64.000       4.000       6.000       0.668       0.396       1.286
##     nchains      nsteps        nori       depth        magn        sign
##      5.000      10.000       8.000       2.000       0.417       0.183
##       rsign        f.1.        X.2.       f.3..      mode_f      outD_m
##       0.100       0.585       0.344       0.071       1.000       2.888
##      outD_v      steps_m      reach_m       fitG_m       opt_i mProbOpt_0
##       2.825       2.297       6.533      32.883      20.000       0.385
##     opt_i.1 mProbOpt_1     opt_i.2 mProbOpt_2     opt_i.3 mProbOpt_3
##      30.000       0.154      43.000       0.013      48.000       0.447

rnk2 <- rfitness(6, K = 4, model = "NK")
Magellan_stats(rnk2)
##      ngeno      npeaks      nsinks       gamma      gamma.         r.s
```

```
##     64.000      4.000      5.000      0.322      0.150      2.813
##     nchains     nsteps       nori      depth       magn       sign
##      6.000     11.000     10.000      2.000      0.354      0.442
##      rsign       f.1.        X.2.      f.3..     mode_f     outD_m
##      0.204      0.231      0.357      0.412      2.000      1.837
##     outD_v     steps_m     reach_m      fitG_m      opt_i mProbOpt_0
##      3.079      2.681     16.883     33.267      9.000      0.041
##    opt_i.1 mProbOpt_1     opt_i.2 mProbOpt_2     opt_i.3 mProbOpt_3
##     23.000      0.468     42.000      0.265     49.000      0.226
```

(These fitness landscapes are, of course, frequency-independent fitness landscapes; with frequency-dependent fitness, as in section @ref{fdf} fitness landscapes as such are not defined.)

10 Frequency-dependent fitness

(*Note that except for the example of 10.2, based on Hurlbut et al. (2018), the examples below are not used because of their biological realism, but rather to show some key features of the software*)

With frequency-dependence fitness we can make fitness (actually, birth rate) depend on the frequency of other genotypes. We specify how the fitness (birth rate) of each genotype depends (or not) on other genotypes. Thus, this is similar to the explicit mapping of genotypes to fitness (see 3.1.1), but fitness can be a function of the abundance (relative or absolute) of other genotypes. Frequency-dependent fitness allows you to examine models from **game theory** and **adaptive dynamics**. Game theory has long tradition in evolutionary biology (Maynard Smith, 1982) and has been widely used in cancer (Archetti & Pienta, 2019, for classical papers that cover from early uses to a very recent review; Basanta & Deutsch, 2008; see, for example, Tomlinson, 1997).

Since birth rate can be an arbitrary function of the frequencies of other clones, we can model competition, cooperation and mutualism, parasitism and predation, and commensalism. (Recall that in the "Exp" model death rate is constant and fixed to 1. In the "McFL" and "McFLD" models, death rate is density-dependent —but not frequency-dependent. You can thus model all those phenomena by, for example, making the effects of clones i, j on each other and on their own be asymmetric on their birth rates). See examples in section 10.4; as explained there, if you use the "Exp" model, you might want to decrease the value of `sampleEvery`.

The procedure for working with the frequency-dependent functionality is the general one with OncoSimulR. We first create a data frame with the mapping between genotypes and their (frequency-dependent) fitness, similar to section 3.1.1. For example, a two-column data frame, where the first column are the genotypes and the second column contains, as strings, the expressions for the function that relate fitness to frequencies of other genotypes. (We can also use a data frame with $g + 1$ columns; each of the first g columns contains a 1 or a 0 indicating that the gene of that column is mutated or not. Column $g + 1$ contains the expressions for the fitness specifications; see `oncoSimulIndiv` and `allFitnessEffects` for examples). Once this data frame is created, we pass it to `allFitnessEffects`. From there, simulation proceeds as usual.

How complex can the functions that specify fitness be? We use library ExprTk for the fitness specifications so the range of functions you can use is very large (http://www.partow.net/programming/exprtk/), including of course the usual arithmetic expressions, logical expressions (so you can model thresholds or jumps and use step functions), and a wide range of mathematical functions (so linear, non-linear, convex, concave, etc, functions can be used, including of course affine fitness functions as in Gerstung, Nakhoul, et al., 2011).

10.1 A first example with frequency-dependent fitness

The following is an arbitrary example. We will model birth rate of some genotypes as a function of the relative frequencies of other genotypes; we use `f_1` to denote the relative frequency of the genotype with the first gene mutated, `f_1_2` to denote the relative frequency of the genotype with the first and second genes mutated, etc, and `f_` to denote the frequency of the WT genotype —below, in sections 10.3, 10.5, and 10.4, we will use absolute number of cells instead of relative frequencies). (As we have discussed already, instead of `f_1` you can, and probably should for any example except trivial ones, use `f_A` or `f_genotype expressed as combination of gene names`).

As you can see below, the birth rate of genotype "A" $= 1.2 + 1.5 * f\_A\_B$ and that of the wildtype $= 1 + 1.5 * f\_A\_B$. Genotype "A, B" in this example could be a genotype whose presence leads to an increase in the growth of other genotypes (maybe via diffusible factors, induction of angiogenesis, etc). Genotype "B" does not show frequency-dependence. The birth rate of genotype "C" increases with the frequency of `f_A_B` and increases (adding 0.7) with the frequency of genotypes "A" and "B", but only if the sum of the frequencies of genotypes "A" and "B" is larger than 0.3. For genotype "A, B" its fitness increases with the square root of the sum of the frequencies of genotypes "A", "B", and "C", but it decreases (i.e., shows increased intra-clone competition) if its own frequency is larger than 0.5. Genotypes not defined explicitly have a fitness of 0.

```
## Define fitness of the different genotypes
gffd <- data.frame(
    Genotype = c("WT", "A", "B", "C", "A, B"),
    Fitness = c("1 + 1.5 * f_A_B",
                "1.3 + 1.5 * f_A_B",
                "1.4",
                "1.1 + 0.7*((f_A + f_B) > 0.3) + f_A_B",
                "1.2 + sqrt(f_1 + f_C + f_B) - 0.3 * (f_A_B > 0.5)"))
```

(In the data frame creation, we use `stringsAsFactors = FALSE` to avoid messages about conversions between factors and characters in former versions of R).

You could also specify that as

```
## Define fitness of the different genotypes
gffdn <- data.frame(
    Genotype = c("WT", "A", "B", "C", "A, B"),
    Fitness = c("1 + 1.5 * f_1_2",
                "1.3 + 1.5 * f_1_2",
                "1.4",
                "1.1 + 0.7*((f_1 + f_2) > 0.3) + f_1_2",
                "1.2 + sqrt(f_1 + f_3 + f_2) - 0.3 * (f_1_2 > 0.5)"),
    stringsAsFactors = FALSE)
```

but it is *strongly preferred* to use explicit gene name letters (otherwise, you must

keep in mind how R orders names of genes when making the mapping from letters to numbers).

Let us verify that we have specified what we think we have specified using `evalAllGenotypes` (we have done this repeatedly in this vignette, for example in 1.3.2 or 1.7 or 5.1. Because fitness can depend on population sizes of different populations, we need to pass the populations sizes at which we want fitness evaluated in `evalAllGenotypes`.

Note that when calling `allFitnessEffects` we have to set the paramenter `frequencyDependentFitness` to TRUE. Since we are using relative frequencies, we can be explicit and specify `freqType = "rel"` (though it is not needed). We will see below (10.3, 10.5, and 10.4) several examples with absolute numbers.

When passing `spPopSizes` it is also strongly preferred to use a named vector as that allows the code to run some checks. Otherwise, the order of the population sizes *must* be identical to that in the table with the fitness descriptions (component `full_FDF_spec` in the fitness effects object).

```
evalAllGenotypes(allFitnessEffects(genotFitness = gffd,
                       frequencyDependentFitness = TRUE,
                       frequencyType = "rel"),
              spPopSizes = c(WT = 100, A = 20, B = 20, C = 30, "A, B" = 0))
## Warning in allFitnessEffects(genotFitness = gffd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 1       WT   1.000
## 2        A   1.300
## 3        B   1.400
## 4        C   1.100
## 5     A, B   1.842
## 6     A, C   0.000
## 7     B, C   0.000
## 8  A, B, C   0.000
```

```
## Notice the warning
evalAllGenotypes(allFitnessEffects(genotFitness = gffd,
                       frequencyDependentFitness = TRUE,
                       frequencyType = "rel"),
              spPopSizes = c(100, 30, 40, 0, 10))
## Warning in allFitnessEffects(genotFitness = gffd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##                to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##                to last version.
##   Genotype Fitness
## 1       WT   1.083
## 2        A   1.383
## 3        B   1.400
## 4        C   1.856
## 5     A, B   1.824
## 6     A, C   0.000
## 7     B, C   0.000
## 8  A, B, C   0.000
```

```r
evalAllGenotypes(allFitnessEffects(genotFitness = gffd,
                         frequencyDependentFitness = TRUE,
                         frequencyType = "rel"),
                 spPopSizes = c(100, 30, 40, 0, 100))
```
```
## Warning in allFitnessEffects(genotFitness = gffd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## Using old version of fitnessEffects. Transforming fitnessEffects
##                to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##                to last version.
##   Genotype Fitness
## 1       WT   1.556
## 2        A   1.856
## 3        B   1.400
## 4        C   1.470
## 5     A, B   1.709
## 6     A, C   0.000
## 7     B, C   0.000
## 8  A, B, C   0.000
```

The numbered one gives the same results. Note as well that using `frequencyType` is not needed (the default, `auto`, infers the type)

```r
evalAllGenotypes(allFitnessEffects(genotFitness = gffdn,
                         frequencyDependentFitness = TRUE),
                 spPopSizes = c(100, 20, 20, 30, 0))
```
```
## Warning in allFitnessEffects(genotFitness = gffdn,
```

```
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
##   Genotype Fitness
## 1       WT   1.000
## 2        A   1.300
## 3        B   1.400
## 4        C   1.100
## 5     A, B   1.842
## 6     A, C   0.000
## 7     B, C   0.000
## 8  A, B, C   0.000
```

```
evalAllGenotypes(allFitnessEffects(genotFitness = gffdn,
                     frequencyDependentFitness = TRUE),
              spPopSizes = c(100, 30, 40, 0, 10))
## Warning in allFitnessEffects(genotFitness = gffdn,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
##   Genotype Fitness
## 1       WT   1.083
## 2        A   1.383
## 3        B   1.400
## 4        C   1.856
## 5     A, B   1.824
## 6     A, C   0.000
## 7     B, C   0.000
## 8  A, B, C   0.000
```

```
evalAllGenotypes(allFitnessEffects(genotFitness = gffdn,
                     frequencyDependentFitness = TRUE),
              spPopSizes = c(100, 30, 40, 0, 100))
```

```
## Warning in allFitnessEffects(genotFitness = gffdn,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
##    Genotype Fitness
## 1        WT   1.556
## 2         A   1.856
## 3         B   1.400
## 4         C   1.470
## 5      A, B   1.709
## 6      A, C   0.000
## 7      B, C   0.000
## 8   A, B, C   0.000
```

The fitness specification is correct. Let us now create the `allFitnessEffects` object and simulate. We will use the McFL model, so in addition to the frequency dependence in the birth rates, there is also density dependence in the death rate (see section 3).

```
afd <- allFitnessEffects(genotFitness = gffd,
                         frequencyDependentFitness = TRUE,
                         frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = gffd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
set.seed(1) ## for reproducibility
sfd <- oncoSimulIndiv(afd,
                      model = "McFL",
                      onlyCancer = FALSE,
                      finalTime = 55, ## short, for speed
                      mu = 1e-4,
                      initSize = 5000,
                      keepPhylog = FALSE,
                      seed = NULL,
                      errorHitMaxTries = FALSE,
                      errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
```

```
##                  to last version.
```

```
plot(sfd, show = "genotypes")
```
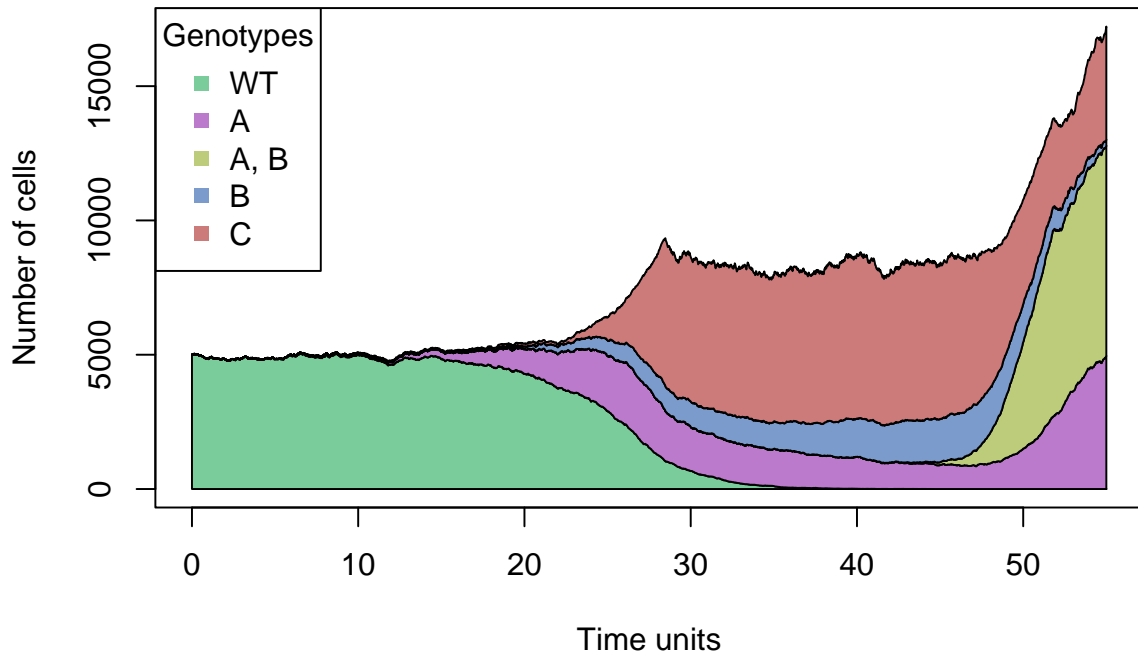

There is no need to specify the fitness of all possible genotypes (and no need to always specify the fitness of a WT): those are taken to be 0. But no fitness expression can, thus, contain a function of the genotypes for which fitness is not specified (e.g., suppose you do not pass the fitness of genotype "B": it will be taken as 0; but no genotype can have, in its fitness, a function such as "2 * f_B").

10.2 Hurlbut et al., 2018: a four-cell example with angiogenesis and cytotoxicity

The following example is based on Hurlbut et al. (2018). As explained in p. 3 of that paper, "Stromal cancer cells (A-) [WT in the code below] have no particular benefit or cost unique to themselves, and they are considered a baseline neutral cell within the context of the model. In contrast, angiogenesis-factor producing cells (A+) [A in the code below] vascularize the local tumor area which consequently introduces a nutrient rich blood to the benefit of all interacting cells. Nutrient recruitment expands when A+ cells interact with one another. Cytotoxic cells (C) release a chemical compound which harms heterospecific cells and increases their rate of cell death. The cytotoxic cells benefit from the resulting disruption in competition caused by the interaction. For simplicity, our model presumes that cytotoxic cells are themselves immune to this class of agent. Finally, proliferative cells (P) possess a reproductive or metabolic advantage relative to the other cell types. In our model this advantage does not compound with the nutrient enrichment produced by vascularization when A+ cells are present; however, it does place the proliferative cell at a greater vulnerability to cytotoxins."

They provide, in p. 4, the payoff matrix reproduced in Figure 7:

Player\Opponent	A$-$	A+	P	C
A$-$	1	$1+d$	1	$1-c$
A+	$1-a+d$	$1-a+d+f$	$1-a+d$	$1-c-a+d$
P	$1+g$	$1+d+g$	$1+g$	$(1+g)(1-c)$
C	$1-b+e$	$1-b+d+e$	$1-b+e$	$1-b$

Figure 7: Payoff matrix from Table 2 of Hurlbut et al., 2018, 'Game Theoretical Model of Cancer Dynamics withFour Cell Phenotypes', *Games*, 9, 61; doi:10.3390/g9030061.

As explained in p. 6 (equation 1) of Hurlbut et al. (2018), we can write the fitness of the four types as $\mathbf{f}=\mathbf{G}\mathbf{u}$, where $\mathbf{f}$ and $\mathbf{u}$ are the vectors with the four fitnesses and frequencies (where each element of $\mathbf{u}$ corresponds to the relative frequencies of stromal, angiogenic, proliferative, and cytotoxic cells), and G is the payoff matrix in Figure 7.

To allow modelling scenarios with different values for the parameters in Figure 7 we will define a function to create the data frame of frequency-dependent fitnesses. First, we will assume that each one of types A+, P, and C, are all derived from WT by a single mutation in one of three genes, say, A, P, C, respectively.

```
create_fe <- function(a, b, c, d, e, f, g,
                      gt = c("WT", "A", "P", "C")) {
  data.frame(Genotype = gt,
             Fitness = c(
               paste0("1 + ",
                      d, " * f_A ",
                      "- ", c, " * f_C"),
               paste0("1 - ", a,
                      " + ", d, " + ",
                      f, " * f_A ",
                      "- ", c, " * f_C"),
               paste0("1 + ", g, " + ",
                      d, " * f_A ",
                      "- ", c, " * (1 + ",
                      g, ") * f_C"),
               paste0("1 - ", b, " + ",
                      e, " * f_ + ",
                      "(", d, " + ",
                      e, ") * f_A + ",
                      e , " * f_P")))
```

```
}
```

We can check we recover Figure 7:

```
create_fe("a", "b", "c", "d", "e", "f", "g")
##   Genotype                                  Fitness
## 1       WT                      1 + d * f_A - c * f_C
## 2        A            1 - a + d + f * f_A - c * f_C
## 3        P      1 + g + d * f_A - c * (1 + g) * f_C
## 4        C 1 - b + e * f_ + (d + e) * f_A + e * f_P
```

We could model a different set of ancestor-dependent relationships:

```
## Different assumption about origins from mutation:
## WT -> P; P -> A,P; P -> C,P

create_fe2 <- function(a, b, c, d, e, f, g,
                       gt = c("WT", "A", "P", "C", "A, P", "A, C",
                              "C, P")) {
  data.frame(Genotype = gt,
             Fitness = c(
                 paste0("1 + ",
                        d, " * f_A_P ",
                        "- ", c, " * f_P_C"),
                 "0",
                 paste0("1 + ", g, " + ",
                        d, " * f_A_P ",
                        "- ", c, " * (1 + ",
                        g, ") * f_P_C"),
                 "0",
                 paste0("1 - ", a, " + ",
                 d, " + ",
                        f, " * f_A_P ",
                        "- ", c, " * f_P_C"),
                 "0",
                 paste0("1 - ", b, " + ",
                        e, " * f_ + ",
                        "(", d, " + ",
                        e, ") * f_A_P + ",
                        e , " * f_P")),
             stringsAsFactors = FALSE)
}


## And check:
create_fe2("a", "b", "c", "d", "e", "f", "g")
##   Genotype                                  Fitness
## 1       WT                  1 + d * f_A_P - c * f_P_C
```

```
## 2        A                                               0
## 3        P     1 + g + d * f_A_P - c * (1 + g) * f_P_C
## 4        C                                               0
## 5     A, P          1 - a + d + f * f_A_P - c * f_P_C
## 6     A, C                                               0
## 7     C, P 1 - b + e * f_ + (d + e) * f_A_P + e * f_P
```

Note: we are writing `f_P_C`: this is remapped internally to `f_C_P` (which is the genotype name, with gene names reordered).

To show two examples, we will run the analyses Hurlbut et al. (2018) use for Figures 3a and 3b (p.8 of their paper):

```
## Figure 3a
afe_3_a <- allFitnessEffects(
                genotFitness =
                        create_fe(0.02, 0.04, 0.08, 0.06,
                                  0.15, 0.1, 0.06),
                frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = create_fe(0.02, 0.04,
## 0.08, : v2 functionality detected. Adapting to v3 functionality.


## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(s_3_a)
if (FALSE) {
set.seed(2)
s_3_a <- oncoSimulIndiv(afe_3_a,
                model = "McFL",
                onlyCancer = FALSE,
                finalTime = 160,
                mu = 1e-4,
                initSize = 5000,
                keepPhylog = FALSE,
                seed = NULL,
                errorHitMaxTries = FALSE,
                errorHitWallTime = FALSE,
                keepEvery = 1)
}
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue"))
```

```
## Figure 3b
afe_3_b <- allFitnessEffects(
                genotFitness =
                    create_fe(0.02, 0.04, 0.08, 0.1,
                              0.15, 0.1, 0.05),
                frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = create_fe(0.02, 0.04,
## 0.08, : v2 functionality detected. Adapting to v3 functionality.
```
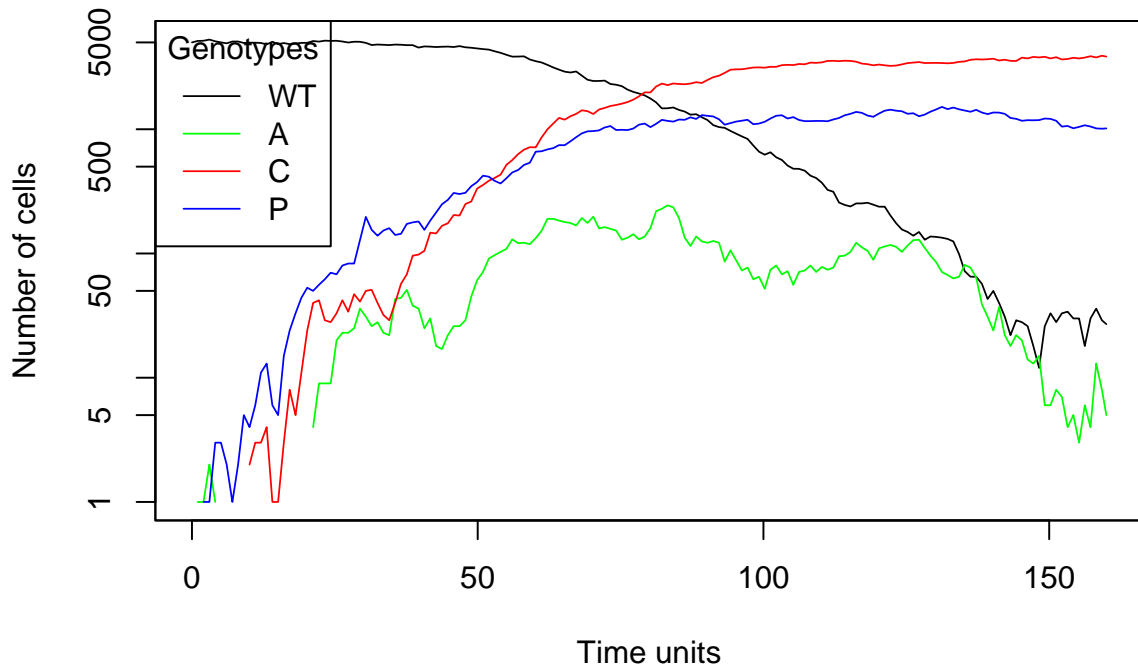
```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below
data(s_3_b)

if (FALSE) {
set.seed(2)
## Use a short finalTime, for speed of vignette execution
s_3_b <- oncoSimulIndiv(afe_3_b,
                model = "McFL",
                onlyCancer = FALSE,
                finalTime = 100,
                mu = 1e-4,
                initSize = 5000,
                keepPhylog = FALSE,
                seed = NULL,
                errorHitMaxTries = FALSE,
                errorHitWallTime = FALSE,
                keepEvery = 1)

}
```

233

```r
plot(s_3_b, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue"))
```
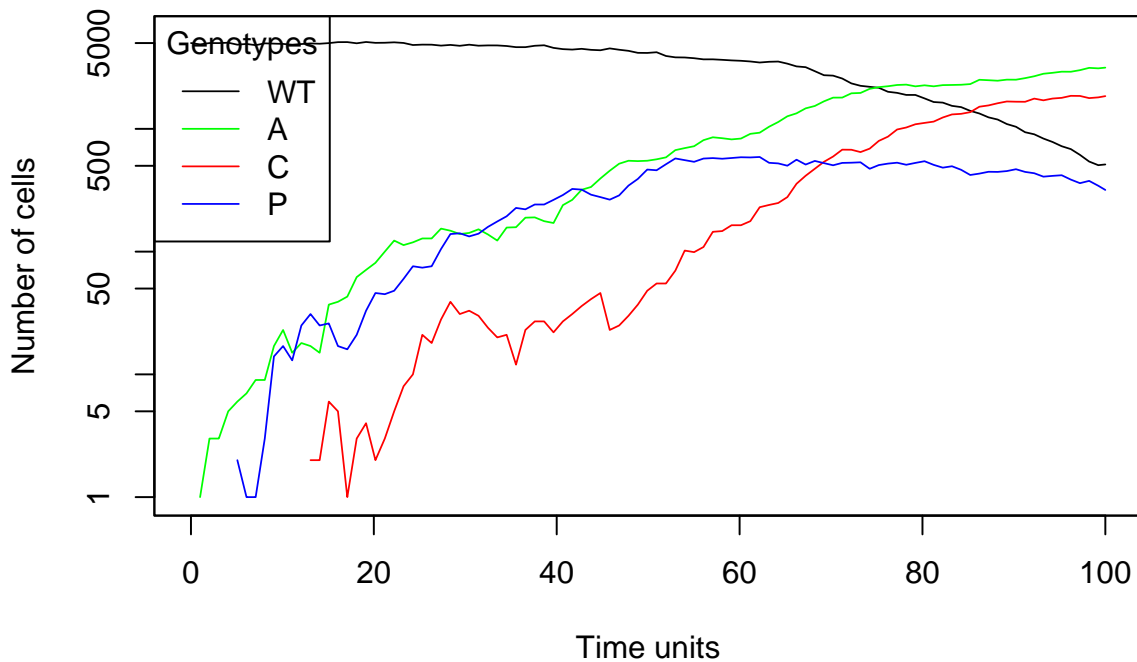


```r
## we are using keepEvery. O.w. we could have used
## thinData = TRUE for faster plotting
```

Of course, if we assume that the mutations leading to the different cell types are different, the results can change:

```r
## Figure 3b. Now with WT -> P; P -> A,P; P -> C,P
## For speed, we set finalTime = 100
afe_3_b_2 <- allFitnessEffects(
                genotFitness =
                    create_fe2(0.02, 0.04, 0.08, 0.1,
                               0.15, 0.1, 0.05),
                frequencyDependentFitness = TRUE,
                frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe2(0.02, 0.04,
## 0.08, : v2 functionality detected. Adapting to v3 functionality.
```

```r
set.seed(2)
s_3_b_2 <- oncoSimulIndiv(afe_3_b_2,
                model = "McFL",
                onlyCancer = FALSE,
                finalTime = 100,
                mu = 1e-4,
                initSize = 5000,
                keepPhylog = FALSE,
                seed = NULL,
                errorHitMaxTries = FALSE,
```
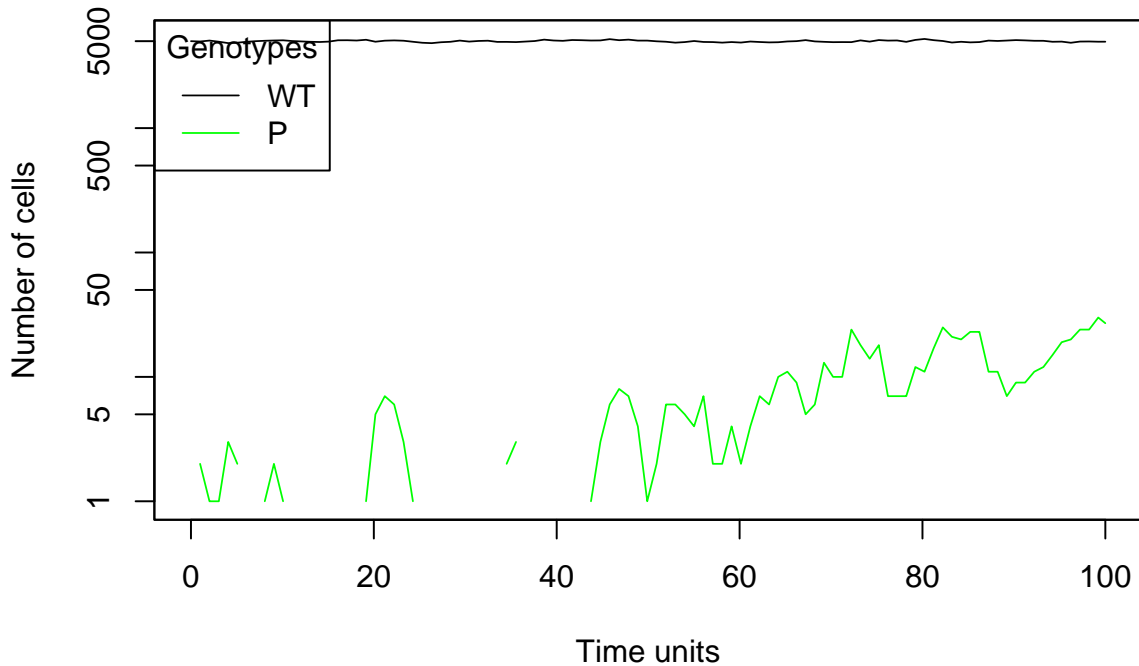
```
                      errorHitWallTime = FALSE,
                      keepEvery = 1)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
plot(s_3_b_2, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue"))
```


(Examples for the remaining Figures 2 and 3 are provided, using also `oncoSimulPop`, in file 'inst/miscell/hurlbut-ex.R')

10.3   An example with absolute numbers and population collapse

In the following example we use absolute numbers (thus the `n_1`, etc, instead of the `f_1` in the fitness definition). This is a toy model where there is a change in fitness in two of the genotypes if the other is above a specified threshold (this also shows again the usage of a logical inequality). The genotype with gene B mutated has birth rate less than 1, unless there are at least more than 10 cells with genotype A mutated.

```
gffd3 <- data.frame(Genotype = c("WT", "A", "B"),
                    Fitness = c("1",
                                "1 + 0.2 * (n_B > 10)",
                                ".9 + 0.4 * (n_A > 10)"
                                ))
afd3 <- allFitnessEffects(genotFitness = gffd3,
                          frequencyDependentFitness = TRUE)
```

235

```
## Warning in allFitnessEffects(genotFitness = gffd3,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
```

As usual, let us verify that we have specified what we think we have specified using
`evalAllGenotypes`:

```
evalAllGenotypes(allFitnessEffects(genotFitness = gffd3,
                        frequencyDependentFitness = TRUE),
                spPopSizes = c(WT = 100, A = 1, B = 11))
## Warning in allFitnessEffects(genotFitness = gffd3,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
##   Genotype Fitness
## 1       WT     1.0
## 2        A     1.2
## 3        B     0.9
## 4     A, B     0.0
```

```
evalAllGenotypes(allFitnessEffects(genotFitness = gffd3,
                        frequencyDependentFitness = TRUE),
                spPopSizes = c(WT = 100, A = 11, B = 1))
Warning in allFitnessEffects(genotFitness = gffd3,
frequencyDependentFitness = TRUE): v2 functionality detected.
Adapting to v3 functionality.
Genotype Fitness
1   WT   1.0
2   A   1.0
3   B   1.3
4   A, B   0.0
```

In this simulation, the population collapses: genotype B is able to invade the
population when there are some A's around. But as soon as A disappears due to
competition from B, B collapses as its birth rate becomes 0.9, less than the death rate;
we are using the "McFLD" model (where death rate $D(N) = \max(1, \log(1 + N/K))$);
see details in 3.2.1.1.

```
set.seed(1)
sfd3 <- oncoSimulIndiv(afd3,
                    model = "McFLD",
                    onlyCancer = FALSE,
                    finalTime = 200,
                    mu = 1e-4,
                    initSize = 5000,
                    keepPhylog = FALSE,
                    seed = NULL,
                    errorHitMaxTries = FALSE,
```
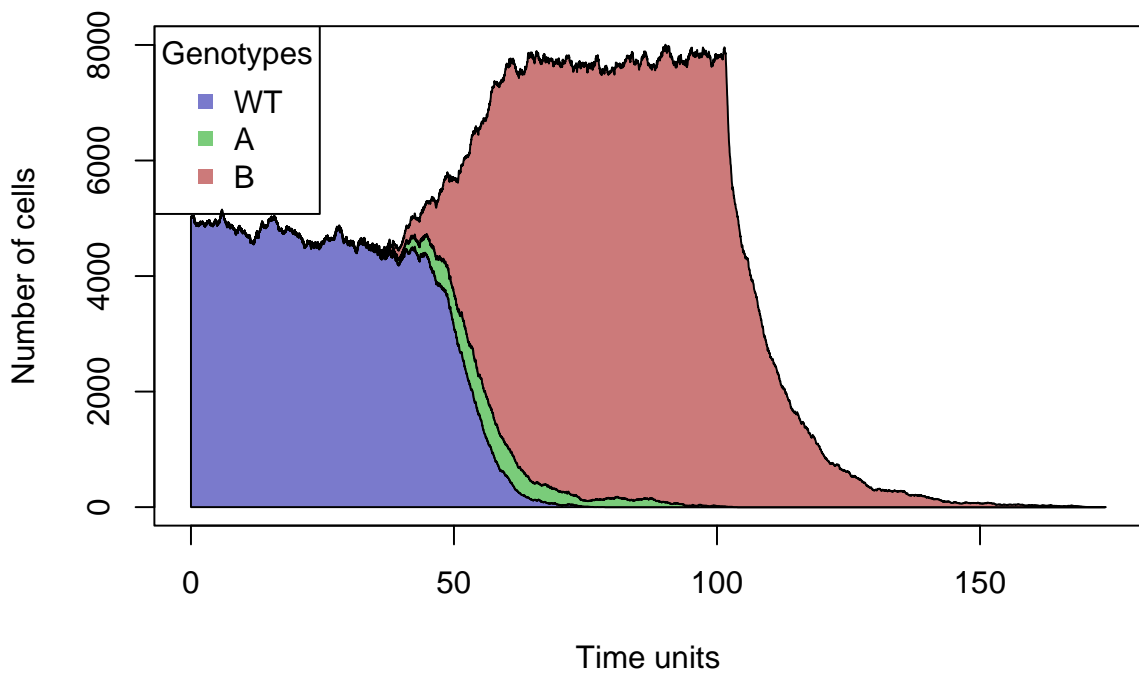
```
                    errorHitWallTime = FALSE)
Using old version of fitnessEffects. Transforming fitnessEffects
to last version.
Using old version of fitnessEffects. Transforming fitnessEffects
to last version.
```

```
plot(sfd3, show = "genotypes", type = "line")
```
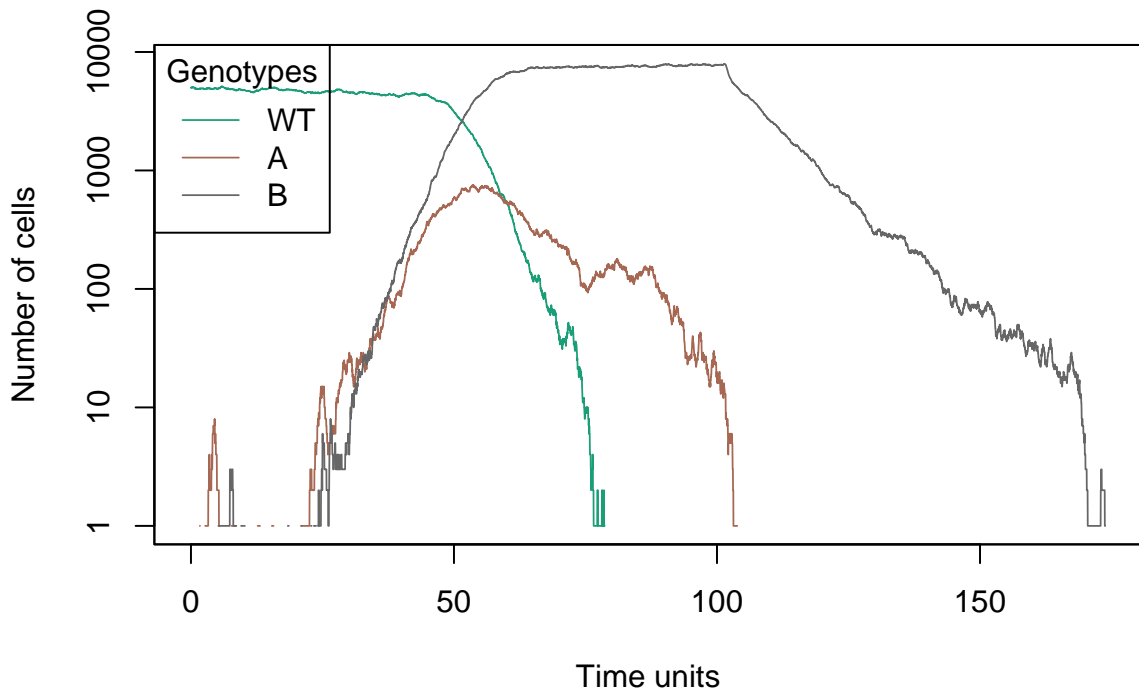


```
plot(sfd3, show = "genotypes")
```

```
sfd3
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = afd3, model = "McFLD", mu = 1e-04, initSize = 5000,
##     finalTime = 200, onlyCancer = FALSE, keepPhylog = FALSE,
##     errorHitWallTime = FALSE, errorHitMaxTries = FALSE, seed = NULL)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         3            0            0             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0     173.8    7037
##   HittedWallTime HittedMaxTries   errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.007211       5000       5000
##   OccurringDrivers
## 1
##
## Final population composition:
##   Genotype N
## 1          0
## 2        A 0
## 3        B 0
```

Had we used the "usual" death rate expression, that can lead to death rates below 1 (see 3.2.1.1), we would have obtained a population that stabilizes around a final value slightly below the initial one (the one that corresponds to a death rate equal to 0.9):

```
set.seed(1)
sfd4 <- oncoSimulIndiv(afd3,
                       model = "McFL",
                       onlyCancer = FALSE,
                       finalTime = 145,
                       mu = 1e-4,
                       initSize = 5000,
                       keepPhylog = FALSE,
                       seed = NULL,
                       errorHitMaxTries = FALSE,
                       errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.

plot(sfd4, show = "genotypes", type = "line")
```
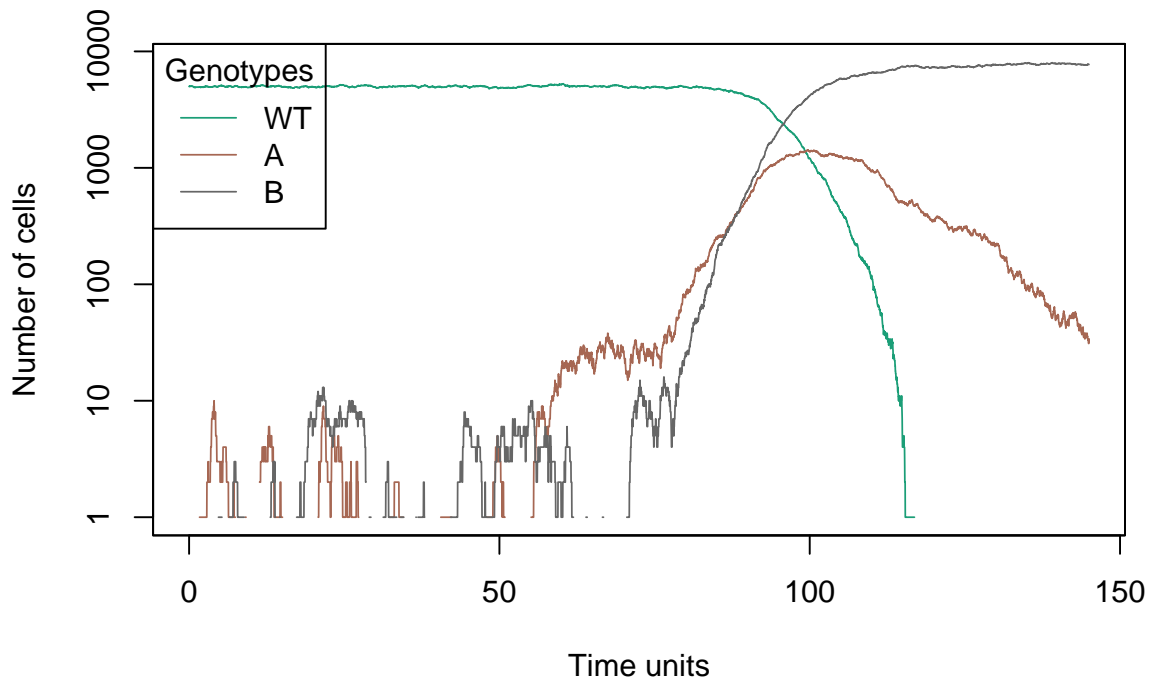
```r
plot(sfd4, show = "genotypes")
```
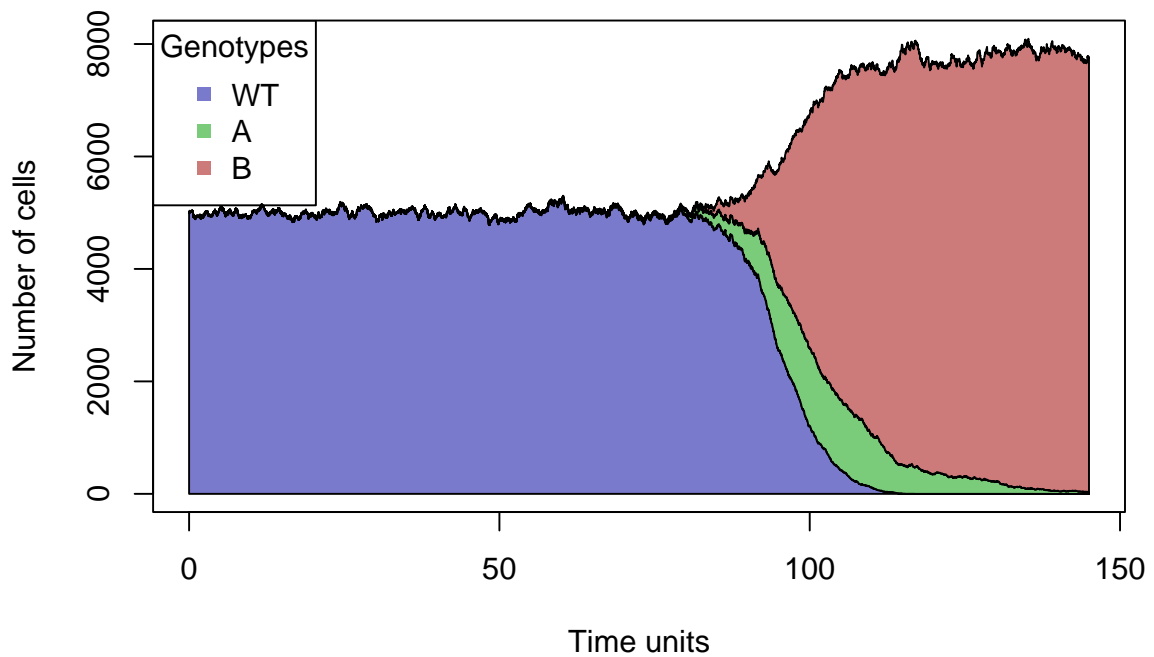


```
sfd4
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = afd3, model = "McFL", mu = 1e-04, initSize = 5000,
##     finalTime = 145, onlyCancer = FALSE, keepPhylog = FALSE,
##     errorHitWallTime = FALSE, errorHitMaxTries = FALSE, seed = NULL)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         3         7750         7716             0              0
```

```
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0       145     5941
##   HittedWallTime HittedMaxTries  errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.007544       4859       5000
##   OccurringDrivers
## 1
##
## Final population composition:
##   Genotype    N
## 1             0
## 2        A   34
## 3        B 7716
```

```
## Check final pop size corresponds to birth = death
K <- 5000/(exp(1) - 1)
K
## [1] 2910
```

```
log1p(4290/K)
## [1] 0.906
```

10.4 Predator-prey, commensalism, and consumer-resource models

Since birth rates can be arbitrary functions of frequencies of other clones, we can easily model classical ecological models, such as predator-prey, competition, commensalism and, more generally, consumer-resource models (see, for example, section 3.4 in Otto & Day, 2007).

OncoSimulR was originally designed as a forward-time genetic simulator; thus, we used to need to use a simple trick to get the system going. For example, suppose we want to model a predator-prey system; we could do this having a WT that can mutate into either preys or predators. (This is no longer necessary since we can specify starting the simulation from populations with arbitrary numbers of different clones; see sections 6.7 and 6.8. But we have not yet updated the examples).

Even with the There is, in fact, a small leakage in the system because both preys and predators are "leaking" a small number of children, via mutation to a non-viable predator-and-prey genotype, but this is negligible relative to their birth/death rates.

We will use below the usual consumer-resource model with Lotka-Volterra equations. To model them, we use the "Exp" model, which has a constant death rate of 1, and directly translate the usual expressions (e.g., expressions 3.15a and 3.15b in p. 73 of Otto & Day, 2007) for rates of change into the birth rate. Of course, you can use any other function for how rates of change of each "species" depend on the numbers of the different species, including constant inflow and outflow, Type I, Type II, and Type III functional responses, etc. And you can model systems that involve multiple different types of preys, predators, commensals, etc (see the example in 10.2 for a

four-type example).

What about `sampleEvery`? In "for real" work, and specially with complex models, you might want to decrease it, or at least examine how much results are affected by changes in `sampleEvery`. Decreasing `sampleEvery` will result in birth rates being updated more frequently, and we use the BNB algorithm, which updates all rates only when the whole population is sampled. (Recall that by default, in the `McFL` and `McFLD` models the setting for `sampleEvery` is smaller than in the `Exp` model). Further discussion of these issues is provided in sections 10.7 and 18.6.

Since the WT genotype is just a trick used to get the system going, we will make sure it disappears from the system soon after we get it to have both preys and predators (and we use the max function to prevent birth rate from ever becoming negative or identically 0). Finally, note that we can make this much more sophisticated; for example, we could get the system going differently by having different mutations from WT to prey and predator. Remember that you can now start the simulation from arbitrary initial compositions (section 6.7). This is left here for historical purposes and to show additional use cases.

10.4.1 Competition

First, create a function that will generate the usual Lotka-Volterra expressions for competition models (see below for this example without using the WT, and directly starting from "S1" and "S2").

```
G_fe_LV <- function(r1, r2, K1, K2, a_12, a_21, awt = 1e-4,
                                    gt = c("WT", "S1", "S2")) {
    data.frame(Genotype = gt,
               Fitness = c(
                   paste0("max(0.1, 1 - ", awt, " * (n_2 + n_1))"),
                   paste0("1 + ", r1,
                          " * ( 1 - (n_1 + ", a_12, " * n_2)/", K1,
                          ")"),
                   paste0("1 + ", r2,
                          " * ( 1 - (n_2 + ", a_21, " * n_1)/", K2,
                          ")")
               ))
}


## Show expressions for birth rates
G_fe_LV("r1", "r2", "K1", "K2", "a_12", "a_21", "awt")
##    Genotype                              Fitness
## 1        WT        max(0.1, 1 - awt * (n_2 + n_1))
## 2        S1 1 + r1 * ( 1 - (n_1 + a_12 * n_2)/K1)
## 3        S2 1 + r2 * ( 1 - (n_2 + a_21 * n_1)/K2)
```

Note we use numbers, not letters, in the expressions above (`n_1`, etc). This allows for reusing the function, but requires extra care making sure that the numbers match

the order of the genotypes (it is not a problem with "S1" and "S2", since there "S1" will always be 1 and "S2" 2; but what about "Predator" and "prey"?)

Remember the above are the birth rates for a model with death rate $= 1$ (the "Exp" model). That is why we added a 1 to the birth rate. If you subtract 1 from the expressions for the birth rates of predators and prey above you get the standard expressions for the differential equations for Lotka-Volterra model of competition: $\frac{dn_1}{dt} = r_1 n_1 (1 - \frac{n_1 + \alpha_{12} n_2}{K_1})$. (Verbosely: we are simulating using a model where we have, in a rather general expression, $\frac{dn_1}{dt} = (b - d) n_1$, where b and d are the birth and death rates (that could be arbitrary functions of other stuff); these are the birth and death rates used in the BNB algorithm. And when we simulate under the "Exp" model we have $d = 1$. So just solve for $(b - d)n_1 = r_1 n_1 (1 - \frac{n_1 + \alpha_{12} n_2}{K_1})$, with $b = 1$ and you get the b you need to use).

Now, run that model by setting appropriate parameters; see how we have $a\_12 > 0$ and $a\_21 > 0$.

```
fe_competition <-
    allFitnessEffects(
        genotFitness =
            G_fe_LV(1.5, 1.4, 10000, 4000, 0.6, 0.2,
                gt = c("WT","S1", "S2")),
        frequencyDependentFitness = TRUE,
        frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = G_fe_LV(1.5, 1.4,
## 10000, 4000, : v2 functionality detected. Adapting to v3
## functionality.
```

```
competition <- oncoSimulIndiv(fe_competition,
                        model = "Exp",
                        onlyCancer = FALSE,
                        finalTime = 100,
                        mu = 1e-4,
                        initSize = 40000,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
```
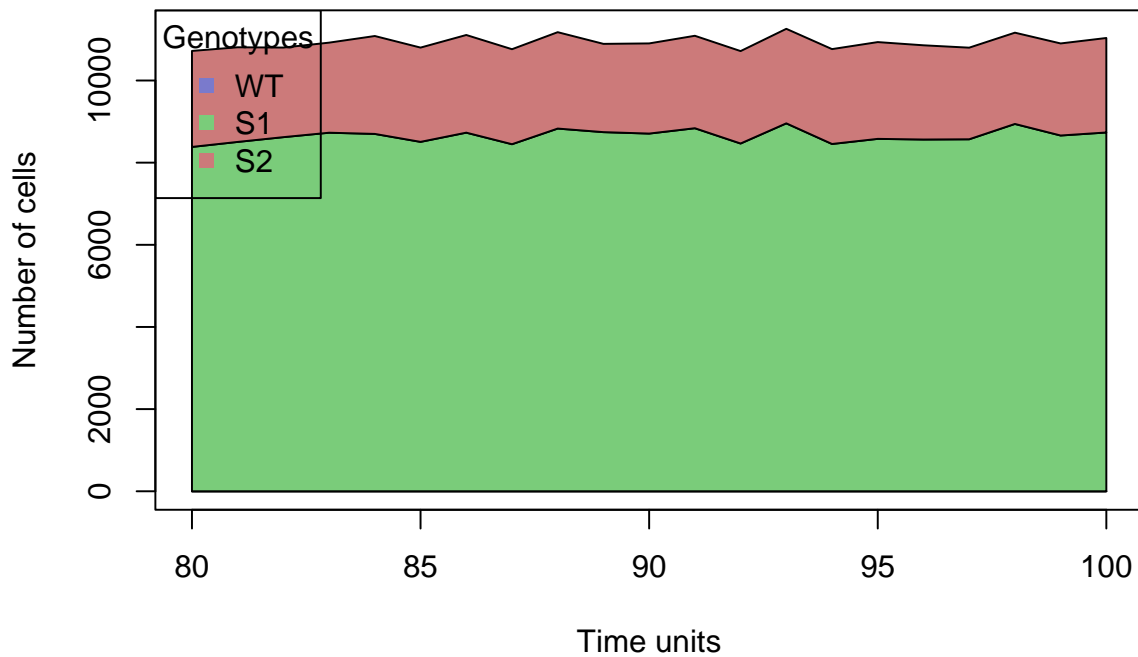
If we plot the whole simulation, we of course see the WT:

```
plot(competition, show = "genotypes")
```
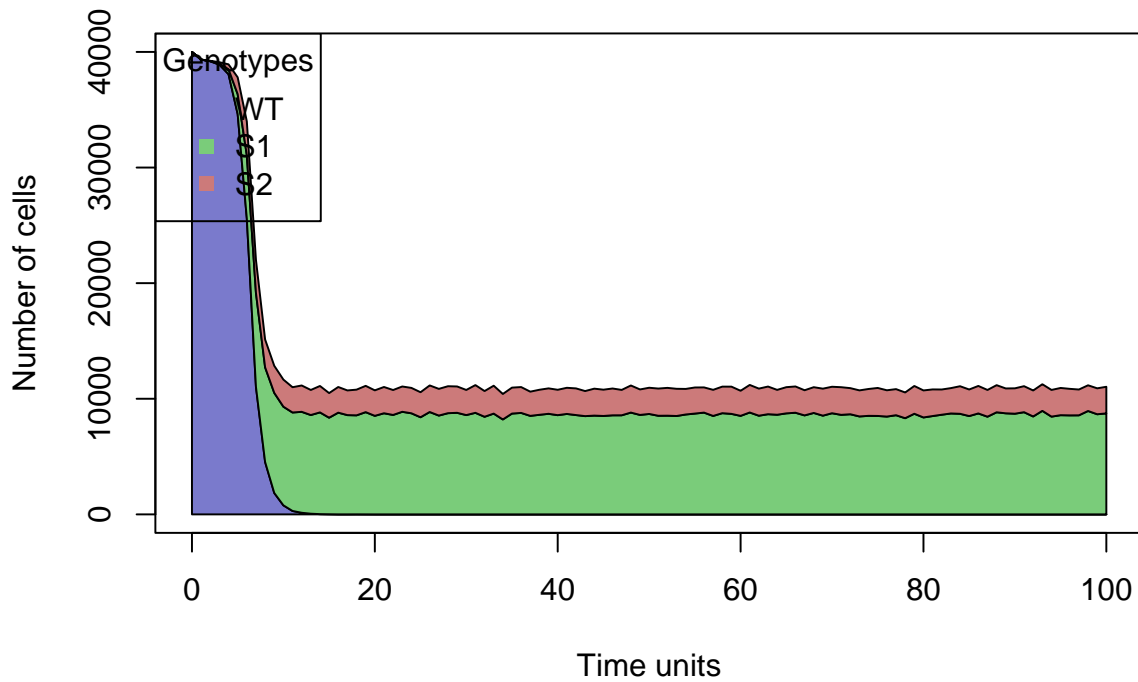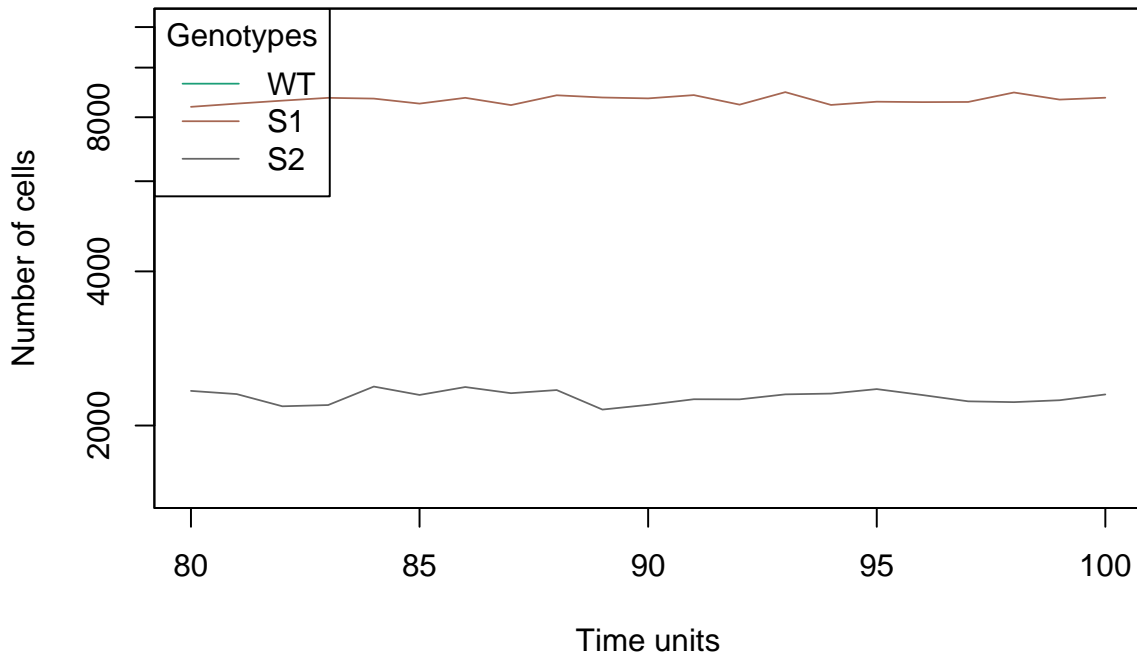
but we can avoid that by showing the plot after the WT are long gone:

```
plot(competition, show = "genotypes",
     xlim = c(80, 100))
```



```
plot(competition, show = "genotypes", type = "line",
     xlim = c(80, 100), ylim = c(1500, 12000))
```

10.4.2 Competition

We repeat the above, but starting directly from the two species, using the 6.8 logic and 6.7.

```r
G_fe_LVm <- function(r1, r2, K1, K2, a_12, a_21, awt = 1e-4,
                                 gt = c("S1", "S2")) {
    data.frame(Genotype = gt,
               Fitness = c(
                   paste0("1 + ", r1,
                          " * ( 1 - (n_1 + ", a_12, " * n_2)/", K1,
                          ")"),
                   paste0("1 + ", r2,
                          " * ( 1 - (n_2 + ", a_21, " * n_1)/", K2,
                          ")")
               ))
}


## Show expressions for birth rates
G_fe_LVm("r1", "r2", "K1", "K2", "a_12", "a_21", "awt")
##   Genotype                                Fitness
## 1       S1 1 + r1 * ( 1 - (n_1 + a_12 * n_2)/K1)
## 2       S2 1 + r2 * ( 1 - (n_2 + a_21 * n_1)/K2)
```

```r
fe_competitionm <-
    allFitnessEffects(
        genotFitness =
            G_fe_LVm(1.5, 1.4, 10000, 4000, 0.6, 0.2,
                gt = c("S1", "S2")),
```

```
        frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = G_fe_LVm(1.5, 1.4,
## 10000, 4000, : v2 functionality detected. Adapting to v3
## functionality.
```

```
fe_competitionm$full_FDF_spec
##   S1 S2 Genotype_as_numbers Genotype_as_letters Genotype_as_fvarsb
## 1  1  0                   1                  S1                 n_1
## 2  0  1                   2                  S2                 n_2
##                    Fitness_as_fvars
## 1 1 + 1.5 * ( 1 - (n_1 + 0.6 * n_2)/10000)
## 2   1 + 1.4 * ( 1 - (n_2 + 0.2 * n_1)/4000)
##                   Fitness_as_letters
## 1 1 + 1.5 * ( 1 - (n_1 + 0.6 * n_2)/10000)
## 2   1 + 1.4 * ( 1 - (n_2 + 0.2 * n_1)/4000)
```

```
competitionm <- oncoSimulIndiv(fe_competitionm,
                    model = "Exp",
                    initMutant = c("S1", "S2"),
                    initSize = c(5000, 2000),
                    onlyCancer = FALSE,
                    finalTime = 100,
                    mu = 1e-4,
                    keepPhylog = FALSE,
                    seed = NULL,
                    errorHitMaxTries = FALSE,
                    errorHitWallTime = FALSE)
```

```
plot(competitionm, show = "genotypes")
```
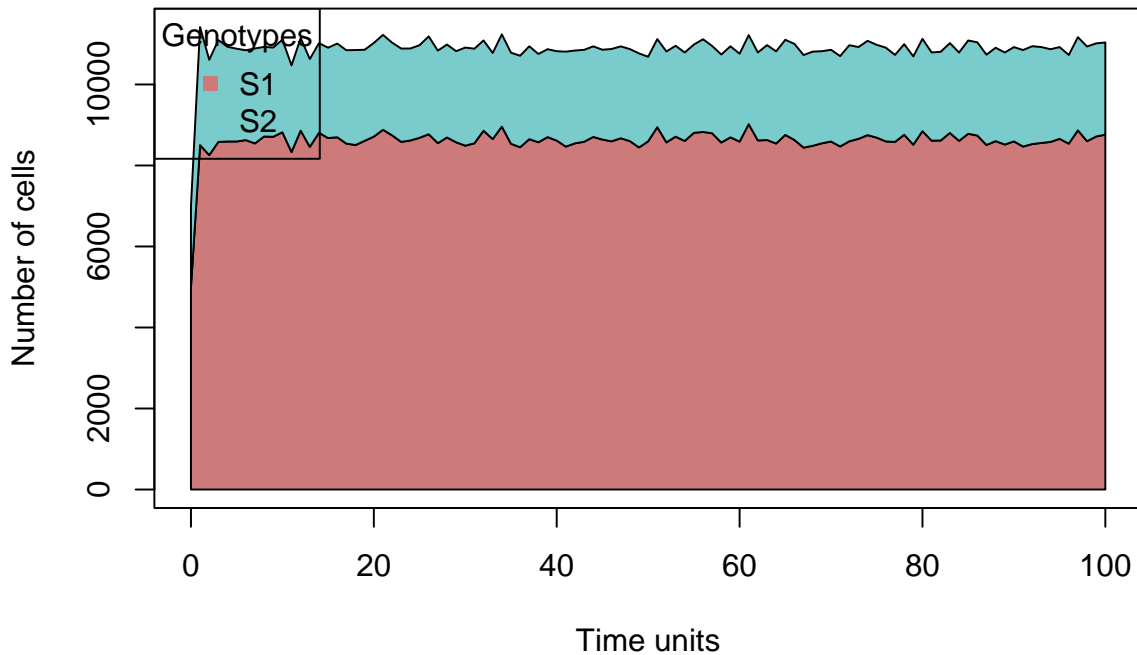
10.4.3 Predator-prey, first example

The simplest model would use the above Lotka-Volterra expressions and set one of the a_ to be negative (see, for example, Otto & Day, 2007, p. 73). Let's turn former "S2" and thus a_21 < 0 (to make the effects more salient, we also increase that value in magnitude).

We will also use a general function to generate fitness expressions. This is actually nicer than we did above, because it allows us to give the names of the species, not codes such as "n_1" that depend on how the names are ordered by R.

```r
G_fe_LVm2 <- function(r1, r2, K1, K2, a_12, a_21, awt = 1e-4,
                      gt = c("S1", "S2")) {
    data.frame(Genotype = gt,
               Fitness = c(
                   paste0("1 + ", r1,
                          " * ( 1 - (n_", gt[1], " + ", a_12, " * n_", gt[2], ")/",
                          ")"),
                   paste0("1 + ", r2,
                          " * ( 1 - (n_", gt[2], " + ", a_21, " * n_", gt[1], ")/",
                          ")")
               ))
}


## But notice that, because of ordering, "prey" ends up being n_2
## but that is not a problem.

fe_pred_preym2 <-
    allFitnessEffects(
```

```
        genotFitness =
            G_fe_LVm2(1.5, 1.4, 10000, 4000, 1.1, -0.5, awt = 1,
                gt = c("prey", "Predator")),
        frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = G_fe_LVm2(1.5, 1.4,
## 10000, 4000, : v2 functionality detected. Adapting to v3
## functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
```

```
fe_pred_preym2$full_FDF_spec
##   Predator prey Genotype_as_numbers Genotype_as_letters
## 1        0    1                   2                prey
## 2        1    0                   1            Predator
##   Genotype_as_fvarsb                          Fitness_as_fvars
## 1                n_2 1 + 1.5 * ( 1 - (n_2 + 1.1 * n_1)/10000)
## 2                n_1 1 + 1.4 * ( 1 - (n_1 + -0.5 * n_2)/4000)
##                                          Fitness_as_letters
## 1 1 + 1.5 * ( 1 - (n_prey + 1.1 * n_Predator)/10000)
## 2 1 + 1.4 * ( 1 - (n_Predator + -0.5 * n_prey)/4000)
```

```
## Change order and note how these are, of course, equivalent

fe_pred_preym3 <-
    allFitnessEffects(
        genotFitness =
            G_fe_LVm2(1.4, 1.5, 4000, 10000, -0.5, 1.1, awt = 1,
                gt = c("Predator", "prey")),
        frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = G_fe_LVm2(1.4, 1.5,
## 4000, 10000, : v2 functionality detected. Adapting to v3
## functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
```

```
fe_pred_preym3$full_FDF_spec
##   Predator prey Genotype_as_numbers Genotype_as_letters
## 1        1    0                   1            Predator
## 2        0    1                   2                prey
##   Genotype_as_fvarsb                          Fitness_as_fvars
## 1                n_1 1 + 1.4 * ( 1 - (n_1 + -0.5 * n_2)/4000)
## 2                n_2 1 + 1.5 * ( 1 - (n_2 + 1.1 * n_1)/10000)
##                                          Fitness_as_letters
## 1 1 + 1.4 * ( 1 - (n_Predator + -0.5 * n_prey)/4000)
## 2 1 + 1.5 * ( 1 - (n_prey + 1.1 * n_Predator)/10000)
```

```
evalAllGenotypes(fe_pred_preym2, spPopSizes = c(1000, 300))
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
##        Genotype Fitness
## 1            WT    0.00
## 2      Predator    2.47
## 3          prey    2.30
## 4 Predator, prey    0.00
```

```
evalAllGenotypes(fe_pred_preym3, spPopSizes = c(300, 1000))
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
##        Genotype Fitness
## 1            WT    0.00
## 2      Predator    2.47
## 3          prey    2.30
## 4 Predator, prey    0.00
```

```
s_pred_preym2 <- oncoSimulIndiv(fe_pred_preym2,
                                model = "Exp",
                                initMutant = c("prey", "Predator"),
                                initSize = c(1000, 1000),
                                onlyCancer = FALSE,
                                finalTime = 200,
                                mu = 1e-3,
                                keepPhylog = FALSE,
                                seed = NULL,
                                errorHitMaxTries = FALSE,
                                errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##          to last version.
```
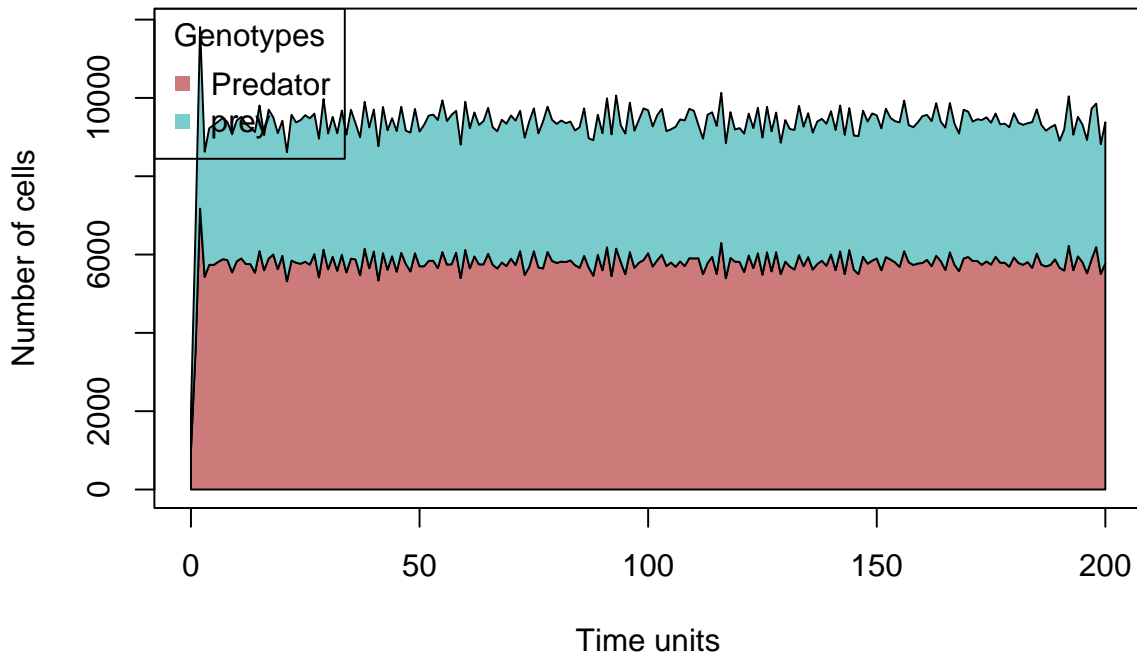
```
plot(s_pred_preym2, show = "genotypes")
```

You can easily play with a range of parameters, say the carrying capacity of one of the species, to see how they affect the stochasticity of the system.

If you run the above model repeatedly, you will frequently find that only one of the species is left; and, yes, that could be the "Predator", as in the Lotka-Volterra expressions above there can be predators without prey. For example, you can check that the birth rate of the predator is larger than 1 even if there are 0 prey (identically 1 when n_1 = K_1:

```
evalAllGenotypes(fe_pred_preym2, spPopSizes = c(0, 300))
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##         Genotype Fitness
## 1             WT   0.000
## 2       Predator   2.295
## 3           prey   2.450
## 4 Predator, prey   0.000
```

```
evalAllGenotypes(allFitnessEffects(
    genotFitness =
        G_fe_LVm2(1.5, 1.4, 100, 40,
                  0.6, -0.5, awt = 0.1,
                  gt = c("prey", "Predator")),
    frequencyDependentFitness = TRUE),
    spPopSizes = c(0, 40))
```

249

```
## Warning in allFitnessEffects(genotFitness = G_fe_LVm2(1.5, 1.4,
## 100, 40, : v2 functionality detected. Adapting to v3 functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
##         Genotype Fitness
## 1            WT    0.00
## 2      Predator    1.00
## 3          prey    2.14
## 4 Predator, prey    0.00
```

10.4.4 Predator-prey, second example

We use now the model in p. 76 of Otto & Day (2007), where prey grow exponentially in the absence of predators (and predators will eventually go extinct in the absence of prey):

$$\frac{\mathrm{d}n_1}{\mathrm{d}t} = r\ n_1 - a\ c\ n_1\ n_2$$

$$\frac{\mathrm{d}n_2}{\mathrm{d}t} = \epsilon\ a\ c\ n_1\ n_2 - \delta\ n2$$

(Recall what we explained in section 10.4.1 for how we find the b, birth rate, to use in our simulations when we are using and "Exp" model with death rate 1: basically, each of the birth rates, b_i is $1 + expression\ above/n_i$).

```
C_fe_pred_prey2 <- function(r, a, c, e, d,
                          gt = c("s1", "s2")) {
   data.frame(Genotype = gt,
             Fitness = c(
                paste0("1 + ", r, " - ", a,
                      " * ", c, " * n_2"),
                paste0("1 + ", e, " * ", a,
                      " * ", c, " * n_1 - ", d)
             ))
}


C_fe_pred_prey2("r", "a", "c", "e", "d")
##   Genotype                Fitness
## 1       s1    1 + r - a * c * n_2
## 2       s2 1 + e * a * c * n_1 - d
```

Given how we wrote `C_fe_pred_prey2`, the prey is hardcoded as `n_1`, so specify names of creatures so that the prey comes first, in terms of order (note we avoided this problem in the example above, 10.4.3, by always using the full name of the genotype we refered to in the function to generate the fitness effects, `G_fe_LVm2`). (Yes, we could have used a classic pair: "Hare" and "Lynx").

```
fe_pred_prey2 <-
    allFitnessEffects(
        genotFitness =
            C_fe_pred_prey2(r = .7, a = 1, c = 0.005,
                            e = 0.02, d = 0.4,
                            gt = c("Fly", "Lizard")),
        frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = C_fe_pred_prey2(r =
## 0.7, a = 1, : v2 functionality detected. Adapting to v3
## functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
```

```
fe_pred_prey2$full_FDF_spec
##   Fly Lizard Genotype_as_numbers Genotype_as_letters
## 1   1      0                   1                 Fly
## 2   0      1                   2              Lizard
##   Genotype_as_fvarsb              Fitness_as_fvars
## 1                n_1        1 + 0.7 - 1 * 0.005 * n_2
## 2                n_2 1 + 0.02 * 1 * 0.005 * n_1 - 0.4
##              Fitness_as_letters
## 1        1 + 0.7 - 1 * 0.005 * n_2
## 2 1 + 0.02 * 1 * 0.005 * n_1 - 0.4
```

```
## You want to make sure you start the simulation from
## a viable condition
```

```
evalAllGenotypes(fe_pred_prey2,
                 spPopSizes = c(5000, 100))
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
##     Genotype Fitness
## 1         WT     0.0
## 2        Fly     1.2
## 3     Lizard     1.1
```

```
## 4 Fly, Lizard     0.0
```

```
set.seed(2)
pred_prey2 <- oncoSimulIndiv(fe_pred_prey2,
                             model = "Exp",
                             initMutant = c("Fly", "Lizard"),
                             initSize = c(500, 100),
                             sampleEvery = 0.1,
                             mu = 1e-3,
                             onlyCancer = FALSE,
                             finalTime = 100,
                             keepPhylog = FALSE,
                             seed = NULL,
                             errorHitMaxTries = FALSE,
                             errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
```

```
op <- par(mfrow = c(1, 2))
## Nicer colors
plot(pred_prey2, show = "genotypes")
## But this shows better what is going on
plot(pred_prey2, show = "genotypes", type = "line")
```
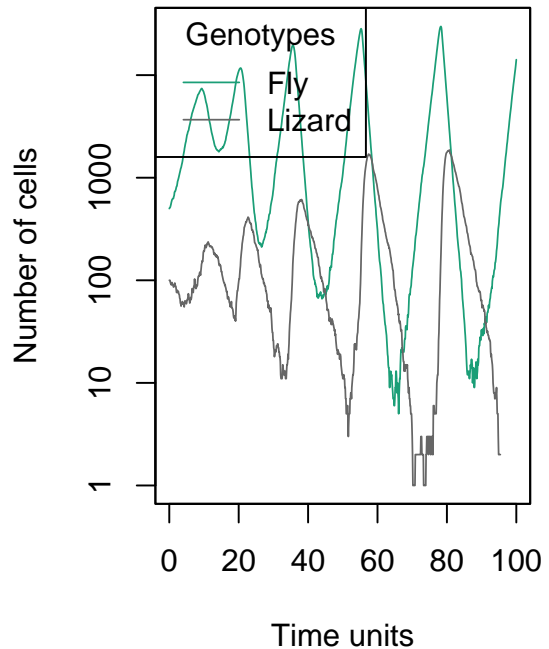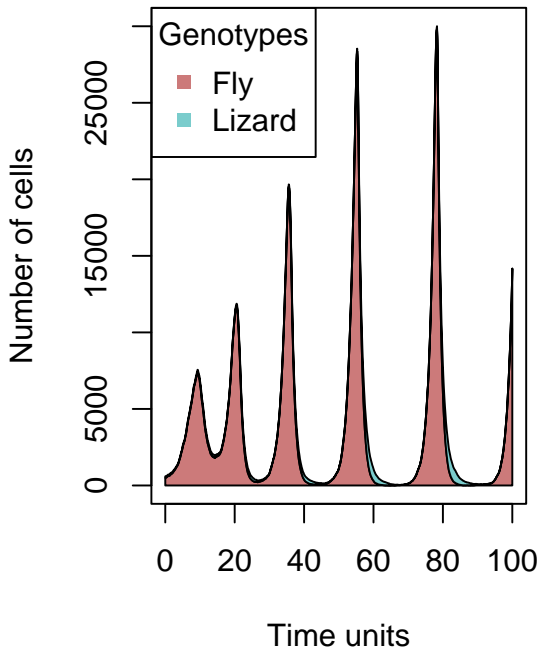


```
par(op)
```

If you run that model repeatedly, sometimes the system will go extinct quickly, or

you will only get prey growing exponentially.

You could now (left as an exercise) build a more complex model to simulate arms-race scenarios between predators and prey (maybe by having mutations with possibly opposing effects on different coefficients above).

10.4.5 Commensalism

Modelling commensalism simply requires changing the values of the α, the `a_12` and `a_21`. Again, we can now avoid starting from a WT and start the simulation directly from "A" and "Commensal" (section 6.7).

For example (not run, as this is just repetitive):

```r
fe_commens <-
    allFitnessEffects(
        genotFitness =
            G_fe_LV(1.2, 1.3, 5000, 20000,
                                0, -0.2,
                                gt = c("WT","A", "Commensal")),
        frequencyDependentFitness = TRUE,
        frequencyType = "abs")


commens <- oncoSimulIndiv(fe_commens,
                            model = "Exp",
                            onlyCancer = FALSE,
                            finalTime = 100,
                            mu = 1e-4,
                            initSize = 40000,
                          keepPhylog = FALSE,
                          seed = NULL,
                            errorHitMaxTries = FALSE,
                            errorHitWallTime = FALSE)

plot(commens, show = "genotypes")

plot(commens, show = "genotypes",
     xlim = c(80, 100))

plot(commens, show = "genotypes", type = "line",
     xlim = c(80, 100), ylim = c(2000, 22000))
```

10.5 Frequency-dependent fitness: can I mix relative and absolute frequencies?

Yes, of course, since you can always use an absolute specification with the appropriate quotient. For example, the following two specifications are identical:

```
rar <- data.frame(Genotype = c("WT", "A", "B", "C"),
                  Fitness = c("1",
                              "1.1 + .3*f_2",
                              "1.2 + .4*f_1",
                              "1.0 + .5 * (f_1 + f_2)"))
afear <- allFitnessEffects(genotFitness = rar,
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = rar,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
evalAllGenotypes(afear, spPopSizes = c(100, 200, 300, 400))
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
##    Genotype Fitness
## 1        WT    1.00
## 2         A    1.19
## 3         B    1.28
## 4         C    1.25
## 5      A, B    0.00
## 6      A, C    0.00
## 7      B, C    0.00
## 8   A, B, C    0.00
```

```
rar2 <- data.frame(Genotype = c("WT", "A", "B", "C"),
                   Fitness = c("1",
                               "1.1 + .3*(n_2/N)",
                               "1.2 + .4*(n_1/N)",
                               "1.0 + .5 * ((n_1 + n_2)/N)"))
afear2 <- allFitnessEffects(genotFitness = rar2,
                            frequencyDependentFitness = TRUE,
                            frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = rar2,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
evalAllGenotypes(afear2, spPopSizes = c(100, 200, 300, 400))
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
##    Genotype Fitness
## 1        WT    1.00
## 2         A    1.19
## 3         B    1.28
```

```
## 4        C     1.25
## 5     A, B     0.00
## 6     A, C     0.00
## 7     B, C     0.00
## 8  A, B, C     0.00
```

and simulating with them leads to identical results

```
set.seed(1)
tmp1 <- oncoSimulIndiv(afear,
                       model = "McFL",
                       onlyCancer = FALSE,
                       finalTime = 30,
                       mu = 1e-4,
                       initSize = 5000,
                       keepPhylog = FALSE,
                       seed = NULL,
                       errorHitMaxTries = FALSE,
                       errorHitWallTime = FALSE)

set.seed(1)
tmp2 <- oncoSimulIndiv(afear2,
                       model = "McFL",
                       onlyCancer = FALSE,
                       finalTime = 30,
                       mu = 1e-4,
                       initSize = 5000,
                       keepPhylog = FALSE,
                       seed = NULL,
                       errorHitMaxTries = FALSE,
                       errorHitWallTime = FALSE)
stopifnot(identical(print(tmp1), print(tmp2)))
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = afear, model = "McFL", mu = 1e-04, initSize = 5000,
##     finalTime = 30, onlyCancer = FALSE, keepPhylog = FALSE, errorHitWallTime =
##     errorHitMaxTries = FALSE, seed = NULL)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         4         5072         4748             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0        30    1250
##   HittedWallTime HittedMaxTries   errorMF minDMratio minBMratio
## 1          FALSE          FALSE  0.006621       3220       3333
##   OccurringDrivers
## 1
##
```

255

```
## Final population composition:
##   Genotype    N
## 1          4748
## 2        A   46
## 3        B  273
## 4        C    5
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = afear2, model = "McFL", mu = 1e-04, initSize = 5000,
##     finalTime = 30, onlyCancer = FALSE, keepPhylog = FALSE, errorHitWallTime =
##     errorHitMaxTries = FALSE, seed = NULL)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         4         5072         4748             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0        30    1250
##   HittedWallTime HittedMaxTries   errorMF minDMratio minBMratio
## 1          FALSE          FALSE 0.006621       3220       3333
##   OccurringDrivers
## 1
##
## Final population composition:
##   Genotype    N
## 1          4748
## 2        A   46
## 3        B  273
## 4        C    5
```

So you can always mix relative and absolute; here fitness of two genotypes depends on the relative frequencies of others, whereas fitness of the third on the absolute frequencies (number of cells):

```
rar3 <- data.frame(Genotype = c("WT", "A", "B", "C"),
                Fitness = c("1",
                            "1.1 + .3*(n_2/N)",
                            "1.2 + .4*(n_1/N)",
                            "1.0 + .5 * ( n_1 > 20)"))
afear3 <- allFitnessEffects(genotFitness = rar3,
                        frequencyDependentFitness = TRUE,
                        frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = rar3,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.

evalAllGenotypes(afear3, spPopSizes = c(100, 200, 300, 400))
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
```
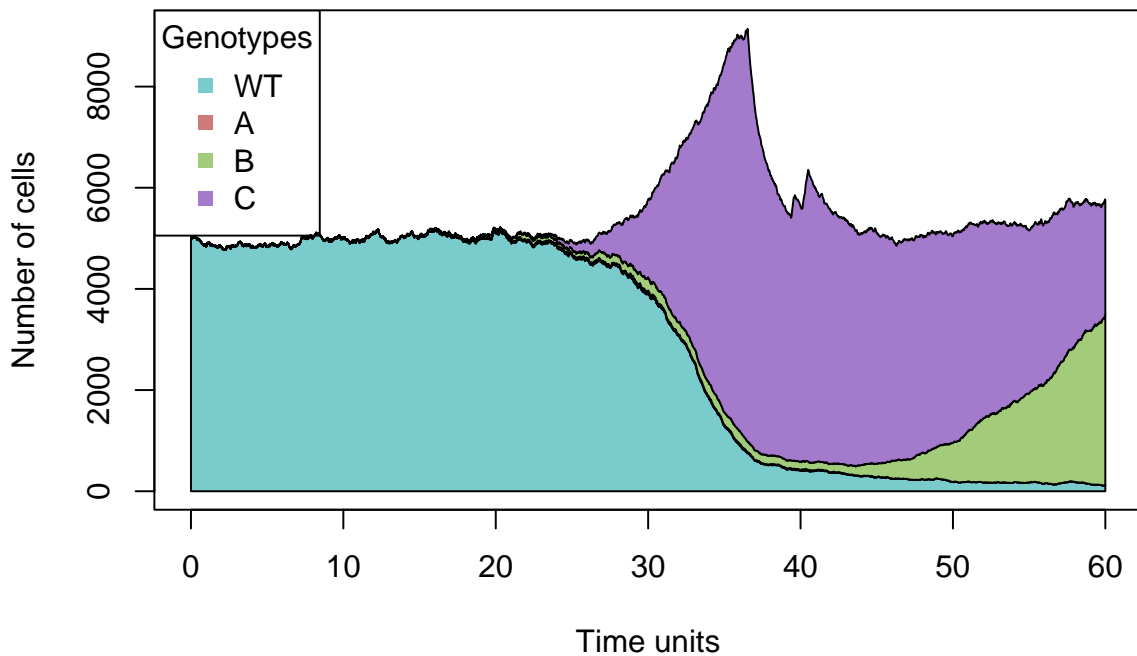
```
## unnamed: cannot check genotype names.
##    Genotype Fitness
## 1        WT    1.00
## 2         A    1.19
## 3         B    1.28
## 4         C    1.50
## 5      A, B    0.00
## 6      A, C    0.00
## 7      B, C    0.00
## 8   A, B, C    0.00
```

```
set.seed(1)
tmp3 <- oncoSimulIndiv(afear3,
                       model = "McFL",
                       onlyCancer = FALSE,
                       finalTime = 60,
                       mu = 1e-4,
                       initSize = 5000,
                       keepPhylog = FALSE,
                       seed = NULL,
                       errorHitMaxTries = FALSE,
                       errorHitWallTime = FALSE)
plot(tmp3, show = "genotypes")
```

10.6 Frequency-dependent fitness: can I use genes with mutator effects?

Yes. The following examples show it:

```
## Relative
r1fd <- data.frame(Genotype = c("WT", "A", "B", "A, B"),
                   Fitness = c("1",
                               "1.4 + 1*(f_2)",
                               "1.4 + 1*(f_1)",
                               "1.6 + f_1 + f_2"))
afe4 <- allFitnessEffects(genotFitness = r1fd,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = r1fd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
mtfd <- allMutatorEffects(epistasis = c("A" = 0.1,
                                        "B" = 10))
```

```
set.seed(1)
s1fd <- oncoSimulIndiv(afe4,
                       model = "McFL",
                       onlyCancer = FALSE,
                       finalTime = 40,
                       mu = 1e-4,
                       initSize = 5000,
                       keepPhylog = TRUE,
                       seed = NULL,
                       errorHitMaxTries = FALSE,
                       errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
plot(s1fd, show = "genotypes")
```
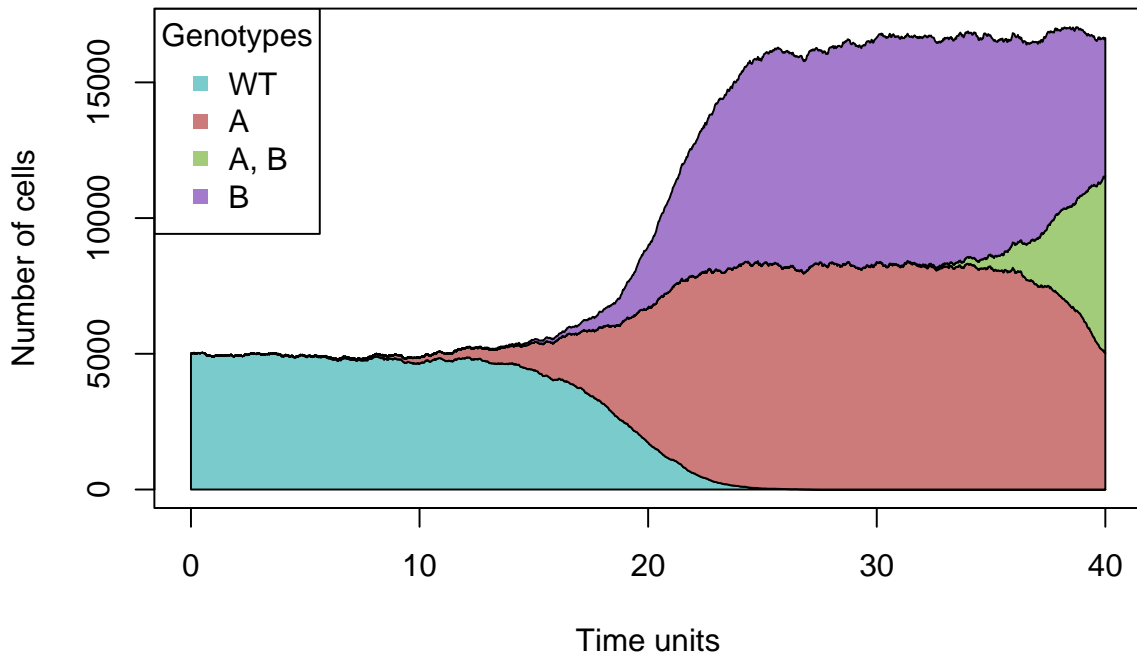
```r
set.seed(1)
s2fd <- oncoSimulIndiv(afe4,
                       muEF = mtfd,
                       model = "McFL",
                       onlyCancer = FALSE,
                       finalTime = 40,
                       mu = 1e-4,
                       initSize = 5000,
                       keepPhylog = TRUE,
                       seed = NULL,
                       errorHitMaxTries = FALSE,
                       errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.

## In the Mac ARM64 architecture, the above
## run leads to an exception, which is really odd.
## While that is debugged, use try to prevent
## failure of the plot to abort vignette building.

try(plot(s2fd, show = "genotypes"))
```
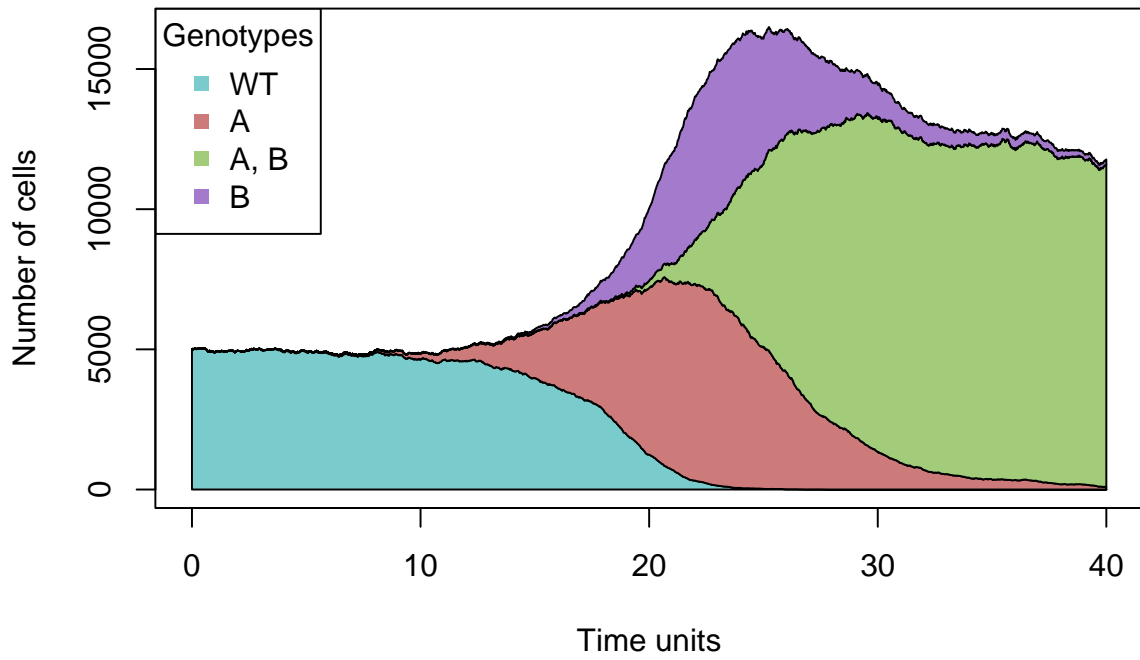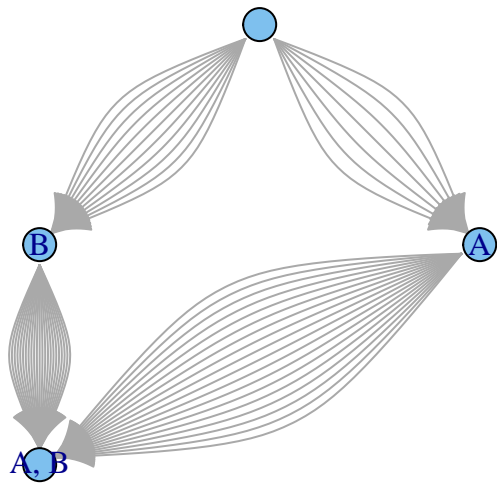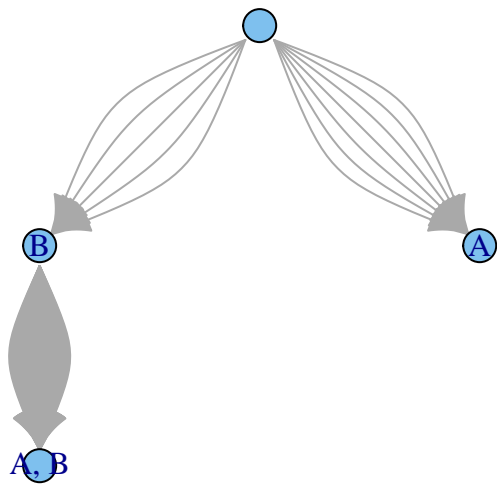
```
plotClonePhylog(s1fd, keepEvents = TRUE)
```



```
plotClonePhylog(s2fd, keepEvents = TRUE)
```

Of course, it also works with absolute frequencies (code not executed for the sake of speed):

```
## Absolute
r5 <- data.frame(Genotype = c("WT", "A", "B", "A, B"),
                 Fitness = c("1",
                             "1.25 - .0025*(n_2)",
                             "1.25 - .0025*(n_1)",
                             "1.4"),
                 stringsAsFactors = FALSE)

afe5 <- allFitnessEffects(genotFitness = r5,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "abs")
set.seed(8)
s5 <- oncoSimulIndiv(afe5,
                     model = "McFL",
                     onlyCancer = FALSE,
                     finalTime = 100,
                     mu = 1e-4,
                     initSize = 5000,
                     keepPhylog = TRUE,
                     seed = NULL,
                     errorHitMaxTries = FALSE,
                     errorHitWallTime = FALSE)
plot(s5, show = "genotypes")
plot(s5, show = "genotypes", log = "y", type = "line")

mt <- allMutatorEffects(epistasis = c("A" = 0.1,
                                      "B" = 10))
set.seed(8)
s6 <- oncoSimulIndiv(afe5,
                     muEF = mt,
                     model = "McFL",
                     onlyCancer = FALSE,
                     finalTime = 100,
                     mu = 1e-4,
                     initSize = 5000,
                     keepPhylog = TRUE,
                     seed = NULL,
                     errorHitMaxTries = FALSE,
                     errorHitWallTime = FALSE)
plot(s6, show = "genotypes")
plot(s6, show = "genotypes", log = "y", type = "line")

plotClonePhylog(s5, keepEvents = TRUE)
```

```r
plotClonePhylog(s6, keepEvents = TRUE)
```

Note that **evalAllGenotypesFitAndMut** currently works with frequency-dependent fitness:

```r
evalAllGenotypes(allFitnessEffects(genotFitness = r1fd,
                                   frequencyDependentFitness = TRUE,
                                   frequencyType = "rel"),
                 spPopSizes = c(10, 20, 30, 40))
## Warning in allFitnessEffects(genotFitness = r1fd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 1       WT     1.0
## 2        A     1.7
## 3        B     1.6
## 4     A, B     2.1
```

```r
evalAllGenotypesFitAndMut(allFitnessEffects(genotFitness = r1fd,
                                   frequencyDependentFitness = TRUE,
                                   frequencyType = "rel"),
                          mtfd,
                          spPopSizes = c(10, 20, 30, 40))
## Warning in allFitnessEffects(genotFitness = r1fd,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness MutatorFactor
## 1       WT     1.0           1.0
## 2        A     1.7           0.1
## 3        B     1.6          10.0
## 4     A, B     2.1           1.0
```

10.7 Can we use the BNB algorithm to model frequency-dependent fitness?

This question is similar to the one we address in 18.6. Briefly, the answer is yes. You can think of this as an approximation to an exact simulation of a stochastic system. You can also think of a delay in the system in the sense that the changes in rates due to changes in the frequencies of the different genotypes are updated at periodic intervals, not immediately.

11 Additional examples of frequency-dependent fitness

In this section, we provide additional examples that use frequency-dependent fitness. As mentioned also in 10,

Note also that in most of these examples we make rather arbitrary and simple assumptions about the genetic basis of the different phenotypes or strategies (most are one-mutation-away from WT); see 10.4 and 10.2 (where we change the ancestor-dependent relationships). In some examples mutation rates are also very high, to speed up processes and because a high mutation rate is used as a procedure (a hack?) to quickly obtain descendants from WT (i.e., to get the game started with some representatives of the non-WT types).

Examples 11.1, 11.2, 11.3, 11.4, 11.5 were originally prepared by Sara Dorado Alfaro, Miguel Hernández del Valle, Álvaro Huertas García, Diego Mañanes Cayero, Alejandro Martín Muñoz; example 11.6 was originally prepared by Marta Couce Iglesias, Silvia García Cobos, Carlos Madariaga Aramendi, Ana Rodríguez Ronchel, and Lucía Sánchez García; examples 11.7, 11.8, 11.9 were prepared by Yolanda Benítez Quesada, Asier Fernández Pato, Esperanza López López, Alberto Manuel Parra Pérez. All of these as an exercisse for the course Programming and Statistics with R (Master's Degree in Bioinformatics and Computational Biology, Universidad Autónoma de Madrid), course 2019-20.

11.1 Rock-paper-scissors model in bacterial community

11.1.1 Introduction

This example is inspired by Kerr et al. (2002). It describes the relationship between three populations of *Escherichia coli*, that turns out to be very similar to a rock-paper-scissors game.

An E. coli community can have a specific strain of colicinogenic bacteria, that are capable of creating colicin, a toxin to which this special strain is resistant. The wild-type bacteria is killed by this toxin, but can mutate into a resistant strain.

So, there are three kinds of bacteria: wild-type (WT), colicinogenic (C) and resistant (R). The presence of C reduces the population of WT, but increases the population of R because R has an advantage over C, since R doesn't have the cost of creating

the toxin. At the same time, WT has an advantage over R, because by losing the toxin receptors, R loses also some important functions. Therefore, every strain "wins" against one strain and "loses" against the other, creating a rock-paper-scissors game.

```r
crs <- function (a, b, c){
  data.frame(Genotype = c("WT", "C", "R"),
             Fitness = c(paste0("1 + ", a, " * f_R - ", b, " * f_C"),
                         paste0("1 + ", b, " * f_ - ", c, " * f_R"),
                         paste0("1 + ", c, " * f_C - ", a, " * f_")
             ))
}
```

The equations are:

$$W\,(WT) = 1 + af_R - bf_C \tag{3}$$
$$W\,(C) = 1 + bf_{WT} - cf_R \tag{4}$$
$$W\,(R) = 1 + cf_C - af_{WT} \tag{5}$$
$$\tag{6}$$

where f_{WT}, f_C and f_R are the frequencies of WT, C and R, respectively.

```r
crs("a", "b", "c")
##   Genotype              Fitness
## 1       WT 1 + a * f_R - b * f_C
## 2        C 1 + b * f_ - c * f_R
## 3        R 1 + c * f_C - a * f_
```

11.1.1.1   Case 1   We are going to study the scenario in which all the relationships have the same relative weight.

```r
afcrs1 <- allFitnessEffects(genotFitness = crs(1, 1, 1),
                            frequencyDependentFitness = TRUE,
                            frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = crs(1, 1, 1),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.

resultscrs1 <- oncoSimulIndiv(afcrs1,
                              model = "McFL",
                              onlyCancer = FALSE,
                              finalTime = 100,
                              mu = 1e-2,
                              initSize = 4000,
                              keepPhylog = FALSE,
                              seed = NULL,
                              errorHitMaxTries = FALSE,
```
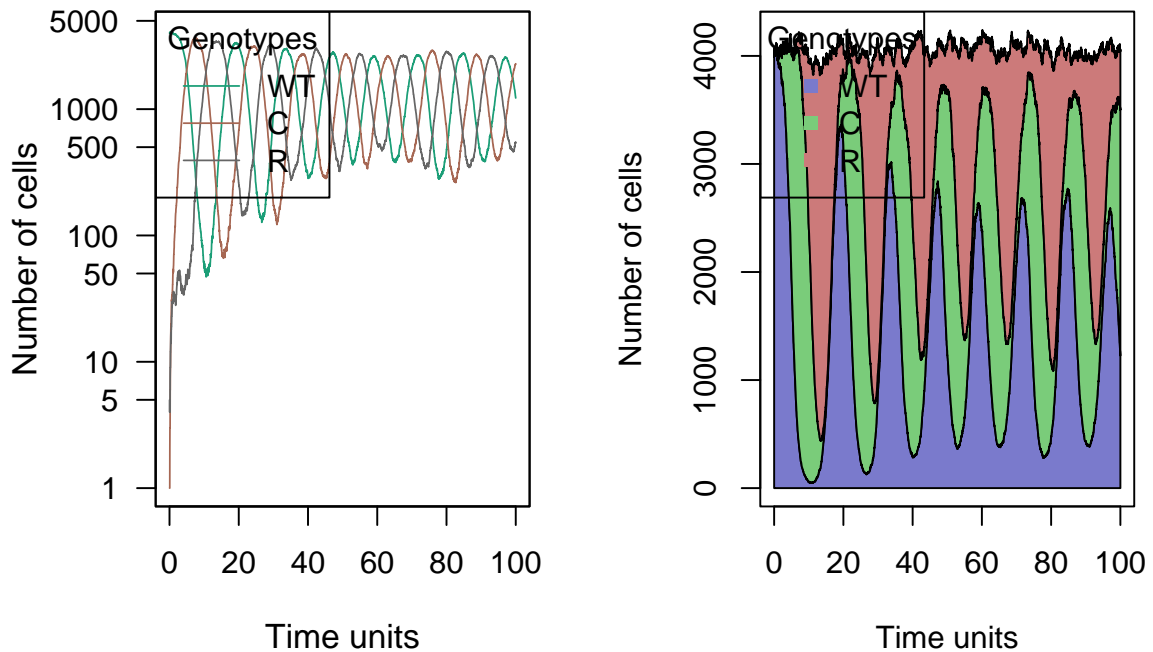
```
                              errorHitWallTime = FALSE)
op <- par(mfrow = c(1, 2))
plot(resultscrs1, show = "genotypes", type = "line", cex.lab=1.1,
     las = 1)
plot(resultscrs1, show = "genotypes", type = "stacked")
```



```
par(op)
```

An oscillatory equilibrium is reached, in which the same populations have a similar number of individuals but oscillates. This makes sense, because the rise on a particular strand will lead to a rise in the one that "wins" against it, and then to a rise in the one that "wins" against the second one, creating this cyclical behaviour. In the stacked plot we can see that the total population remains almost constant.

Note, though, that altering mutation rate (which is huge here) can change the results of the model.

11.1.1.2 Case 2 We are going to put a bigger weight in one of the coefficients, so a=10, b=1, c=1.

```
afcrs2 <- allFitnessEffects(genotFitness = crs(10, 1, 1),
                            frequencyDependentFitness = TRUE,
                            frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = crs(10, 1, 1),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

If we run multiple simulations, for example by doing

```
resultscrs2 <- oncoSimulPop(10,
                            afcrs2,
                               model = "McFL",
                               onlyCancer = FALSE,
                               finalTime = 100,
                               mu = 1e-2,
                               initSize = 4000,
                               keepPhylog = FALSE,
                               seed = NULL,
                               errorHitMaxTries = FALSE,
                               errorHitWallTime = FALSE)
```

we can verify there are two different scenarios.

The first one is the one in which all the strains coexist, with the colicinogenic bacteria having a much bigger population.

```
set.seed(1)

resultscrs2a <- oncoSimulIndiv(afcrs2,
                               model = "McFL",
                               onlyCancer = FALSE,
                               finalTime = 100,
                               mu = 1e-2,
                               initSize = 4000,
                               keepPhylog = FALSE,
                               seed = NULL,
                               errorHitMaxTries = FALSE,
                               errorHitWallTime = FALSE)

plot(resultscrs2a, show = "genotypes", type = "line")
```
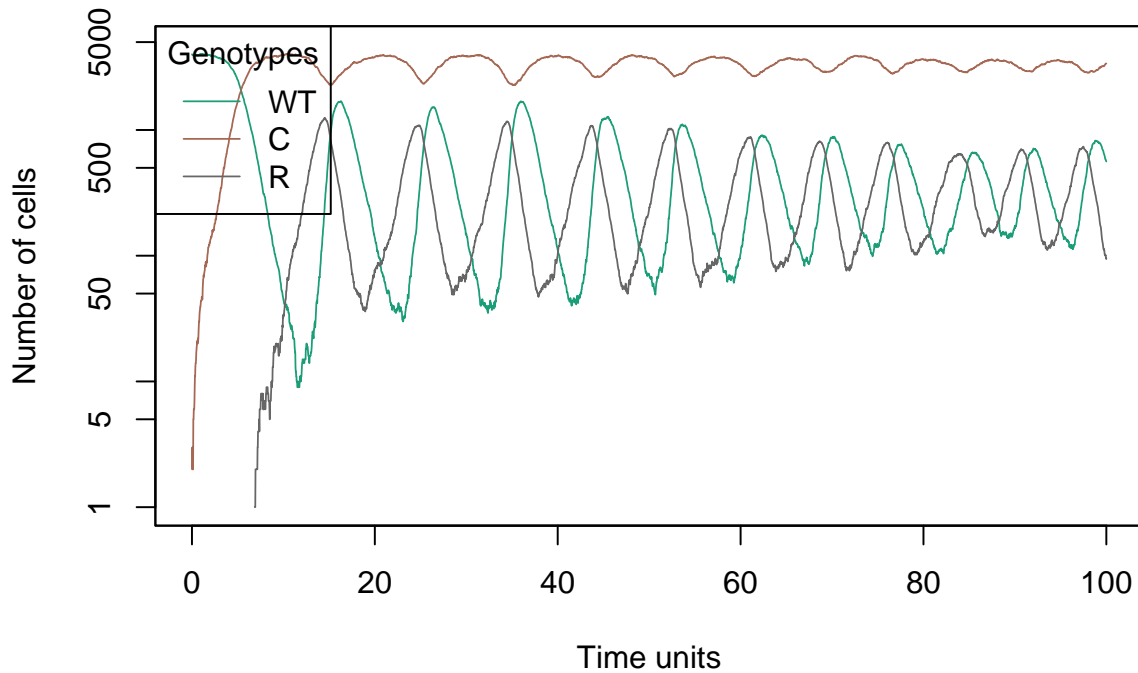
In the second one, the wild type and the colicinogenic bacteria dissapear, so the resistant strain is the only one that survives.

As above, though, decreasing the mutation rate can lead to a different solution and you will want to run the model for much longer to see the resistant strain appear and outcompete the others.

```
set.seed(3)

resultscrs2b <- oncoSimulIndiv(afcrs2,
                               model = "McFL",
                               onlyCancer = FALSE,
                               finalTime = 60,
                               mu = 1e-2,
                               initSize = 4000,
                               keepPhylog = FALSE,
                               seed = NULL,
                               errorHitMaxTries = FALSE,
                               errorHitWallTime = FALSE)

plot(resultscrs2b, show = "genotypes", type = "line", cex.lab=1.1,
     las = 1)
```
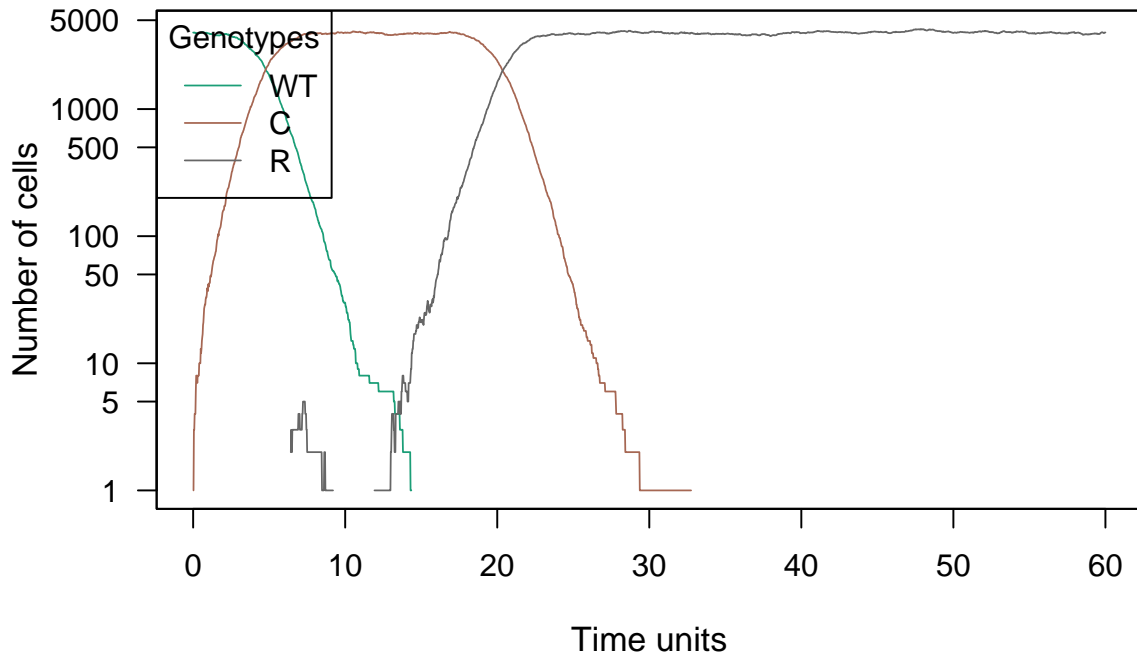
11.1.1.3 Case 3
Finally, we are going to put more weight in two coefficients, so a=1, b=5, c=5.

```
afcrs3 <- allFitnessEffects(genotFitness = crs(1, 5, 5),
                            frequencyDependentFitness = TRUE,
                            frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = crs(1, 5, 5),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
resultscrs3 <- oncoSimulIndiv(afcrs3,
                            model = "McFL",
                            onlyCancer = FALSE,
                            finalTime = 60,
                            mu = 1e-2,
                            initSize = 4000,
                            keepPhylog = FALSE,
                            seed = NULL,
                            errorHitMaxTries = FALSE,
                            errorHitWallTime = FALSE)

plot(resultscrs3, show = "genotypes", type = "line", cex.lab=1.1,
     las = 1)
```
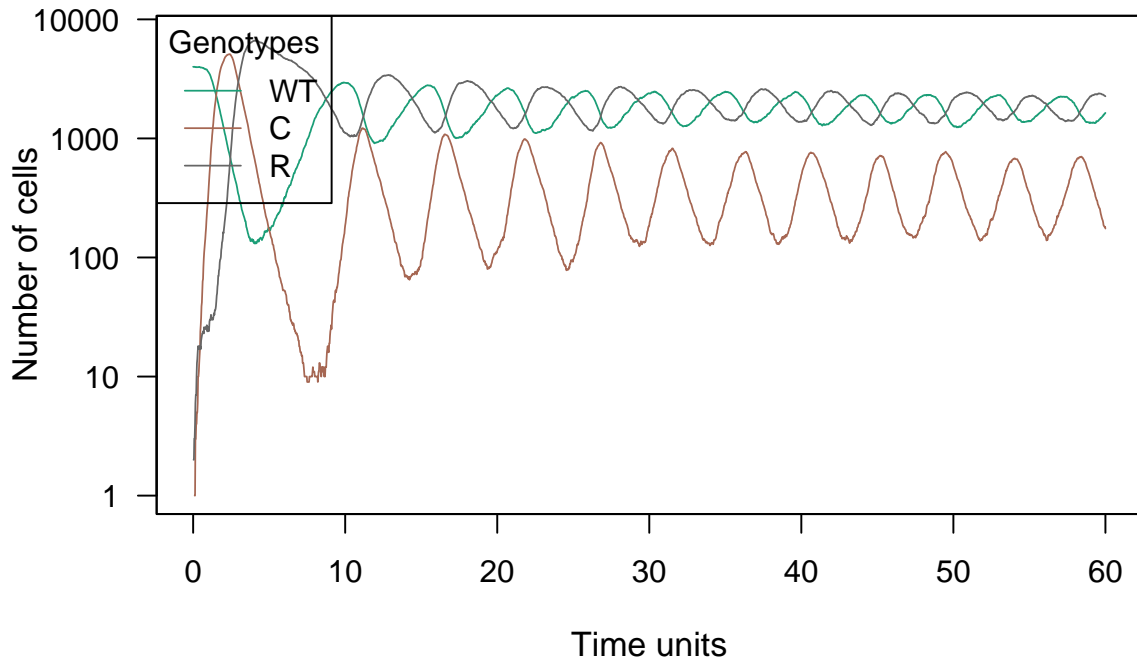
In all the cases all three strains survive, with C having a much smaller population than the other two.

11.2 Hawk and Dove example

The example we are going to show is one of the first games that Maynard Smith analyzed, for example in his classic Maynard Smith (1982) (see also, e.g., https://en.wikipedia.org/wiki/Chicken_%28game%29).

In this game, the two competitors are subtypes of the same species but with different strategies. The Hawk first displays aggression, then escalates into a fight until it either wins or is injured (loses). The Dove first displays aggression, but if faced with major escalation runs for safety. If not faced with such escalation, the Dove attempts to share the resource (see the payoff matrix, for instance in https://en.wikipedia.org/wiki/Chicken_%28game%29#Hawk%E2%80%93dove).

Given that the resource is given the value V, the damage from losing a fight is given cost C:

- If a Hawk meets a Dove he gets the full resource V to himself
- If a Hawk meets a Hawk – half the time he wins, half the time he loses. . . so his average outcome is then V/2 minus C/2
- If a Dove meets a Hawk he will back off and get nothing – 0
- If a Dove meets a Dove both share the resource and get V/2

The actual payoff however depends on the probability of meeting a Hawk or Dove, which in turn is a representation of the percentage of Hawks and Doves in the population when a particular contest takes place. That in turn is determined by the results of all of the previous contests. If the cost of losing C is greater than the value of winning V (the normal situation in the natural world) the mathematics ends in an

269

stationary point (ESS), a mix of the two strategies where the population of Hawks is
V/C.

In this case we assume a stable equilibrium in the population dynamics, that is,
although there are external variations in the model, it recovers and returns to
equilibrium.

We are going to simulate with OncoSimulR the situation in which the cost of losing
C is greater than the value of gaining V (C = 10, V = 2). We assume that both
Hawk and Dove are derived from WT by one mutation (see also 10.4) and we will
use very high mutation rates to get some hawks and doves from WT quickly (see
above).

Before performing the simulation, let's look at the fitness of each competitor.

```r
## Stablish Genotype-Fitnees mapping. D = Dove, H = Hawk

## With newer OncoSimulR functionality, using WT to start the simulation
## would no longer be needed.
H_D_fitness <- function(c, v,
                     gt = c("WT", "H", "D")) {
  data.frame(Genotype = gt,
            Fitness = c(
              paste0("1"),
              paste0("1 + f_H *", (v-c)/2, "+ f_D *", v),
              paste0("1 + f_D *", v/2)))
}


## Fitness Effects specification
HD_competition <-allFitnessEffects(
  genotFitness = H_D_fitness(10, 2,
                     gt = c("WT", "H", "D")),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = H_D_fitness(10, 2, gt =
## c("WT", : v2 functionality detected. Adapting to v3 functionality.


## Plot fitness landscape of genotype "H, D" evaluation
data.frame("Doves_fitness" = evalGenotype(genotype = "D",
                                       fitnessEffects = HD_competition,
                                       spPopSizes = c(5000, 5000, 5000)),
          "Hawks_fitness" = evalGenotype(genotype = "H",
                                       fitnessEffects = HD_competition,
                                       spPopSizes = c(5000, 5000, 5000))
          )
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
```

```
## unnamed: cannot check genotype names.
##   Doves_fitness Hawks_fitness
## 1         1.333        0.3333
```

We observe that the penalty of fighting (C > V) benefits the dove in terms of fitness respect to the hawk.

```
## Simulated trajectories
## run only a few for the sake of speed
simulation <- oncoSimulPop(2,
                           mc.cores = 2,
                           HD_competition,
                           model = "McFL", # There is no collapse
                           onlyCancer = FALSE,
                           finalTime = 50,
                           mu = 1e-2, # Quick emergence of D and H
                           initSize = 4000,
                           keepPhylog = FALSE,
                           seed = NULL,
                           errorHitMaxTries = FALSE,
                           errorHitWallTime = FALSE)



## Plot first trajectory as an example
plot(simulation[[1]], show = "genotypes", type = "line",
    xlim = c(40, 50),
     lwdClone = 2, ylab = "Number of individuals",
     main = "Hawk and Dove trajectory",
     col = c("#a37acc", "#f8776d", "#7daf00"),
     font.main=2, font.lab=2,
     cex.main=1.4, cex.lab=1.1,
     las = 1)
```

Hawk and Dove trajectory
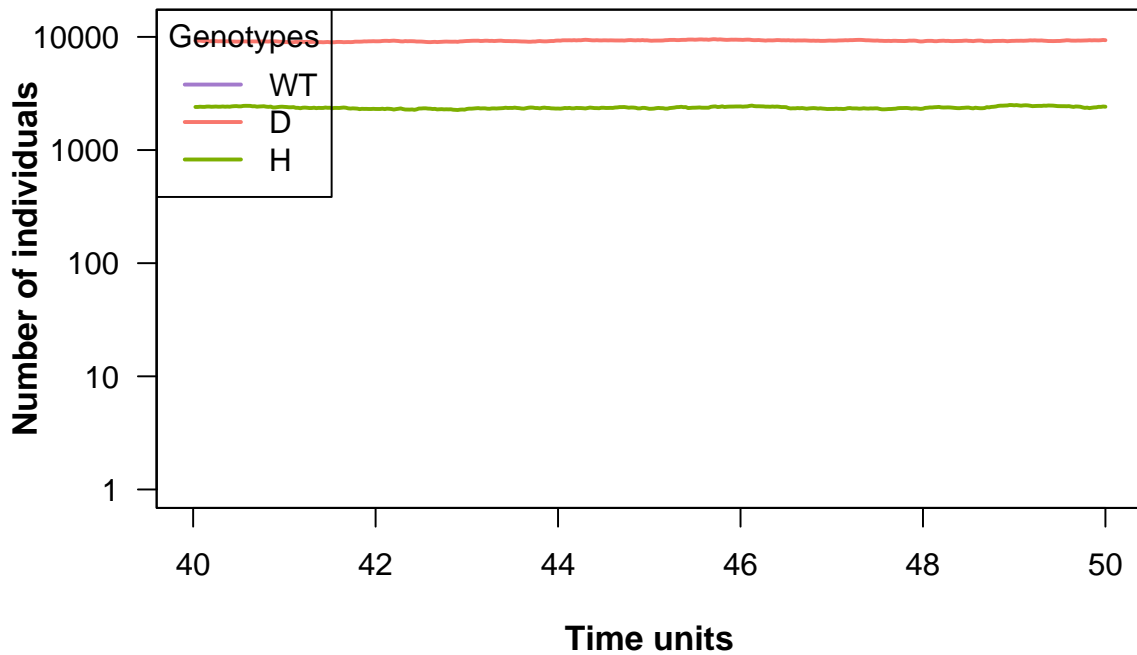
As mentioned above, mathematically when a stationary point (ESS) is reached the relative frequency of hawks is V/C and doves 1-(V/C). Considering f_H as relative frecuency of hawks and f_D = 1-f_H as frequency of doves:

Hawk:

$$1 + f_H * (v - c)/2 + (1 - f_H) * v$$

Dove:

$$1 + (1 - f_H) * v/2$$

Hawk = Dove:

$$1 + f_H * (v - c)/2 + (1 - f_H) * v = 1 + (1 - f_H) * v/2$$

Resolving for f_H:

$$f_H = v/c$$

Therefore, the relative frequency of hawks in equilibrium is equal to V/C. In our case it would be 20% (C = 10, V = 2). Let's check it:

```r
## Recover the final result from first simulation
result <- tail(simulation[[1]][[1]], 1)

## Get the number of organisms from each species
n_WT <- result[2]
n_D <- result[3]
n_H <- result[4]
total <- n_WT + n_D + n_H
```

```
## Dove percentage
data.frame("Doves" = round(n_D/total, 2)*100,
           "Hawks" = round(n_H/total, 2)*100  )
##   Doves Hawks
## 1    79    21
```

To sum up, this example shows that when the risks of contest injury or death (the Cost C) is significantly greater than the potential reward (the benefit value V), the stable population will be mixed between aggressors and doves, and the proportion of doves will exceed that of the aggressors. This explains behaviours observed in nature.

11.3 Game Theory with social dilemmas of tumour acidity and vasculature

This example is based on Kaznatcheev et al. (2017). In this work, it is explained that the progression of cancer is marked by the acquisition of a number of hallmarks, including self-sufficiency of growth factor production for angiogenesis and reprogramming energy metabolism for aerobic glycolysis. Moreover, there is evidence of intra-tumour heterogeneity. Given that some cancer cells can not invest in something that benefits the whole tumor while others can free-ride on the benefits created by them (evolutionary social dilemmas), how do these population level traits evolve, and how are they maintained? The authors answer this question with a mathematical model that treats acid production through glycolysis as a tumour-wide public good that is coupled to the club good of oxygen from better vascularisation.

The cell types of the model are:

- VOP: VEGF (over)-producers.
- GLY: glycolytic cells.
- DEF: aerobic cells that do not call for more vasculature.

On the other hand, the micro-environmental parameters of the model are:

- a: the benefit per unit of acidification.
- v: the benefit from oxygen per unit of vascularisation.
- c: the cost of (over)-producing VEGF.

The fitness equations derived from those populations and parameters are:

$$W\left(GLY\right) = 1 + a * (f_1 + 1) \tag{7}$$

$$W\left(VOP\right) = 1 + a * f_1 + v * (f_2 + 1) - c \tag{8}$$

$$W\left(DEF\right) = 1 + a * f_1 + v * f_2 \tag{9}$$

Where f_1 is the GLY cells' frequency and f_2 is the VOF cells' frequency at a given time. All fitness equations start from balance by the sum of 1.

Finally, depending of the parameter's values, the model can lead to three different situations (as in other examples, the different types are one mutation away from WT):

11.3.1 Fully glycolytic tumours:

If the fitness benefit of a single unit of acidification is higher than the maximum benefit from the club good for aerobic cells, then GLY cells will always have a strictly higher fitness than aerobic cells, and be selected for. In this scenario, the population will converge towards all GLY, regardless of the initial proportions (as long as there is at least some GLY in the population).

```
# Definition of the function for creating the corresponding dataframe.
avc <- function (a, v, c) {
  data.frame(Genotype = c("WT", "GLY", "VOP", "DEF"),
            Fitness = c("1",
                        paste0("1 + ",a," * (f_GLY + 1)"),
                        paste0("1 + ",a," * f_GLY + ",v," * (f_VOP + 1) - ",c),
                        paste0("1 + ",a," * f_GLY + ",v," * f_VOP")
                        ))
             }

# Specification of the different effects on fitness.
afavc <- allFitnessEffects(genotFitness = avc(2.5, 2, 1),
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = avc(2.5, 2, 1),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
## For real, you would probably want to run
## this multiple times with oncoSimulPop
simulation <- oncoSimulIndiv(afavc,
                          model = "McFL",
                          onlyCancer = FALSE,
                          finalTime = 15,
                          mu = 1e-3,
                          initSize = 4000,
                          keepPhylog = FALSE,
                          seed = NULL,
                          errorHitMaxTries = FALSE,
                          errorHitWallTime = FALSE)
```

```
# Representation of the plot of one simulation as an example (the others are
# highly similar).
plot(simulation, show = "genotypes", type = "line",
```

```
      ylab = "Number of individuals", main = "Fully glycolytic tumours",
      font.main=2, font.lab=2, cex.main=1.4, cex.lab=1.1, las = 1)
```

Fully glycolytic tumours
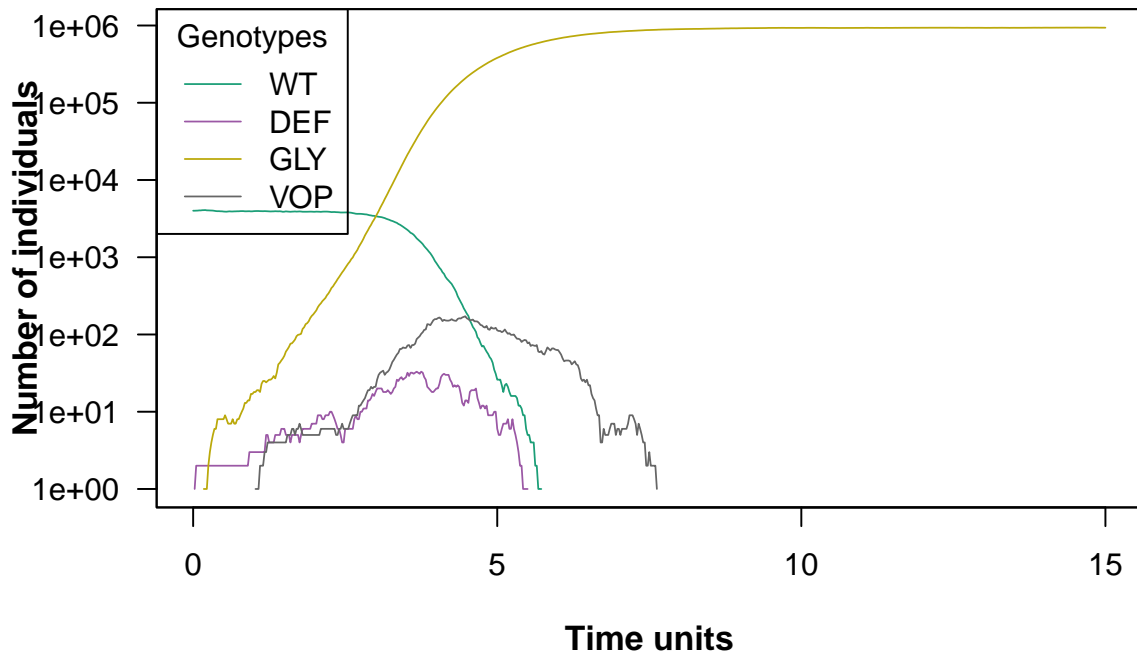
11.3.2   Fully angiogenic tumours:

If the benefit to VOP from their extra unit of vascularisation is higher than the cost
c to produce that unit, then VOP will always have a strictly higher fitness than
DEF, selecting the proportion of VOP cells towards 1. In addition, if the maximum
possible benefit of the club good to aerobic cells is higher than the benefit of an
extra unit of acidification, then for sufficiently high number of VOP, GLY will have
lower fitness than aerobic cells. When both conditions are satisfied, the population
will converge towards all VOP.

```
# Definition of the function for creating the corresponding dataframe.
avc <- function (a, v, c) {
  data.frame(Genotype = c("WT", "GLY", "VOP", "DEF"),
          Fitness = c("1",
                    paste0("1 + ",a," * (f_GLY + 1)"),
                    paste0("1 + ",a," * f_GLY + ",v, " * (f_VOP + 1) - ",c),
                    paste0("1 + ",a," * f_GLY + ",v, " * f_VOP")
                    ))
          }

# Specification of the different effects on fitness.
afavc <- allFitnessEffects(genotFitness = avc(2.5, 7, 1),
                      frequencyDependentFitness = TRUE,
```

```
                                    frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = avc(2.5, 7, 1),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
simulation <- oncoSimulIndiv(afavc,
                             model = "McFL",
                             onlyCancer = FALSE,
                             finalTime = 15,
                             mu = 1e-4,
                             initSize = 4000,
                             keepPhylog = FALSE,
                             seed = NULL,
                             errorHitMaxTries = FALSE,
                             errorHitWallTime = FALSE)
```

```
## We get a huge number of VOP very quickly
## (too quickly?)
plot(simulation, show = "genotypes", type = "line",
     ylab = "Number of individuals", main = "Fully angiogenic tumours",
     font.main=2, font.lab=2, cex.main=1.4, cex.lab=1.1, las = 1)
```


Fully angiogenic tumours
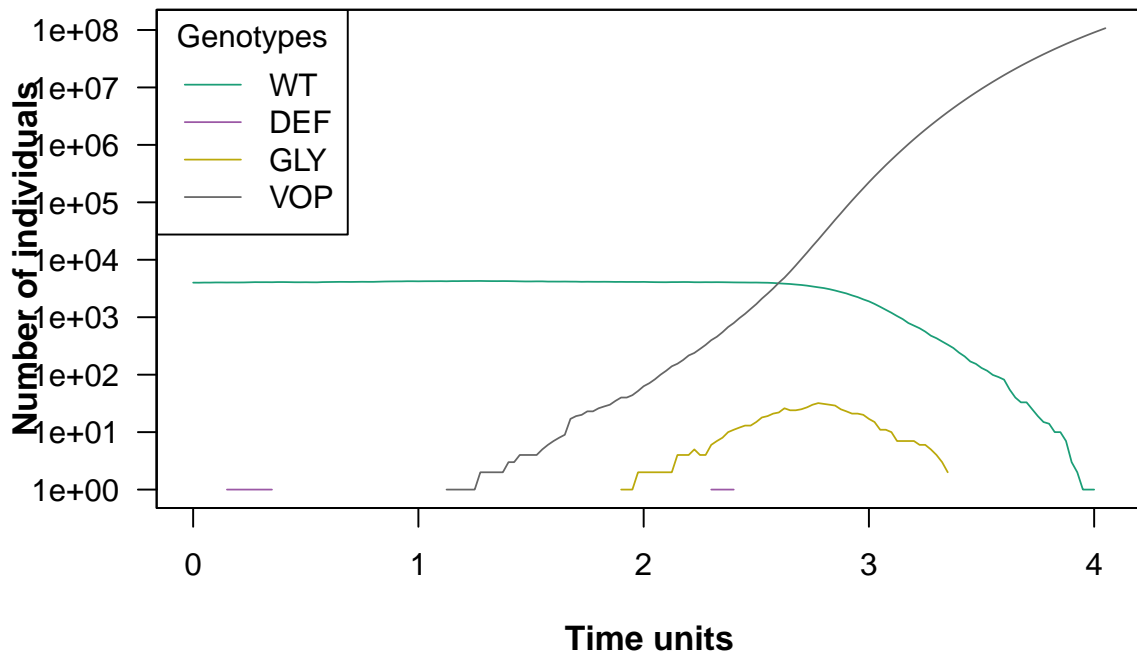
11.3.3 Heterogeneous tumours:

If the benefit from an extra unit of vascularisation in a fully aerobic group is lower
than the cost c to produce that unit, then for a sufficiently low proportion of GLY

and thus sufficiently large number of aerobic cells sharing the club good, DEF will have higher fitness than VOP. This will lead to a decrease in the proportion of VOP among aerobic cells and thus a decrease in the average fitness of aerobic cells. A lower fitness in aerobic cells will lead to an increase in the proportion of GLY until the aerobic groups (among which the club good is split) get sufficiently small and fitness starts to favour VOP over DEF, swinging the dynamics back.

```r
# Definition of the function for creating the corresponding dataframe.
avc <- function (a, v, c) {
  data.frame(Genotype = c("WT", "GLY", "VOP", "DEF"),
             Fitness = c("1",
                         paste0("1 + ",a," * (f_GLY + 1)"),
                         paste0("1 + ",a," * f_GLY + ",v," * (f_VOP + 1) - ",c),
                         paste0("1 + ",a," * f_GLY + ",v," * f_VOP")
                         ))
             }

# Specification of the different effects on fitness.
afavc <- allFitnessEffects(genotFitness = avc(7.5, 2, 1),
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = avc(7.5, 2, 1),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```r
# Launching of the simulation (20 times).
simulation <- oncoSimulIndiv(afavc,
                          model = "McFL",
                          onlyCancer = FALSE,
                          finalTime = 25,
                          mu = 1e-4,
                          initSize = 4000,
                          keepPhylog = FALSE,
                          seed = NULL,
                          errorHitMaxTries = FALSE,
                          errorHitWallTime = FALSE)
```

```r
# Representation of the plot of one simulation as an example (the others are
# highly similar).
plot(simulation, show = "genotypes", type = "line",
     ylab = "Number of individuals", main = "Heterogeneous tumours",
     font.main=2, font.lab=2, cex.main=1.4, cex.lab=1.1, las = 1)
```

Heterogeneous tumours
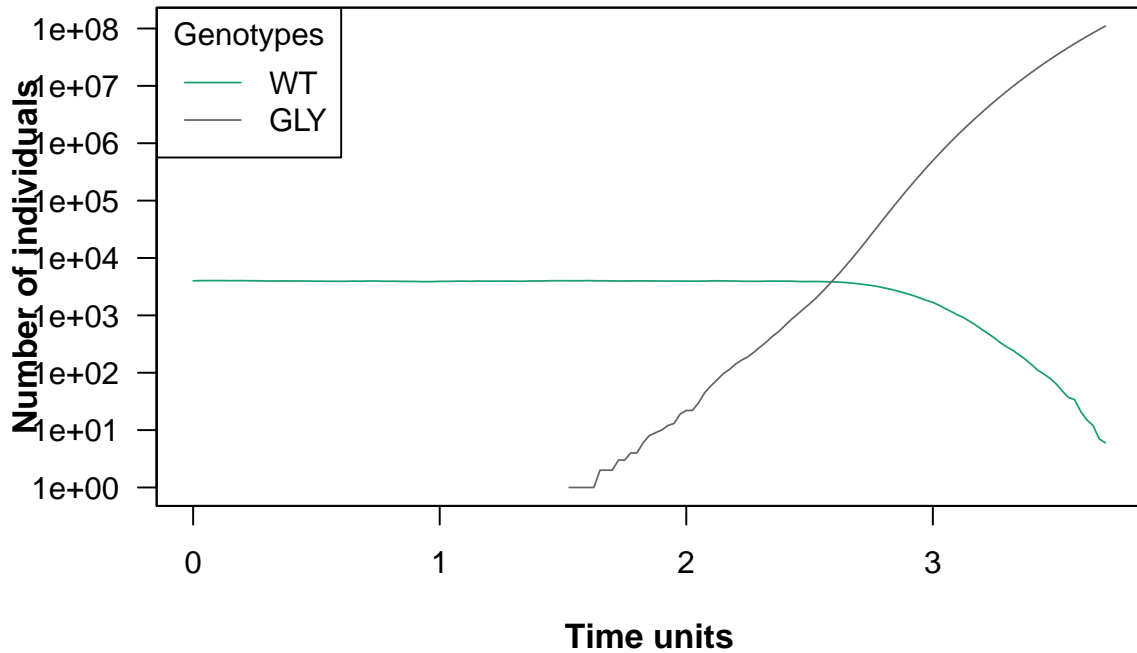
11.4   Prostate cancer tumour–stroma interactions

This example is based on Basanta et al. (2012). The authors apply evolutionary game theory to model the behavior and progression of a prostate tumour formed by three different cell populations: stromal cells, a dependant tumour phenotype capable of co-opting stromal cells to support its growth and an independent tumour phenotype that does not require microenvironmental support, be it stromal associated or not. To enable this, the model has four variables, which is the minimun necessary to describe the relationships in terms of costs and benefits between the different types of cells and, of course, to describe the progression of the cancer.

The different cell types, hence, are as follows:

1. S: stromal cells.
2. D: microenvironmental-dependent tumour cells.
3. I: microenvironmental-independent tumour cells.

And the parameters that describe the relationships are as follows:

- α: benefit derived from the cooperation between a S cell and a D cell.
- β: cost of extracting resources from the microenvironment.
- γ: cost of being microenvironmentally independent.
- ρ: benefit derived by D from paracrine growth factors produced by I cells.

Table 18 shows the payoffs for each cell type when interacting with others. We consider no other phenotypes are relevant in the context of the game and disregard spatial considerations.

Table 18: Payoff table that represents the interactions between the three cell types considered by de model

.	S	D	I
S	0	α	0
D	$1 + \alpha - \beta$	$1 - 2\beta$	$1 - \beta + \rho$
I	$1 - \gamma$	$1 - \gamma$	$1 - \gamma$

As in Basanta et al. (2012), the I cells are relatively independent from the microenvironment and produce their own growth factors (e.g. testosterone) and thus are considered to have a comparatively constant fitness $(1 - \gamma)$, where γ represents the fitness cost for I cells to be independent. The D cells rely more on their microenvironment for survival and growth at a fitness cost (β) that represents the scarcity of resources or space that I cells can procure themselves. A resource-poor microenvironment would then be characterised by a higher value of β. As I cells produce space and shareable growth factors, this model assumes that D cells derive a fitness advantage from their interactions with I cells represented by the variable ρ. On the other hand, D cells interacting with other D cells will have a harder time sharing existing microenvironmental resources with other equally dependant cells and thus are assumed to have double the cost 2β for relying on the microenvironment for survival and growth and thus have a fitness of $1\check{}2\beta$. The S cells can interact with tumour cells. In a normal situation, this population are relatively growth quiescent with low rates of proliferation and death. For this reason the fitness benefit derived by stromal cells from the interactions with tumour cells is assumed to be zero. However, they are able to undergo rapid proliferation and produce growth factors if they are stimulated by factors produced by I cells, giving rise to a mutualistic relationship. This relationship is represented by the parameter α. A low α represents tumours in which the stroma cannot be co-opted and vice versa.

From these variables, the fitness of each cell population $(W(S), W(I), W(D))$ is as follows:

$$W(S) = 1 + f_3\alpha \tag{10}$$

$$W(I) = 2 - \gamma \tag{11}$$

$$W(D) = 1 + (1 - f_2 - f_3)(1 - \beta + \alpha) + f_2(1 - \beta + \rho) + f_3(1 - 2\beta) + 1 - \beta + \alpha + f_2(\rho - \alpha) - f_3(\beta + \alpha) \tag{12}$$

where f_2 is the frequency of I cells and f_3 is the frequency of D cells at a given time. All fitness equations start from balance by the sum of 1.

11.4.1 Simulations

First, we define the fitness of the different genotypes (see Equations (10), (11) and (12)) through the function *fitness_rel* that builds a data frame.

It is important to note that this program models a situation where, from a WT cell population, the rest of the cell population types are formed. However, this model has also stromal cells that are not formed from a WT, since they are not tumour cells although interacting with it. Hence, for this model, we can not represent scenarios with total biological accuracy, something that we must consider when interpreting the results.

```r
fitness_rel <- function(a, b, r, g, gt = c("WT", "S", "I", "D")) {
    data.frame(
      Genotype = gt,
      Fitness = c("1",
                  paste0("1 + ", a, " * f_D"),
                  paste0("1 + 1 - ", g),
                  paste0("1 + (1 - f_I - f_D) * (1 - ", b, " + ",
                         a, ") + f_I * (1 - ", b, " + ", r,
                         ") + f_D * (1 - 2 * ", b, ") + 1 - ", b,
                         " + ", a, " + f_I * (", r, " - ",
                         a, ") - f_D * (", b, " + ", a, ")"))
                  )
}
```

Then, we are going to model different scenarios that represent different biological situations. In this case, we are going to explain four possible situations.

Note: for these simulations the values of paratemers are normalised in the range (0 : 1) so 1 represents the maximum value for any parameter being positive of negative to fitness depending on the parameter.

11.4.1.1   First scenario   In this simulation, we are modelling a situation where the environment is relatively resource-poor. In addition, we set a intermediate cooperation between D-D and D-I and a very low benefit from coexistence of D with I.

- α (a) = 0.5: intermediate cooperation between D and D cells.
- β (b) = 0.7: relatively resource-poor microenvironment.
- ρ (p) = 0.1: low benefit of D cells.
- γ (g) = 0.8: high cost of independence of I cells.

We can observe that high values of α and low values of ρ are translated in a larger profit of D cells from his interaction with S cells than from his interaction with I cells. Also, because of the high cost of independence of I cells (γ), it is not surprise that this population ends up becoming extinct. Finally, the tumour is composed by two cellular types: D and S cells.

```r
scen1 <- allFitnessEffects(genotFitness = fitness_rel(a = 0.5, b = 0.7,
                                                      r = 0.1, g = 0.8),
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = fitness_rel(a = 0.5, b
```

```r
set.seed(1)
simulScen1 <- oncoSimulIndiv(scen1,
                             model = "McFL",
                             onlyCancer = FALSE,
                             finalTime = 70,
                             mu = 1e-4,
                             initSize = 5000,
                             keepPhylog = FALSE,
                             seed = NULL,
                             errorHitMaxTries = FALSE,
                             errorHitWallTime = FALSE)
op <- par(mfrow = c(1, 2))
plot(simulScen1, show = "genotypes", type = "line",
     main = "First scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
plot(simulScen1, show = "genotypes",
     main = "First scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
```
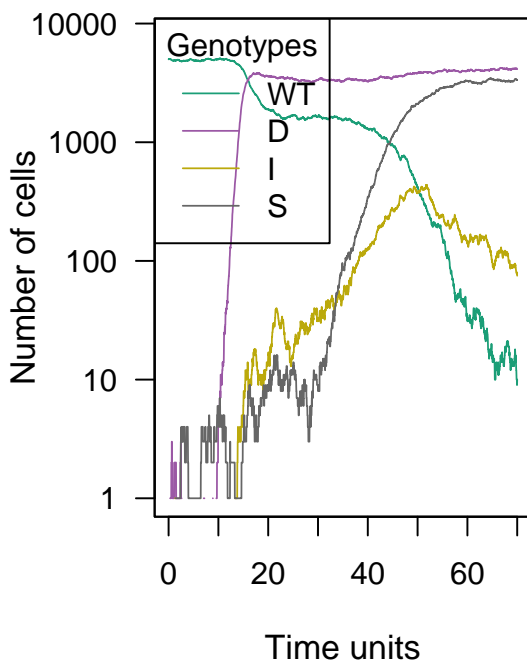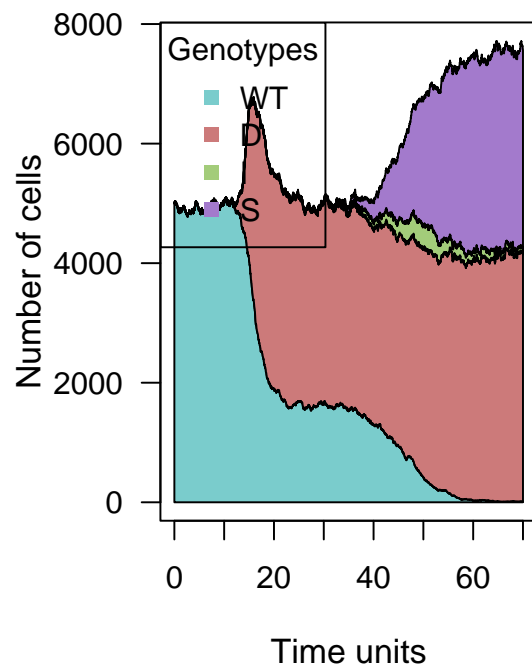


```r
par(op)
```

To understand the stability of the results, we should run multiple simulations. We will not pursue that here. Note that the results can be sensitive to the initial population

size and the mutation rate.

11.4.1.2 Second scenario In this case, we set α lower than in the first scenario and we enable the indepenence of I cells through a lower γ.

- α (a) = 0.3: low cooperation between D cells.
- β (b) = 0.7: relatively resource-poor microenvironment.
- ρ (r) = 0.1: low benefit of D cells from coexisting with I cells.
- γ (g) = 0.7: lower cost of independence of I cells than in the first scenario.

Because of we are easing the possibility of independence of I cells, instead of extinguishing as in the first scenario, they compose the bulk of the tumour along with D cells in spite of the low benefit of cooperation between them (low ρ). Besides, we can observe that the population of I cells is bigger than the population of D cells, being at the end of the simulation in balance. On the other hand, stromal cells drop at the beginning of the simulation.

```
scen2 <- allFitnessEffects(genotFitness = fitness_rel(a = 0.3, b = 0.7,
                                                      r = 0.1, g = 0.7),
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = fitness_rel(a = 0.3, b
## = 0.7, : v2 functionality detected. Adapting to v3 functionality.
```

```
set.seed(1)

simulScen2 <- oncoSimulIndiv(scen2,
                             model = "McFL",
                             onlyCancer = FALSE,
                             finalTime = 70,
                             mu = 1e-4,
                             initSize = 4000,
                             keepPhylog = FALSE,
                             seed = NULL,
                             errorHitMaxTries = FALSE,
                             errorHitWallTime = FALSE)
op <- par(mfrow = c(1, 2))
plot(simulScen2, show = "genotypes", type = "line",
     main = "Second scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
plot(simulScen2, show = "genotypes",
     main = "Second scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
```
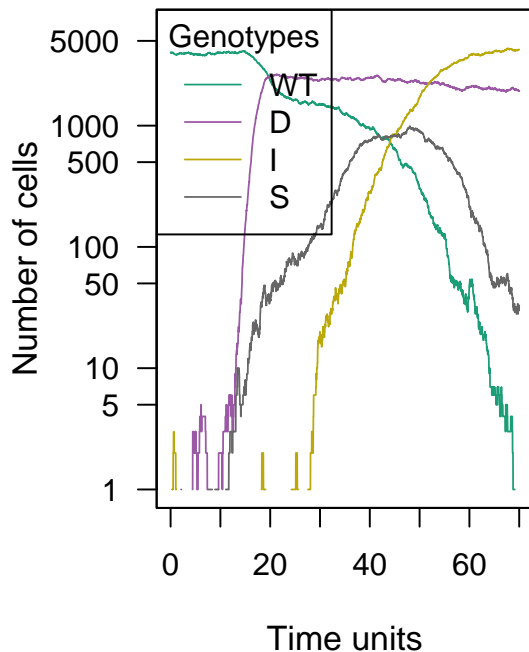
Second scenario
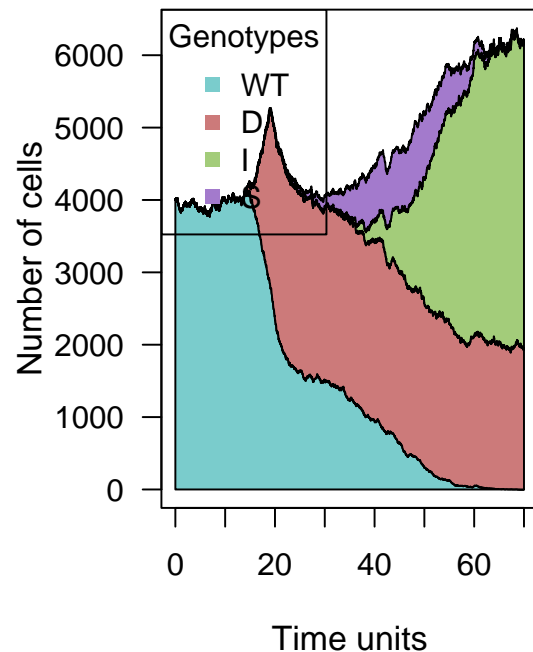

```
par(op)
```

11.4.1.3   Third scenario   In this case, we have a extreme situation where the microenvironment is rich (high β) and the independence costs are very low (γ) in relation with the previous scenarios.

- α (a) = 0.2: low cooperation between D cells.
- β (b) = 0.3: rich microenvironment, being beneficial for D cells.
- ρ (r) = 0.1: low benefit of D cells from coexisting with I cells.
- γ (g) = 0.3: independence costs very low, being beneficial for I cells.

Although γ and ρ are very low, which could make us think that I cells will control the tumour, we can observe that the fact that the microenvironment is very rich (with a low value of β) allows to D cells lead the progression of the tumour over the rest of cell populations, including I cells.

```
scen3 <- allFitnessEffects(genotFitness = fitness_rel(a = 0.2, b = 0.3,
                                                        r = 0.1, g = 0.3),
                            frequencyDependentFitness = TRUE,
                            frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = fitness_rel(a = 0.2, b
## = 0.3, : v2 functionality detected. Adapting to v3 functionality.


set.seed(1)
simulScen3 <- oncoSimulIndiv(scen3,
                                model = "McFL",
                                onlyCancer = FALSE,
```

283

```
                                   finalTime = 50,
                                   mu = 1e-4,
                                   initSize = 4000,
                                keepPhylog = FALSE,
                                seed = NULL,
                                   errorHitMaxTries = FALSE,
                                   errorHitWallTime = FALSE)
op <- par(mfrow = c(1, 2))
plot(simulScen3, show = "genotypes", type = "line",
     main = "Third scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
plot(simulScen3, show = "genotypes",
     main = "Third scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
```
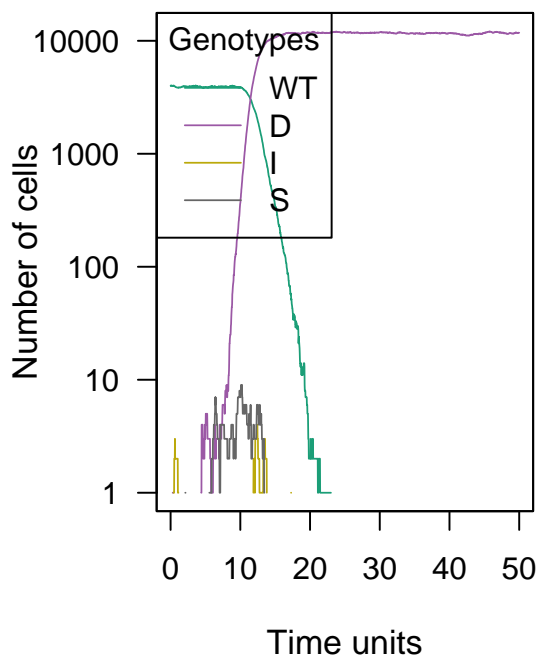


```
par(op)
```

11.4.1.4 Fourth scenario
This is a variation of the third scenario to illustrate that, if we set a microenvironment more rich than in the previous scenario, we get a cooperation between D and I cells, although we can still observe the superiority of D cells over I cells.

- α (a) = 0.2: low cooperation between D cells.
- β (b) = 0.4: lower richness than in the previous scenario.
- ρ (r) = 0.1: low benefit of D cells from coexisting with I cells.

- γ (g) = 0.3: independence costs very low, being beneficial for I cells.

```
scen4 <- allFitnessEffects(genotFitness = fitness_rel(a = 0.2, b = 0.4,
                                                      r = 0.1, g = 0.3),
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = fitness_rel(a = 0.2, b
## = 0.4, : v2 functionality detected. Adapting to v3 functionality.
```

```
## Set a different seed to show the results better since
## with set.seed(1) the progression of I cells was not shown
set.seed(2)

simulScen4 <- oncoSimulIndiv(scen4,
                             model = "McFL",
                             onlyCancer = FALSE,
                             finalTime = 40,
                             mu = 1e-4,
                             initSize = 4000,
                             keepPhylog = FALSE,
                             seed = NULL,
                             errorHitMaxTries = FALSE,
                             errorHitWallTime = FALSE)
op <- par(mfrow = c(1, 2))
plot(simulScen4, show = "genotypes", type = "line",
     main = "Fourth scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
plot(simulScen4, show = "genotypes",
     main = "Fourth scenario",
     cex.main = 1.4, cex.lab = 1.1,
     las = 1)
```
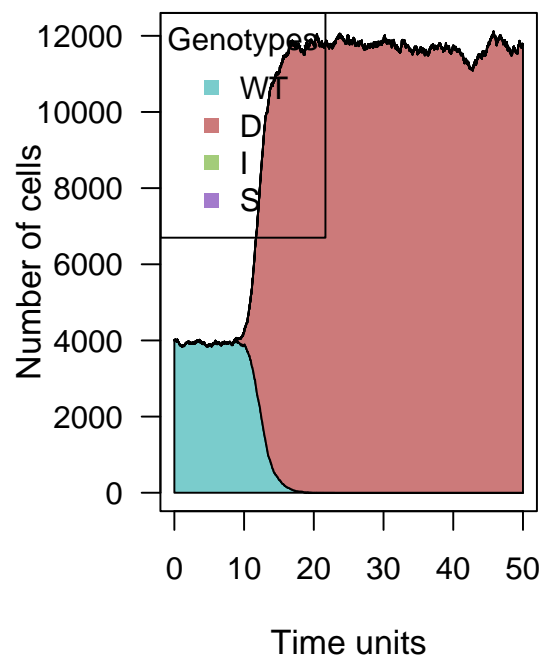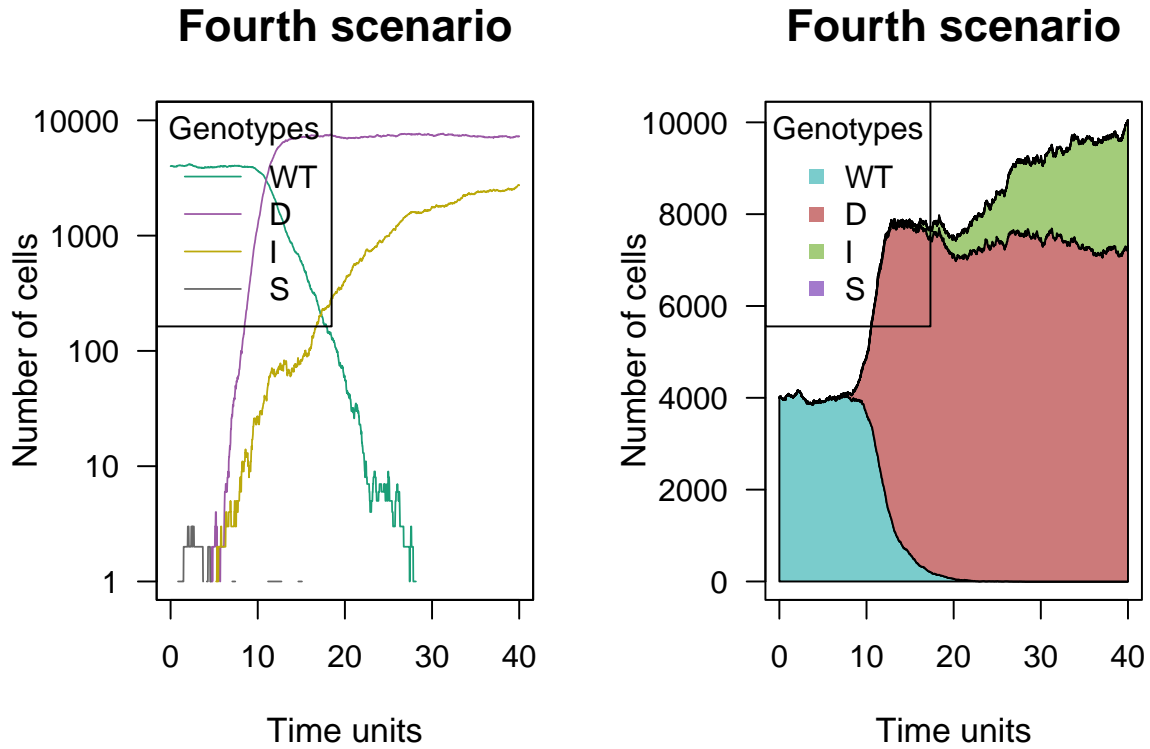
Fourth scenario   **Fourth scenario**

```r
par(op)
```

In this case, we can see that there is more variation in the size of population of I cells. There are cases where the I cell population cooperates with D cells and, in others, there is not cooperation. You can examine this running multiple simulations (or manually rerun the example above changing the seed).

11.5 Evolutionary Dynamics of Tumor-Stroma Interactions in Multiple Myeloma

This example is based on Sartakhti et al. (2016). The authors provide a frequency-dependent model to study the growth of malignant plasma cells in multiple myeloma. Assuming that cancer cells and stromal cells cooperate by exchanging diffusible factors, the study is carried out in the framework of evolutionary game theory.

We first need to define a payoff strategy for this kind of scenario. The following definitions are needed:

- There are n phenotypes in a population denoted by $\{P_1, \ldots, P_n\}$.
- Each phenotype can produce one diffusible factor $\{G_1, \ldots, G_n\}$.
- Each diffusible factor j has a different effect $r_{i,j}$ on the other phenotypes i.
- The cost for P_i for growth factor G_i is denoted as c_i.
- M is the number of cells within the diffusion range.
- There are M_j individuals of type P_j among the other group members.

Then, the payoff for strategy P_j is:

$$\pi_{P_j}(M_1, \ldots, M_n) = \frac{(M_j + 1) \times c_j}{M} r_{j,j} + \sum_{i=1, i \neq j}^{n} \frac{M_i \times c_i}{M} r_{j,i} - c_j . \tag{13}$$

In multiple Myeloma we have three different types of cells that have autocrine and paracrine effects on the cells within their diffusion range: Malignant plasma cells (MM), Osteoblasts (OB) and Osteoclasts (OC). The specification of fitness is the following (see [10]):

$$W_{OC} = \frac{(f_1(M-1)+1)c_1}{M} r11 + \frac{((1-f_3)(M-1) - f_1(M-1) - 1)c_2}{M} r12 \tag{14}$$

$$+ \frac{(M - (1-f_3)(M-1))c_3}{M} r13 - c_1 \tag{15}$$

$$W_{OB} = \frac{(f_2(M-1)+1)c_2}{M} r22 + \frac{((1-f_1)(M-1) - f_2(M-1) - 1)c_3}{M} r23 \tag{16}$$

$$+ \frac{(M - (1-f_1)(M-1))c_1}{M} r21 - c_2 \tag{17}$$

$$W_{MM} = \frac{(f_3(M-1)+1)c_3}{M} r33 + \frac{((1-f_2)(M-1) - f_3(M-1) - 1)c_1}{M} r31 \tag{18}$$

$$+ \frac{(M - (1-f_2)(M-1))c_2}{M} r32 - c_3 , \tag{19}$$

where f_1, f_2 and f_3 denote the frequency of the phenotype OC, OB and MM in the population. The multiplication factors for diffusible factors produced by the cells are shown in the following table (taken from Sartakhti et al. (2016)):

Several scenarios varying the values of the parameters are shown in Sartakhti et al. (2016). Here we reproduce some of them.

11.5.1 Simulations

First, we define the fitness of the different genotypes through the function *fitness_rel*.

```
f_cells <- function(c1, c2, c3, r11, r12, r13,
                    r21, r22, r23, r31, r32, r33, M, awt = 1e-4,
                        gt = c("WT", "OC", "OB", "MM")) {
  data.frame(Genotype = gt,
             Fitness = c(
                 paste0("max(0.1, 1 - ", awt, " * (f_OC + f_OB+f_MM)*N)"),
                 paste0("1", "+(((f_OC * (", M, "-1)+1)*", c1, ")/", M, ")*",r11,
                        "+(((((1-f_MM) * (", M, "-1)-f_OC*(", M, "-1)-1)*", c2, ")
                        "+(((", M, "-(1-f_MM)*(", M, "-1))*", c3, ")/", M, ")*",
                        "-", c1
                        ),
                 paste0("1", "+(((f_OB*(", M, "-1)+1)*", c2, ")/", M, ")*", r22,
```

Table 1. Multiplication factors for diffusible f

Phenotypes	G_{OC}
OC	neutral
OB	equilibr
MM	stimula

Figure 8: Multiplication factor in myeloma interaction. Table 1 in Sartakhti et al., 2016, 'Evolutionary Dynamics of Tumor-Stroma Interactions in Multiple Myeloma' https://doi.org/10.1371/journal.pone.0168856 .

```
                              "+((((1-f_OC)*(", M, "-1)-f_OB*(", M, "-1)-1)*", c3, ")/"
                              "+(((", M, "-(1-f_OC)*(", M, "-1))*", c1, ")/", M, ")*",
                              "-", c2
                              ),
                       paste0("1", "+(((f_MM*(", M, "-1)+1)*", c3, ")/", M, ")*", r33,
                              "+((((1-f_OB)*(", M, "-1)-f_MM*(", M, "-1)-1)*", c1, ")/"
                              "+(((", M, "-(1-f_OB)*(", M, "-1))*", c2, ")/", M, ")*",
                              "-", c3
                              )
                   )
               ,stringsAsFactors = FALSE
               )
}
```

It is important to note that, in order to exactly reproduce the experiments of the paper, we need to create an initial population with three different types of cell, but we do not need the presence of a wild type. For this reason, we will increase the probability of mutation of the wild type, which will disappear in early stages of the simulation; this is a procedure we have used before in several cases too (e.g., 11.2). This is something that we must consider when interpreting the results.

11.5.2 Scenario 1

Here we model a common situation in multiple myeloma in which $c_1 < c_2 < c_3$. In the presence of a small number of MM cells, the stable point on the OB-OC border becomes a saddle point and clonal selection leads to a stable coexistence of OC and MM cells. Parameters for the simulation can be seen in the R code.

```
N <- 40000
M <- 10
c1 <- 1
c2 <- 1.2
c3 <- 1.4
r11 <- 0
r12 <- 1
r13 <- 2.5
r21 <- 1
r22 <- 0
r23 <- -0.3
r31 <- 2.5
r32 <- 0
r33 <- 0

fe_cells <-
    allFitnessEffects(
        genotFitness =
            f_cells(c1, c2, c3, r11, r12, r13,
```

```
                          r21, r22, r23, r31, r32, r33, M,
                          gt = c("WT", "OC", "OB", "MM")),
        frequencyDependentFitness = TRUE,
        frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = f_cells(c1, c2, c3,
## r11, r12, : v2 functionality detected. Adapting to v3
## functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
## Simulated trajectories

## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below
data(smyelo3v57)

if (FALSE) {
set.seed(2)
smyelo3v57 <- oncoSimulIndiv(fe_cells,
                          model = "McFL",
                          onlyCancer = FALSE,
                          finalTime = 20,
                          mu = c("OC"=1e-1, "OB"=1e-1, "MM"=1e-4),
                          initSize = N,
                          keepPhylog = FALSE,
                          seed = NULL,
                          errorHitMaxTries = FALSE,
                          errorHitWallTime = FALSE,
                          keepEvery = 0.1)
}
## Plot trajectories
plot(smyelo3v57, show = "genotypes")
```
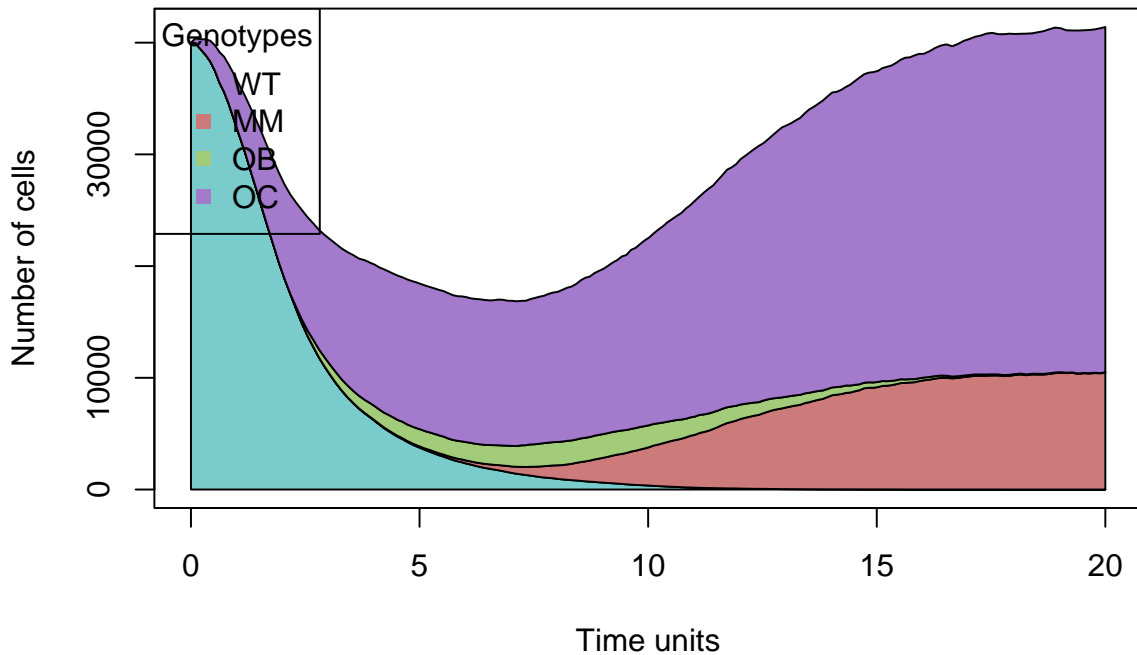
Clearly, the appearance of MM cells quickly brings the system to an equilibrium point, which is stable. OB cells are extinct and cancer has propelled.

11.5.3 Scenario 2

As in the second scenario of [10] ($c1 = c2 = c3$) configuration A (upper row in the grid of images). We should find one stable point on the OC-OB edge under certain conditions, which are met in the example. Further information about the parameters can be found in the R code shown below.

```R
N <- 40000
M <- 10
c1 <- 1
c2 <- 1
c3 <- 1
r11 <- 0
r12 <- 1
r13 <- 0.5
r21 <- 1
r22 <- 0
r23 <- -0.3
r31 <- 0.5
r32 <- 0
r33 <- 0

fe_cells <-
    allFitnessEffects(
        genotFitness =
            f_cells(c1, c2, c3, r11, r12, r13,
                    r21, r22, r23, r31, r32, r33, M,
```

```
                   gt = c("WT", "OC", "OB", "MM")),
        frequencyDependentFitness = TRUE,
        frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = f_cells(c1, c2, c3,
## r11, r12, : v2 functionality detected. Adapting to v3
## functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
## Simulated trajectories
set.seed(1)
simulation <- oncoSimulIndiv(fe_cells,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 15, ## 25
                        mu = c("OC"=1e-1, "OB"=1e-1, "MM"=1e-4),
                        initSize = N,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

```
#Plot trajectorie
plot(simulation, show = "genotypes", thinData = TRUE)
```
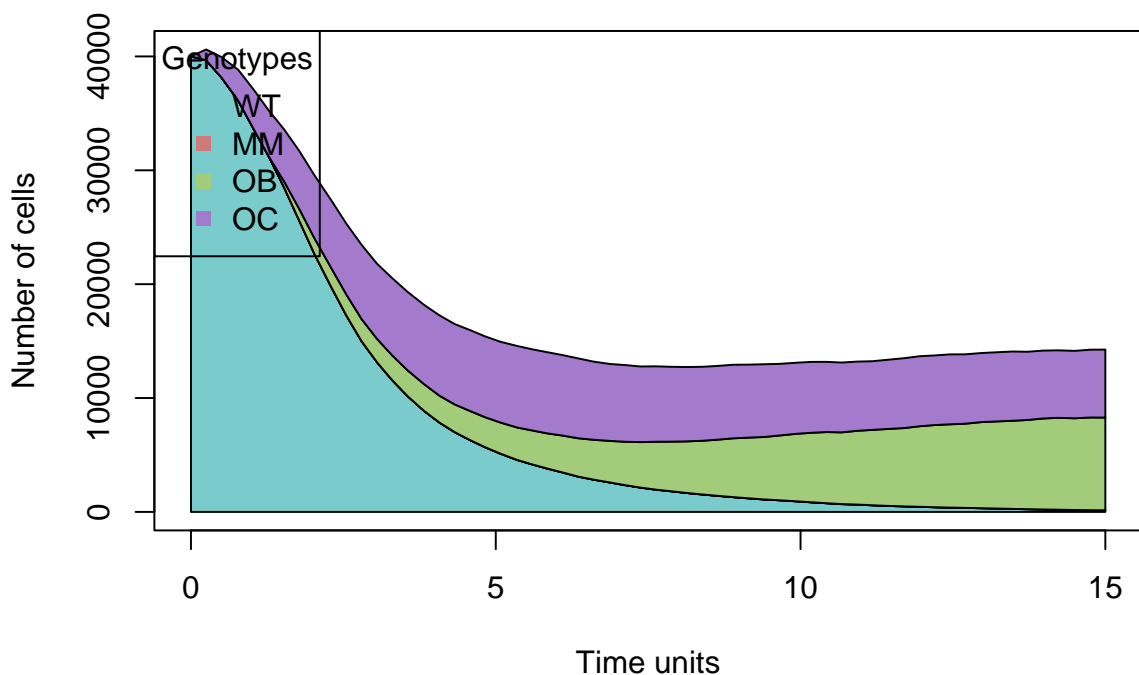
As expected, under these condtitions, MM cells are not able to propagate. The equilibrium point in the OB-OC edge is stable, resisting small variations in the number of MM cells.

11.6 An example of modellization in Parkinson disease related cell community

The following example is based on Masliah (2007), and it discusses the coexistance between cells that produce α-synuclein and β-synuclein, related with pore-like oligomer development and Parkinson disease.

```
park1<- data.frame(Genotype = c("WT", "A", "B", "A,B"),
                Fitness = c("1",
                "1 + 3*(f_1 + f_2 + f_1_2)",
                "1 + 2*(f_1 + f_2 + f_1_2)", ## We establish
                ## the fitness of B smaller than the one of A because
                ## it is an indirect cause of the disease and not a direct one.
                "1.5 + 4.5*(f_1 + f_2 + f_1_2)")) ## The baseline
                 ## of the fitness is higher in the
                 ## AB population (their growth is favored).


parkgen1<- allFitnessEffects(genotFitness = park1,
                        frequencyDependentFitness = TRUE,
                        frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = park1,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

- α- – normal a-synuclein production
- β- – normal b-synuclein production
- α+ – deleterious a-synuclein production, causing aggregation and the rest of mentioned effects
- β+ – deleterious b-synuclein production, preventing it from slowing down the aggregation of a-synuclein
- WT - α- and β- (balance)
- A - α+ and β- (increases [slight] the probability of Parkinson's disease)
- B – α- and β+ (increases [slight] the probability of Parkinson's disease)
- AB – α+ and β+ (massive increase of probability of Parkinson's disease)

In this simulation, at the very end, the only cells that remain alive are those from AB population (which means that both α and β are mutated, α+ and β+, which means that the individual has a high risk of developing the disease, and all the cells keep this mutation). Genotype AB is able to invade the population when there are some A's and B's around. Cooperation increases the fitness at 1.5 level respecting the fitness for just A's or just B's, so the rest of population (appart from AB) collapses. We can observe this cell behaviour in the following code and graphic:

```
set.seed(1)
fpark1 <- oncoSimulIndiv(parkgen1,
                         model = "McFL",
                         onlyCancer = FALSE,
                         finalTime = 100,
                         mu = 1e-4,
                         initSize = 5000,
                         keepPhylog = TRUE,
                         seed = NULL,
                         errorHitMaxTries = FALSE,
                         errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```
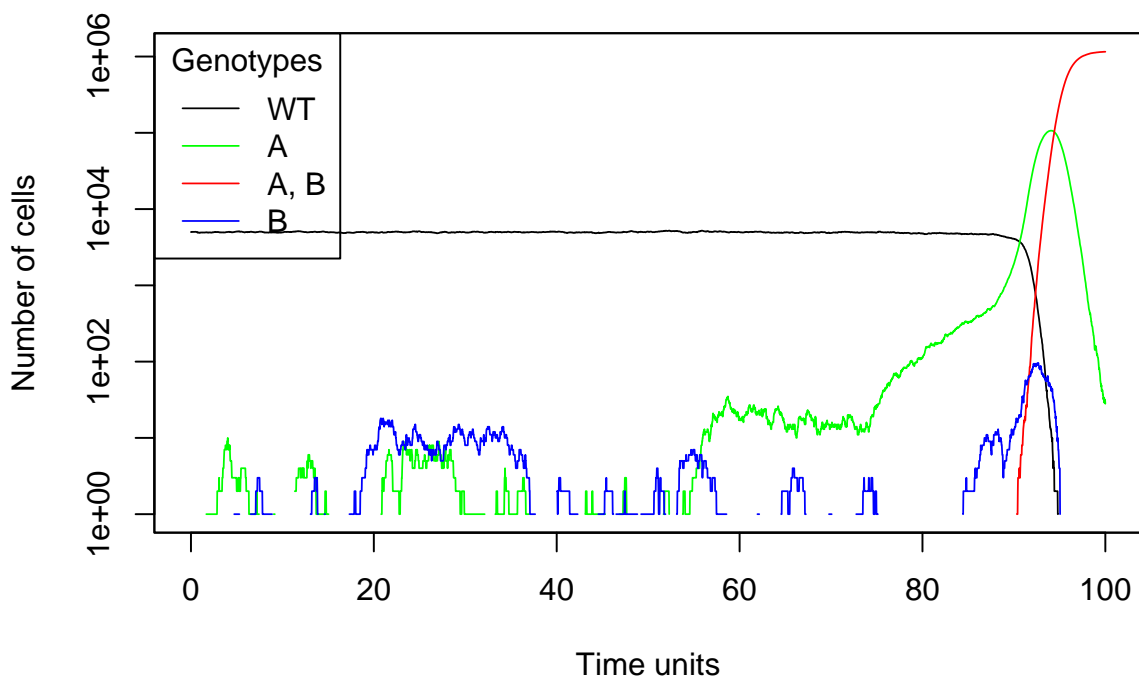
```
plot(fpark1, show = "genotypes",  type = "line",
     col = c("black", "green", "red", "blue"))
```


Analyzing the graphic obtain from OncoSimulR we can see that A, B and the WT population dissapear at (more or less) the same point that AB population increases drastically, which makes sense because the fitness of AB population is greater than the other three population fitnesses. As the birth rate of the population depends directly of the fitness, at the end, just the AB population survives. We have a model of frequency-dependent fitness here, but the results are not really surprising given the fitness of each type, and no coexistence is possible.

The AB population comes mainly from the A population, as would be expected, because of its fitness, and abundance, relative to that of the B population:

```
plotClonePhylog(fpark1, N = 0, keepEvents=TRUE, timeEvents=TRUE)
```
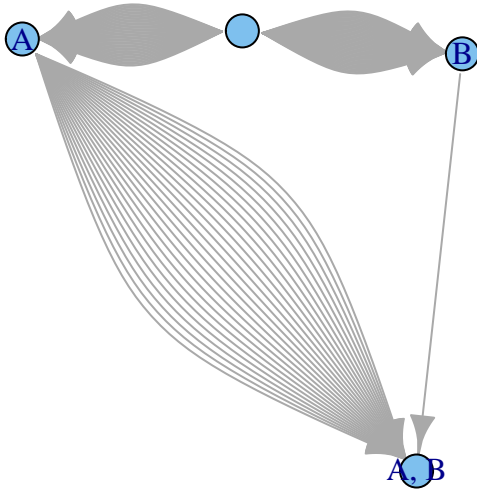

11.7 Evolutionary Game between Commensal and Pathogenic Microbes in Intestinal Microbiota

The following adapted example is based on Wu & Ross (2016). As explained in p. 2 of the aforementioned paper, the commensal microbiota has been simplified into two phenotypic groups: antibiotic-sensitive bacteria (CS) [WT in the code below] and antibiotic-tolerant bacteria (CT). In addition, a third phenotypic group of pathogenic bacteria (PA) is considered which are kept in low numbers in absence of intestinal microbiota disturbances. We assume CS and CT bacteria cooperate and depend on each other for optimal proliferation, leading to a benefit (bG) for both of them as well as to a cost for factor growth (cG) production which permits a stable coexistence between CS and CT cells if the fraction of PA cells is negligible. Meanwhile, PA possess a reproductive or metabolic advantage relative to CS and CT. Without antibiotic administration CS population inhibits PA population via the release of a chemical compound which harms PA (iPA) when their relative frequency is equal to or greater than 0.2, carrying a production cost (cI). However, PA population takes over CS population in the presence of antibiotic (ab). In this situation PA and CT compete for resources. Finally, a cohabit cost (cS) is considered for all three cell types.

The adapted payoff matrix obtained is shown in the next table.

Table 19: Payoff matrix for the adapted example
from Wu A, Ross D., 2016, 'Evolutionary Game
between Commensal and Pathogenic Microbes in
Intestinal Microbiota'. Games. 2016;7(3); doi:
10.3390/g7030026 .

	CS	CT	PA
CS	bG – cG – cS - ab	bG – cG – cS - ab	bG – cG – cS -cI - ab
CT	bG – cG – cS	bG – cG – cS	bG – cG – cS
PA	bPA – iPA - cS	bPA – cS	bPA – cS

On the basis of the interactions between the different microbe populations and the
payoff matrix, we establish fitness as:

- CS (Fcs = bG([CS] + [CT]) - cS*([CS] + [CT] + [PA]) - cI - cG - ab)
- CT (Fct = bG([CS] + [CT]) - cS*([CS] + [CT] + [PA]) - cG)
- PA (Fpa = bPA([PA]) - cS*([CS] + [CT] + [PA]) - iPA[CS])*

A function to create the data frame of frequency-dependent fitnesses is defined in
order to be able to model situations with different values for the parameters, and it
is assumed that WT derive to the other cell types by a single mutation.

```r
create_fe <- function(bG, cG, iPA, cI, cS, bPA, ab,
                      gt = c("WT", "CT", "PA")) {
  data.frame(Genotype = gt,
             Fitness = c(
               paste0("1 + ", bG, " * (f_ + f_CT) - ", cS,
                      " * (f_ +  f_CT + f_PA) - ", cI, "(f_PA > 0.2) - ", cG,
                      " - ", ab),
               paste0("1 + ", bG, " * (f_ + f_CT) - ", cS,
                      " * (f_ +  f_CT + f_PA) - ", cG),
               paste0("1 +", bPA, " - ", cS, " * (f_ +  f_CT + f_PA) - ", iPA,
                      " *(f_(f_PA > 0.2))")),
             stringsAsFactors = FALSE)
}
```

We can check we recover the Table 19 :

```r
create_fe("bG", "cG", "iPA", "cI", "cS", "bPA", "ab")
##   Genotype
## 1       WT
## 2       CT
## 3       PA
##                                                                       Fitness
## 1 1 + bG * (f_ + f_CT) - cS * (f_ +  f_CT + f_PA) - cI(f_PA > 0.2) - cG - ab
## 2                   1 + bG * (f_ + f_CT) - cS * (f_ +  f_CT + f_PA) - cG
## 3               1 +bPA - cS * (f_ +  f_CT + f_PA) - iPA *(f_(f_PA > 0.2))
```

We verify what we have specified executing `evalAllGenotypes`. We specify populations sizes for evaluating fitness at different moments in the total population evolution.

In the absence of antibiotic we observe how, even though CS (WT) and CT population size is equal, CT fitness is greater than CS fitness due to the PA inhibitory factor releasing cost. PA fitness is decreased by means of this inhibitor produced by CS.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                    create_fe(7, 1, 9, 0.5, 2, 5, 0),
                                    frequencyDependentFitness = TRUE,
                                    frequencyType = "rel"),
                  spPopSizes = c(1000, 1000, 1000))
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##    Genotype Fitness
## 1        WT   2.167
## 2        CT   2.667
## 3        PA   1.000
## 4    CT, PA   0.000
```

CS (WT) fitness decrease in the presence of antibiotic, while PA and CT fitness are not affected, and CT fitness is the largest.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                    create_fe(7, 1, 9, 0.5, 2, 5, 2),
                                    frequencyDependentFitness = TRUE,
                                    frequencyType = "rel"),
                  spPopSizes = c(1000, 1000, 1000))
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##    Genotype Fitness
## 1        WT  0.1667
## 2        CT  2.6667
```

```
## 3       PA  1.0000
## 4   CT, PA  0.0000
```

In the extreme situation in in which antibiotic is administrated and only PA is present, neither WT (CS) nor CT would be able to grow:

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                   create_fe(7, 1, 9, 0.5, 2, 5, 2),
                                   frequencyDependentFitness = TRUE,
                                   frequencyType = "rel"),
                 spPopSizes = c(0, 0, 1000))
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 1       WT       0
## 2       CT       0
## 3       PA       4
## 4   CT, PA       0
```

From a PA population size much greater than WT (CS) population size, the presence of WT decreases PA fitness via inhibitor production while WT fitness does not increase due to antibiotic administration.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                   create_fe(7, 1, 9, 0.5, 2, 5, 2),
                                   frequencyDependentFitness = TRUE,
                                   frequencyType = "rel"),
                 spPopSizes = c(100, 0, 1000))
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 1       WT   0.000
## 2       CT   0.000
```

```
## 3       PA    3.182
## 4   CT, PA   0.000
```

Starting from a population of WT and with high antibiotic doses, WT does not have the capacity of growing while CT shows the largest fitness (and PA fitness is decreased by the inhibitory compound produced by WT):

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                  create_fe(7, 1, 9, 0.5, 2, 5, 5),
                                  frequencyDependentFitness = TRUE,
                                  frequencyType = "rel"),
                 spPopSizes = c(1000, 0, 0))
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 1       WT       0
## 2       CT       5
## 3       PA       4
## 4   CT, PA       0
```

At the same WT and PA population size and with high antibiotic dose administration, the fitness of both of them is decreased to the same extent since they inhibit each other while CT grow a little.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                  create_fe(7, 1, 9, 0.5, 2, 5, 5),
                  frequencyDependentFitness = TRUE,
                                  frequencyType = "rel"),
                 spPopSizes = c(1000, 0, 1000))
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 1       WT     0.0
```

```
## 2       CT      1.5
## 3       PA      0.0
## 4   CT, PA      0.0
```

We have verified that fitness specification gives the expected results:

Now we create **allFitnessEffects** object and simulate two different situations: microbiota evolution in the absence and presence of antibiotic. The model used is McFL so density dependence in the death rate is considered.

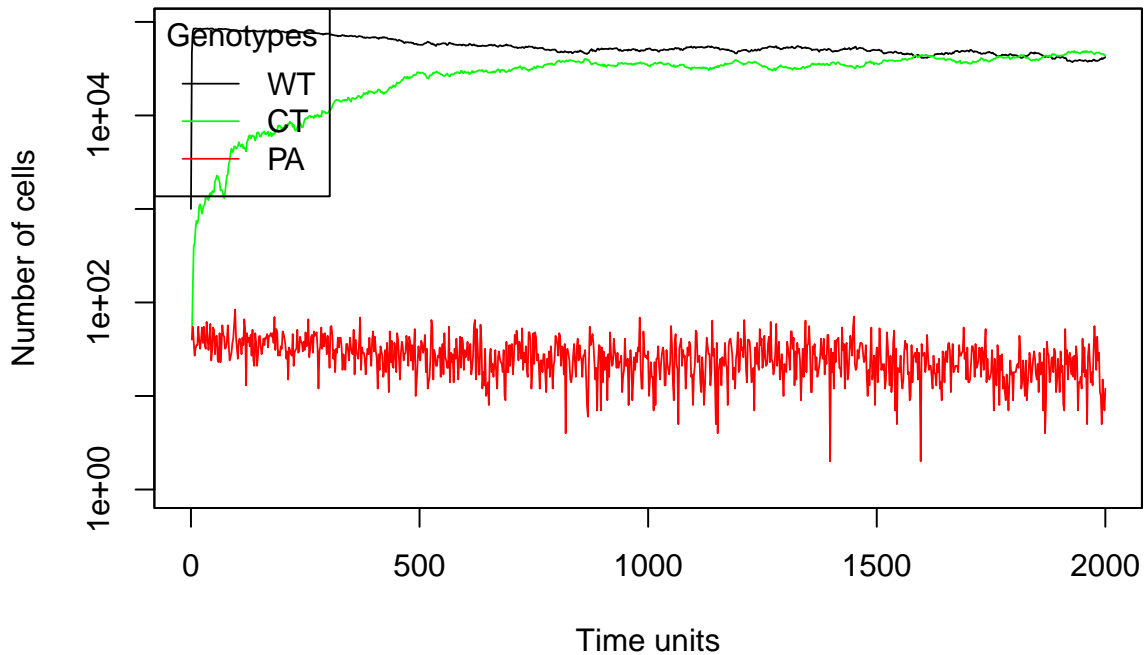### 11.7.1 Antibiotic absence situation

We observe how CT and PA cells emerge at time 0 from WT mutations. PA population starts to grow but when its frequency is greater than the established threshold, 0.2, WT population produces inhibitory compounds which harm PA and affect its fitness; when its frequency is under that threshold, WT stop releasing inhibitory PA growth factor and PA starts to grow again. This loop remains over time. Meanwhile, WT population decreases slowly due to the cost of producing the inhibitor when PA frequency exceed the established 0.2 and is reached by CT population, which grow until WT and CT are stabilized and cohabit taking in account the cost for sharing space.

```
woAntib <- allFitnessEffects(
  genotFitness = create_fe(7, 1, 9, 0.5, 2, 5, 0),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")


## We do not run this for speed but load it below
set.seed(2)
woAntibS <- oncoSimulIndiv(woAntib,
                    model = "McFL",
                    onlyCancer = FALSE,
                    finalTime = 2000,
                    mu = 1e-4,
                    initSize = 1000,
                    keepPhylog = FALSE,
                    seed = NULL,
                    errorHitMaxTries = FALSE,
                    errorHitWallTime = FALSE,
                    keepEvery = 2 ## store a smaller object
                    )

## Load stored results
data(woAntibS)


plot(woAntibS, show = "genotypes", type = "line",
     col = c("black", "green", "red"))
```

11.7.2 Antibiotic presence situation

We observe how CT and PA cells emerge from WT mutations. WT cells decrease in number due to antibiotic administration and inhibitory PA growth factor when the frequency of this latest surpass the threshold imposed, so PA population grow with difficulty in comparison to CT. WT population finally disappear and CT and PA compete for resources, but CT takes over PA given its larger population size and CT population remain stable over time.

```
wiAntib <- allFitnessEffects(
  genotFitness = create_fe(7, 1, 9, 0.5, 2, 5, 2),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(7, 1, 9, 0.5,
## 2, 5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
set.seed(2)
wiAntibS <- oncoSimulIndiv(wiAntib,
                      model = "McFL",
                      onlyCancer = FALSE,
                      finalTime = 100,
                      mu = 1e-4,
                      initSize = 1000,
                      keepPhylog = FALSE,
                      seed = NULL,
                      errorHitMaxTries = FALSE,
                      errorHitWallTime = FALSE)
```

301

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

```
plot(wiAntibS, show = "genotypes", type = "line",
     col = c("black", "green", "red"))
```
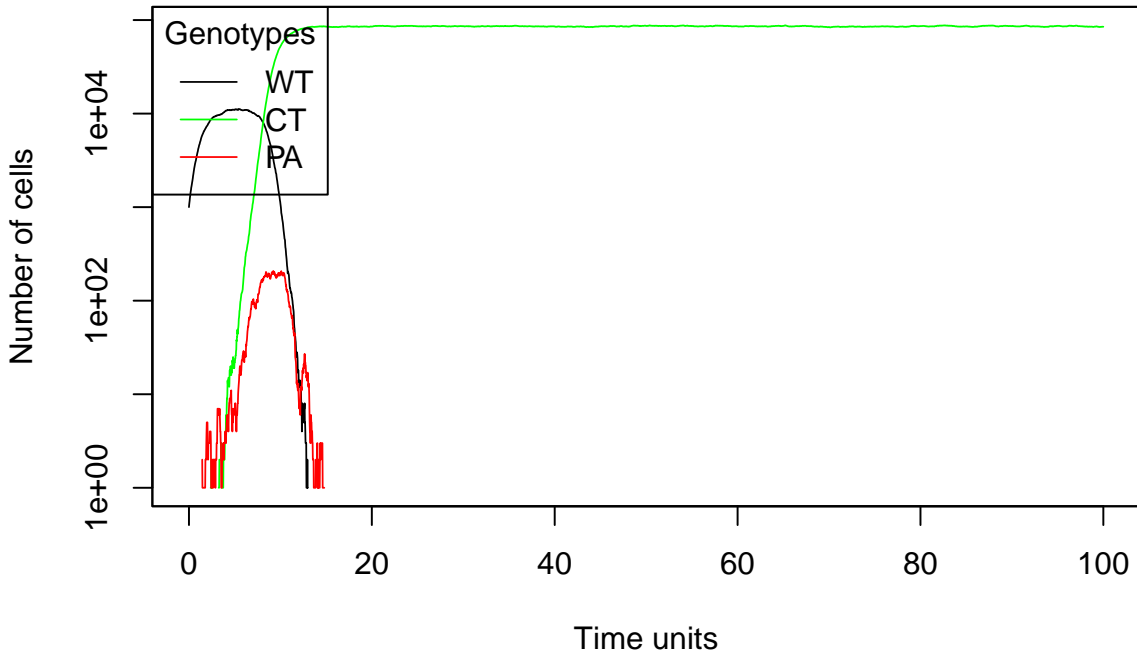

11.8   Modeling of breast cancer through evolutionary game theory.

The following example is based on Barton & Sendova (2018). This model assumes that there are four different types of cells in the body: (a) the native cells (NC), which are the healthy stromal cells; (b) the macrophages (Mph), which are part of the immune system; (c) the benign tumor cells (BTC), lump-forming cancer cells that lack the ability to metastasize; (d) the motile tumor cells (MTC), metastatic cancer cells that can invade neighboring tissues.

Both the native cells and macrophages produce growth factor, which benefits all types of cells. The cost of producing the growth factor, cG, and the benefits of the growth factor, bG, will be assumed to be the same for all types of the cells. The macrophages and motile tumor cells can move and we will assume that the ability comes at the costs cM,Mph, and cM,MTC respectively. The native cells and benign tumor cells stay in place and thus have to share the resources with other native and benign tumor cells, which comes at the cost cS. The cancer cells can reproduce faster than native cells or macrophages, which we model by additional benefit bR to the cancer cells, but the cancer cells can be destroyed by macrophages, which we model by additional cost cD to the cancer cells.

They provide also the payoff matrix reproduced in next table :

Table 20: Payoff matrix from Barton et al., 2018, 'Modeling of breast cancer through evolutionary game theory', Involve, a Journal of Mathematics, 11(4), 541-548; https://doi.org/10.2140/involve.2018.11.541 .

	MTC	Mph	NC	BTC
MTC	bR - cMMTC	bR - cMMTC - cD + bG	bR - cMMTC + bG	bR - cMMTC
Mph	- cG - cMMph	bG - cG - cMMph	bG - cG - cMMph	-cG - cMMph
NC	- cG	bG - cG	bG - cG - cS	- cG - cS
BTC	bR	bR + bG - cD	bR + bG - cS	bR - cS

Overall, when the concentrations of the cells are [NC], [Mph], [BTC] and [MTC], the net benefits (benefits minus the costs) to each type of the cells are:

- W(NC) = bG([NC] + [Mph]) - cG - cS([NC] + [BTC])
- W(Mph) = bG([NC] + [Mph]) - cG - cMMph
- W(BTC) = bR + bG([NC] + [Mph]) - cS([NC] + [BTC]) - cD[Mph]
- W(MTC) = bR + bG([NC] + [Mph]) - cMMTC - cD[Mph]

To allow modelling scenarios with different values for the parameters above we will define a function to create the data frame of frequency-dependent fitnesses. First, we will consider the NC cell type as the WT. Moreover, we will assume that each one of the other cell types types (Mph , BTC and MTC) are all derived from WT by a single mutation in one of three genes, say, O, B, M, respectively.

```r
create_fe <- function(cG, bG, cS, cMMph, cMMTC, bR, cD,
                  gt = c("WT", "Mph", "BTC", "MTC")) {
  data.frame(Genotype = gt,
          Fitness = c(
            paste0("1 + ", bG, "*(f_ + f_Mph) - ", cG, " - ", cS, "*(f_ + f_BTC)
            paste0("1 + ", bG, "*(f_ + f_Mph) - ", cG, " - ", cMMph),
            paste0("1 + ", bR, " + ", bG, "*(f_ + f_Mph) - ", cS, "* (f_ + f_BTC
                cD , " * f_Mph"),
            paste0("1 + ", bR, " + ", bG, " *(f_ + f_Mph) -", cMMTC, " - ",
                cD , " * f_Mph")
            ),
          stringsAsFactors = FALSE)
}
```

We can check we recover the Table 20 :

```r
create_fe("cG", "bG","cS", "cMMph", "cMMTC", "bR", "cD")
##   Genotype                                                          Fitness
```

```
## 1       WT                1 + bG*(f_ + f_Mph) - cG - cS*(f_ + f_BTC)
## 2       Mph                      1 + bG*(f_ + f_Mph) - cG - cMMph
## 3      BTC 1 + bR + bG*(f_ + f_Mph) - cS* (f_ + f_BTC) -cD * f_Mph
## 4      MTC          1 + bR + bG *(f_ + f_Mph) -cMMTC - cD * f_Mph
```

We check that we have correctly specified the different parameters executing `evalAllGenotypes`. For this, we specify populations sizes for evaluating fitness at different moments in the total population evolution.

When there are only wild-type cells, the fitness of the cancer cells is higher than the other type of cells' fitness. This makes sense, since there are no macrophages that can affect them.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                               create_fe(2, 5, 1, 0.8, 1, 1, 9),
                               frequencyDependentFitness = TRUE,
                               frequencyType = "rel"),
                  spPopSizes = c(WT = 1000, Mph = 0, BTC = 0, MTC = 0))
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##       Genotype Fitness
## 1           WT     3.0
## 2          BTC     6.0
## 3          Mph     3.2
## 4          MTC     6.0
## 5     BTC, Mph     0.0
## 6     BTC, MTC     0.0
## 7     Mph, MTC     0.0
## 8 BTC, Mph, MTC     0.0
```

In case that there are wild-type cells and macrophages, fitness of cancer cells is lower than before.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                               create_fe(2, 5, 1, 0.8, 1, 1, 9),
                               frequencyDependentFitness = TRUE,
                               frequencyType = "rel"),
                  spPopSizes = c(WT = 1000, Mph = 1000, BTC = 0, MTC = 0))
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##       Genotype Fitness
## 1            WT    3.5
## 2           BTC    2.0
## 3           Mph    3.2
## 4           MTC    1.5
## 5      BTC, Mph    0.0
## 6      BTC, MTC    0.0
## 7      Mph, MTC    0.0
## 8 BTC, Mph, MTC    0.0
```

When cancer cells start to grow, the fitness of wild type cells and macrophages decrease. This makes sense for wild-type cells, since they will share space with BTC cells, reducing their resources; and for macrophages, because the decrease of wild-type cells will affect the benefit of growth factor produced by them.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                   create_fe(2, 5, 1, 0.8, 1, 1, 9),
                                   frequencyDependentFitness = TRUE,
                                   frequencyType = "rel"),
                 spPopSizes = c(WT = 1000, Mph = 1000, BTC = 100, MTC = 100))
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##       Genotype Fitness
## 1            WT  3.045
## 2           BTC  1.955
## 3           Mph  2.745
## 4           MTC  1.455
## 5      BTC, Mph  0.000
## 6      BTC, MTC  0.000
## 7      Mph, MTC  0.000
## 8 BTC, Mph, MTC  0.000
```
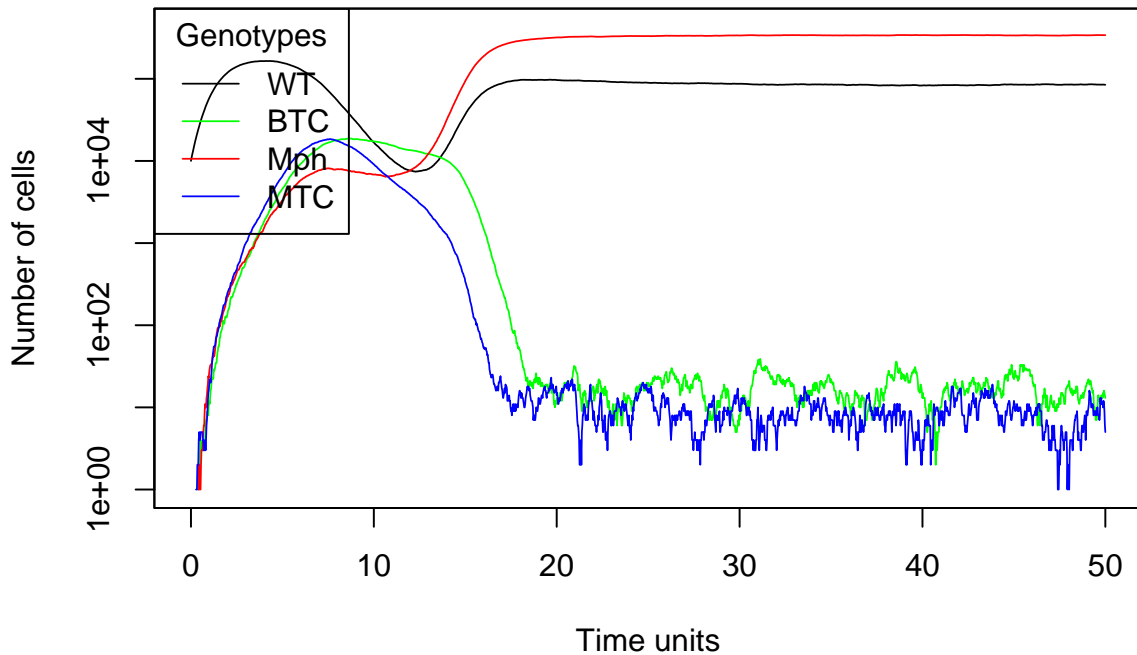
So, since the model seems correct, we create now the allFitnessEffects object and do the simulation. Here, we use the McFL model, with death density dependence in addition to the frequency dependence in the birth rates. In this case, we model three different situations: cancer being controlled, development of a non-metastatic cancer, and development of metastatic cancer.

11.8.1 Cancer kept under control

In this example, cD value is increased in order to represent a highly functioning immune system that helps fighting against cancer cells, while the cost of producing growth factor (cG) is low to allow a better fitness of non-cancer cells. This results in the fitness of wild-type cells and macrophages being kept in high levels, helping to control the proliferation of cancer cells, which is maintained under acceptable levels.

```
afe_3_a <- allFitnessEffects(
  genotFitness =
    create_fe(0.5, 4, 1, 0.2, 1, 0.5, 4),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(0.5, 4, 1,
## 0.2, 1, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
set.seed(2)

s_3_a <- oncoSimulIndiv(afe_3_a,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 50,
                        mu = 1e-4,
                        initSize = 10000,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
```

```
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue", "yellow"))
```

11.8.2 Development of a non-metastatic cancer

In this second scenario, bR and cMMTC are increased, promoting a higher proliferative capacity of cancer cells but increasing the cost of the ability to move by metastatic cells (MTC), while Cs is slightly decreased, allowing a better fitness of non-motile cells that have to share resources (WT and BTC). This situation leads to the appearance of a non-metastatic cancer due to the higher fitness of BTC cells, thanks to their absence of mobility cost and their increased proliferative capacity, specially in the absence of other cell types that compete for resources.

```
afe_3_a <- allFitnessEffects(
  genotFitness =
    create_fe(1, 4, 0.5, 1, 1.5, 1, 4),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(1, 4, 0.5, 1,
## 1.5, : v2 functionality detected. Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
set.seed(2)

s_3_a <- oncoSimulIndiv(afe_3_a,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 50,
                        mu = 1e-4,
                        initSize = 10000,
                        keepPhylog = FALSE,
                        seed = NULL,
```

```
                          errorHitMaxTries = FALSE,
                          errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```
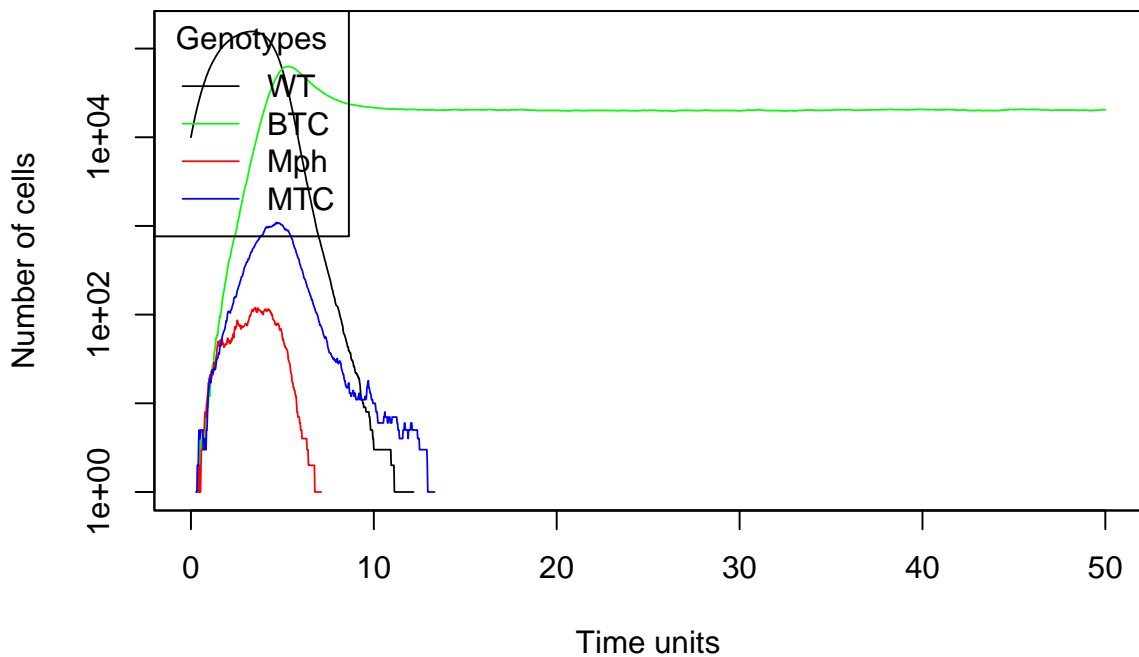
```
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue", "yellow"))
```


11.8.3 Development of a metastatic cancer

In this last example, cS value is increased, hindering the growth and fitness of non-motile cells that compete for space and resources, whereas the cost of mobility of metastatic cancer cells (cMMTC) is considerably reduced, thus favoring their proliferation. This scenario leads to the development of metastatic cancer, due to a rapid increase in the proliferation of metastatic cancer cells, thanks to their low mobility cost, and a slightly slower increase in the fitness of benign tumor cells, that have to compete with resources with the wild-type cells until their disappearance.

```
afe_3_a <- allFitnessEffects(
  genotFitness =
    create_fe(0.5, 4, 2, 0.5, 0.5, 1, 4),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(0.5, 4, 2,
## 0.5, 0.5, : v2 functionality detected. Adapting to v3
```

```
## functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
set.seed(2)

s_3_a <- oncoSimulIndiv(afe_3_a,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 50,
                        mu = 1e-4,
                        initSize = 10000,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##               to last version.
```
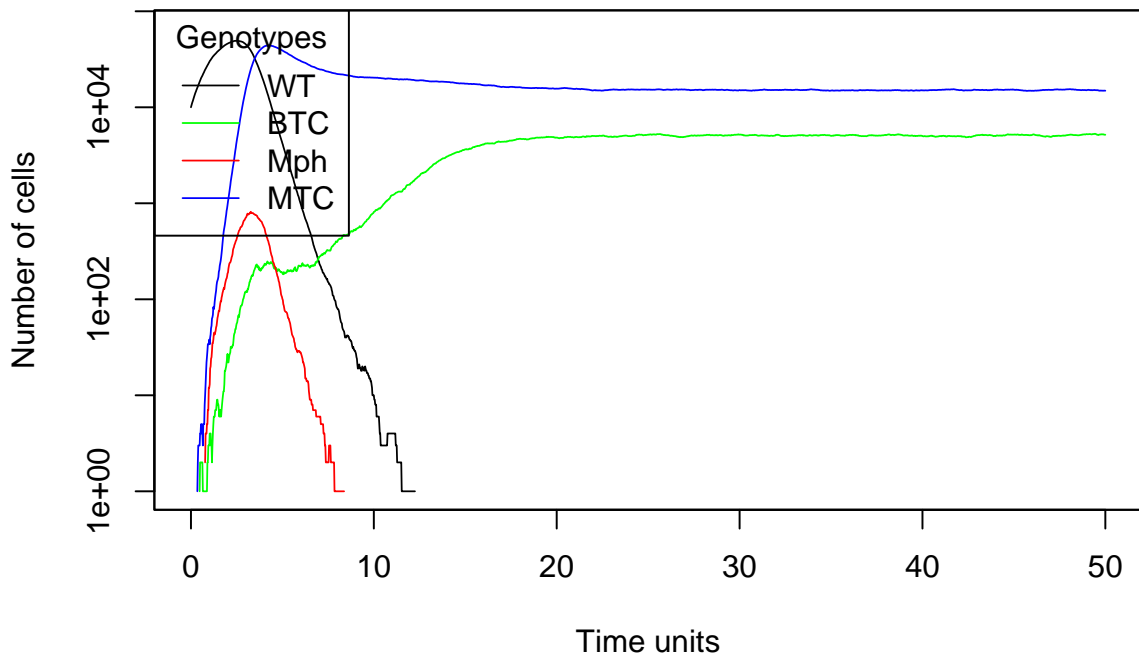
```
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue", "yellow"))
```

11.9 Improving the previous example. Modeling of breast cancer with the presence chemotherapy and resistance.

With the aim of obtaining a more reliable and representative picture of what happens in a situation of cancer development and treatment, we model a more complex scenario inproving the previous example, 11.8.

This alternative model considers the presence of seven cell types, four of which are the same that have been described in the previous model: the native cells (NC), macrophages (Mph), benign tumor cells (BTC) and metastatic cancer cells (MTC). Moreover, we include here the chemotherapy-resistant normal cells (R), chemotherapy-resistant benign tumor cells (BTC,R) and chemotherapy-resistant metastatic cancer cells (MTC,R).

Most of the costs and benefits of this new model are similar to those described in the previous situation. Native cells and macrophages, but also the chemotherapy-resistant normal cells, produce growth factor bG, which benefits all cell types. These three cells types are considered to have the same cost of producing the growth factor, cG. The macrophages and motile tumor cells (both MTC and MTC,R) have the ability to move, assuming that the cost of this ability is cM,Mph, cM,MTC and cM,MTC,R respectively. The native cells and benign tumor cells (both BTC and BTC,R) have no motile capacity and have to stay in place, competing for the available resources with other native and benign tumor cells, which comes at the cost cS for all these cell types. As occurred in the previous model, considering that cancer cells have a higher reproductive rate than native cells or macrophages, we add an additional benefit bR to the cancer cells. However, as they can also be destroyed and attacked by macrophages, we add an additional cost cD to the cancer cells. Moreover, in this model we add an additional factor Q that represents the effect of chemotherapeutic treatment, which will not have any effect on cells that have been able to develop resistance to it, but it will have a negative effect on the rest of the cell types, being especially important on cancer cells, as they have a higher rate of reproduction.

The payoff matrix will also introduce slight changes, which reproduced in next table:

Table 21: Payoff matrix adapted from Barton et al., 2018, 'Modeling of breast cancer through evolutionary game theory', Involve, a Journal of Mathematics, 11(4), 541-548; https://doi.org/10.2140/involve.2018.11.541 .

	MTC	Mph	NC	BTC	R	BTC,R	MTC,R
MTC	bR - cMMTC - Q	bR - cMMTC - cD + bG -Q	bR - cMMTC + bG -Q	bR - cMMTC - Q	bR - cMMTC + bG - Q	bR - cMMTC -Q	bR - cMMTC -Q

	MTC	Mph	NC	BTC	R	BTC,R	MTC,R
Mph	- cG - cMMph - 0.01 * Q	bG - cG - cMMph - 0.01 * Q	bG - cG - cMMph - 0.01 * Q	-cG - cMMph - 0.01 * Q	bG - cG - cMMph - 0.01 * Q	-cG - cMMph - 0.01 * Q	- cG - cMMph - 0.01 * Q
NC	- cG - 0.01 * Q	bG - cG - 0.01 * Q	bG - cG - cS - 0.01 * Q	- cG - cS - 0.01 * Q	bG - cG - cS - 0.01 * Q	- cG - cS - 0.01 * Q	- cG - 0.01 * Q
BTC	bR - Q	bR + bG - cD - Q	bR + bG - cS - Q	bR - cS - Q	bR + bG - cS - Q	bR - cS - Q	bR - Q
R	- cG	bG - cG	bG - cG - cS	- cG - cS	bG - cG - cS	- cG - cS	- cG
BTC,R	bR	bR + bG - cD	bR + bG - cS	bR - cS	bR + bG - cS	bR - cS	bR
MTC,R	bR - cMMTC	bR - cMMTC - cD + bG	bR - cMMTC + bG	bR - cMMTC	bR - cMMTC + bG	bR - cMMTC + bG	bR - cMMTC + bG

Overall, when the concentrations of the cells are QNC, QMph, QBTC, QMTC, QR, QBTC,R and QMTC,R, the net benefits (benefits minus the costs) to each cell type are:

- W(NC) = bG([NC] + [Mph] + [R]) - cG - cS([NC] + [BTC] + [R] + [BTC,R]) - 0.01*Q
- W(Mph) = bG([NC] + [Mph] + [R]) - cG - cMMph - 0.001*Q
- W(BTC) = bR + bG([NC] + [Mph] + [R]) - cS([NC] + [Mph] + [R] + [BTC,R]) - cD[Mph] - Q
- W(MTC) = bR + bG([NC] + [Mph] + [R]) - cMMTC - cD[Mph] - Q
- W(R) = bG ([NC] + [Mph] + [R]) - cG - cS ([NC] + [BTC] + [R] + [BTC,R])
- W(BTC,R) = bR + bG([NC] + [Mph] + [R]) - cS([NC] + [Mph] + [R] + [BTC,R]) - cD[Mph]
- W(MTC,R) = bR + bG([NC] + [Mph] + [R]) - cMMTC - cD[Mph]

Now, we will define a function to create the data frame of frequency-dependent fitnesses to allow modelling several situations. As in the previous model, we will consider the NC cell type as the WT and will assume that Mph, BTC and MTC cells are all derived from WT by a single mutation, as previously described. Likewise, we consider that the chemotherapy-resistant normal cells (R) are derived from a single mutation in a gene, say, R. Furthermore, we will assume that the two chemotherapy-resistant cancer cells (BTC,R and MTC,R) are all derived from WT by two different mutations.

```
create_fe <- function(cG, bG, cS, cMMph, cMMTC, bR, cD, Q,
                      gt = c("WT", "BTC", "R", "MTC", "Mph", "BTC,R", "MTC,R")) {
  data.frame(Genotype = gt,
             Fitness = c(

               paste0("1 + ", bG, "(f_ + f_R + f_Mph) - ", cG, " - ", cS, "(f_ + f_
                      "0.01*", Q),
               paste0("1 + ", bR, " + ", bG, "(f_ + f_R + f_Mph) - ", cS, " (f_ + f
                      cD , " * f_Mph -", Q),
               paste0("1 + ", bG, "(f_ + f_R + f_Mph) - ", cG, " - ", cS, "(f_ + f_
               paste0("1 + ", bR, " + ", bG, " *(f_ + f_R + f_Mph) -", cMMTC, " - "
                      cD , " * f_Mph -", Q),
               paste0("1 + ", bG, "(f_ + f_R + f_Mph) - ", cG, " - ", cMMph, "- 0.0
               paste0("1 + ", bR, " + ", bG, "(f_ + f_R + f_Mph) - ", cS, " (f_ + f
                      cD , " * f_Mph"),
               paste0("1 + ", bR, " + ", bG, " *(f_ + f_R + f_Mph) -", cMMTC, " - "
                      cD , " * f_Mph")
             ),
             stringsAsFactors = FALSE)
}
```

We verify that we recover Table 21 :

```
create_fe("cG", "bG","cS", "cMMph", "cMMTC", "bR", "cD", "Q")
##   Genotype
## 1       WT
## 2      BTC
## 3        R
```

```
## 4       MTC
## 5       Mph
## 6     BTC,R
## 7     MTC,R
##                                                                    Fitne
## 1           1 + bG(f_ + f_R + f_Mph) - cG - cS(f_ + f_BTC + f_R + f_BTC_R) -0.01
## 2 1 + bR + bG(f_ + f_R + f_Mph) - cS (f_ + f_BTC + f_R + f_BTC_R) -cD * f_Mph
## 3               1 + bG(f_ + f_R + f_Mph) - cG - cS(f_ + f_BTC + f_R + f_BTC_
## 4                     1 + bR + bG *(f_ + f_R + f_Mph) -cMMTC - cD * f_Mph
## 5                            1 + bG(f_ + f_R + f_Mph) - cG - cMMph- 0.01
## 6   1 + bR + bG(f_ + f_R + f_Mph) - cS (f_ + f_BTC + f_R + f_BTC_R) -cD * f_M
## 7                            1 + bR + bG *(f_ + f_R + f_Mph) -cMMTC - cD * f_M
```

We check once again that we have correctly specified the different parameters executing evalAllGenotypes. We are trying the same case than in the previous example, having wild-type cells, macrophages and both cancer cells. But in this case, we are introducing the parameter "Q" (chemotherapy), with a value of 5.

```
evalAllGenotypes(allFitnessEffects(genotFitness =
                                   create_fe(2,5,1,0.8,1,1,9,5),
                                   frequencyDependentFitness = TRUE,
                                   frequencyType = "rel"),
              spPopSizes = c(WT = 1000, BTC = 100, R = 0,
                             MTC = 100, Mph = 1000,
                             "BTC, R" = 0, "MTC, R" = 0))
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##           Genotype Fitness
## 1               WT   2.995
## 2              BTC   0.000
## 3              Mph   2.695
## 4              MTC   0.000
## 5                R   3.045
## 6         BTC, Mph   0.000
## 7         BTC, MTC   0.000
## 8           BTC, R   1.955
## 9         Mph, MTC   0.000
## 10          Mph, R   0.000
## 11          MTC, R   1.455
## 12    BTC, Mph, MTC   0.000
## 13      BTC, Mph, R   0.000
## 14      BTC, MTC, R   0.000
```

```
## 15      Mph, MTC, R    0.000
## 16 BTC, Mph, MTC, R    0.000
```

As we can see, BTC and MTC have a fitness of 0, so the will not proliferate. But cancer cells mutants with a double mutation (BTC, R and MTC, R) are resistant to chemotherapy, so they can still grow. This is what we were expecting to happen, so we started with the simulation.

Now, we create the allFitnessEffects object and do the simulation using the McFL model. In this case, we specify gene-specific mutations rates, so we can model a more realistic scenario where the appearence of chemotherapy-resistant cells is hindered. Once again, we simulate different situations by changing the values of the different parameters.

11.9.1   Absence of chemotherapy

In the first scenario, we simulate a situation where no chemotherapy is applied. In this condition, there is a very low R mutation rate, which hampers the proliferation of chemotherapy-resistant cells. A fast appearance of macrophages is observed, as well as non-resistant tumor cells. As there is no chemoterapy treatment, the fitness of non-resistant tumor cells is favoured against macrophages, leading to the appearance of cancer, where BTC cells show the greatest fitness due to the mobility cost of MTC cells.

```
afe_3_a <- allFitnessEffects(
  genotFitness =
    create_fe(2, 5, 1, 0.8, 1, 1, 9, 0),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
```

```
#Set mutation rates
muvar2 <- c("Mph" = 1e-2, "BTC" = 1e-3, "MTC"=1e-3, "R" = 1e-7)

set.seed(2)
s_3_a <- oncoSimulIndiv(afe_3_a,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 20,
                        mu = muvar2,
                        initSize = 10000,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
```

```
##                to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##                to last version.
```

```
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue", "pink", "orange", "brown"))
```
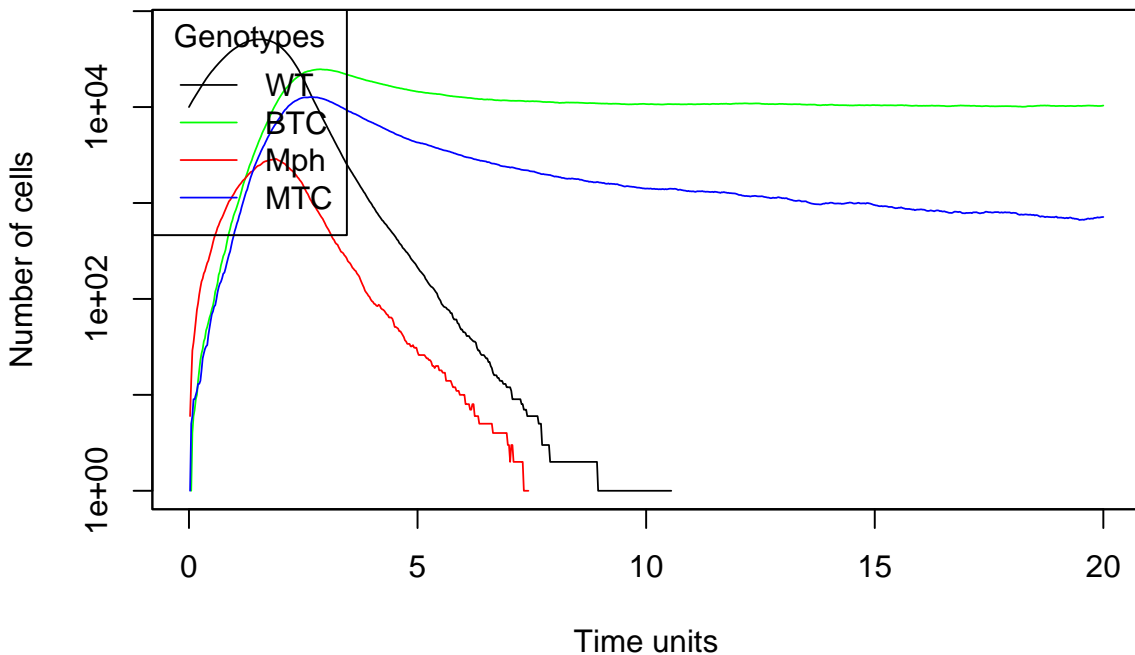

11.9.2   Chemotherapy with low R mutation rate

In the second scenario, we perform a simulation including chemotherapy as a treatment. However, here we maintain a low R mutation rate, hampering once again the appearance of chemotherapy-resistant cells. This situation can be reflecting the application of a combination chemotherapy that reduces or limits the appearance of resistance. We observe that the fitness of wild-type cells and macrophages increases rapidly and remains elevated, while tumor cells undergo some proliferation at first, but it remains under control over time thanks to the negative effect on them of chemotherapy.

```
afe_3_a <- allFitnessEffects(
  genotFitness =
    create_fe(2, 5, 1, 0.8, 1, 1, 9, 2),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
```

```
muvar2 <- c("Mph" = 1e-2, "BTC" = 1e-3, "MTC"=1e-3, "R" = 1e-7)
```

```
set.seed(2)

s_3_a <- oncoSimulIndiv(afe_3_a,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 20,
                        mu = muvar2,
                        initSize = 10000,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
```
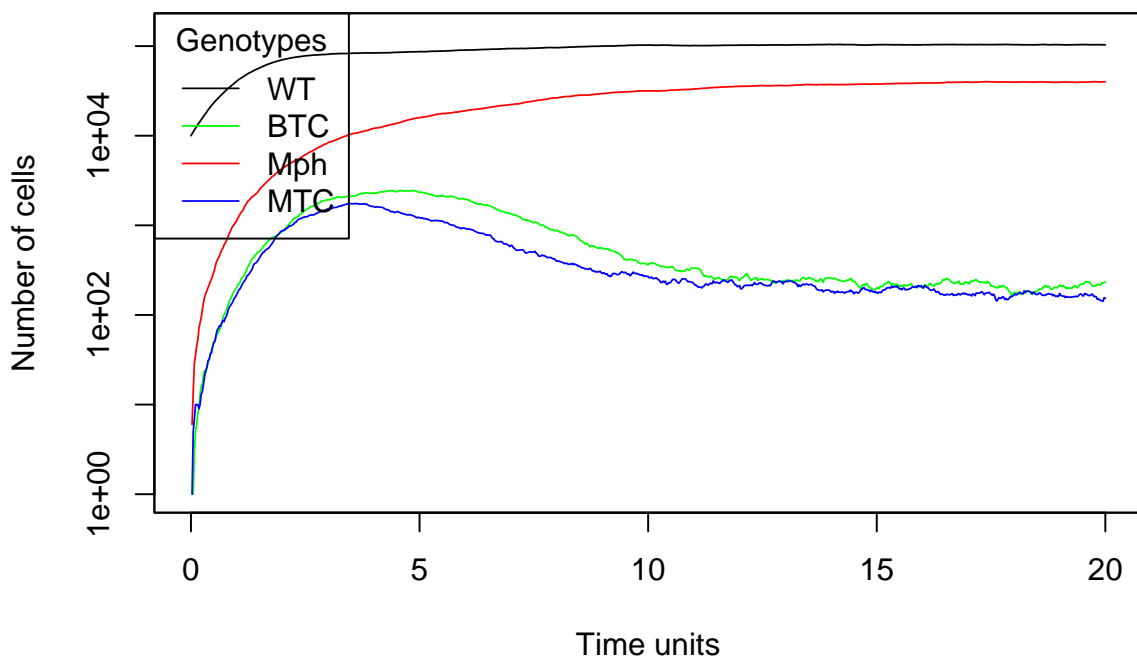
```
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue", "pink", "orange", "brown"))
```


11.9.3 Chemotherapy with considerable R mutation rate

Finally, we simulate a scenario in the presence of chemotherapy as a treatment and also a considerable R mutation rate. This would allow the appearance of chemotherapy-resistant cells. This simulation reflects the situation of using chemotherapy treatments against which tumor cells develop resistance. Here, we observe a similar fitness evolution of wild-type cells and macrophages to the previous example, rapidly

316

increasing their population and remaining elevated for a period of time. The
proliferation of non-resistant tumor cells is also maintained under acceptable levels
thanks to the effect of chemotherapy. However, due to the increased R mutation
rate, chemotherapy-resistant cells begin to appear in low levels, until the fitness of
chemotherapy-resistant benign tumor cells stats to increase considerably, leading
to the disappearance of the other cell types and allowing the development of a
non-metastatic cancer.

```r
afe_3_a <- allFitnessEffects(
  genotFitness =
    create_fe(2, 5, 1, 0.8, 1, 1, 9, 2),
  frequencyDependentFitness = TRUE,
  frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = create_fe(2, 5, 1, 0.8,
## 1, 1, : v2 functionality detected. Adapting to v3 functionality.
```

```r
muvar2 <- c("Mph" = 1e-2, "BTC" = 1e-3, "MTC"=1e-3, "R" = 1e-5)
```

```r
set.seed(2)
```

```r
s_3_a <- oncoSimulIndiv(afe_3_a,
                        model = "McFL",
                        onlyCancer = FALSE,
                        finalTime = 20, ## short for speed; increase for "real"
                        mu = muvar2,
                        initSize = 10000,
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
```

```r
plot(s_3_a, show = "genotypes", type = "line",
     col = c("black", "green", "red", "blue", "pink", "orange", "brown"))
```
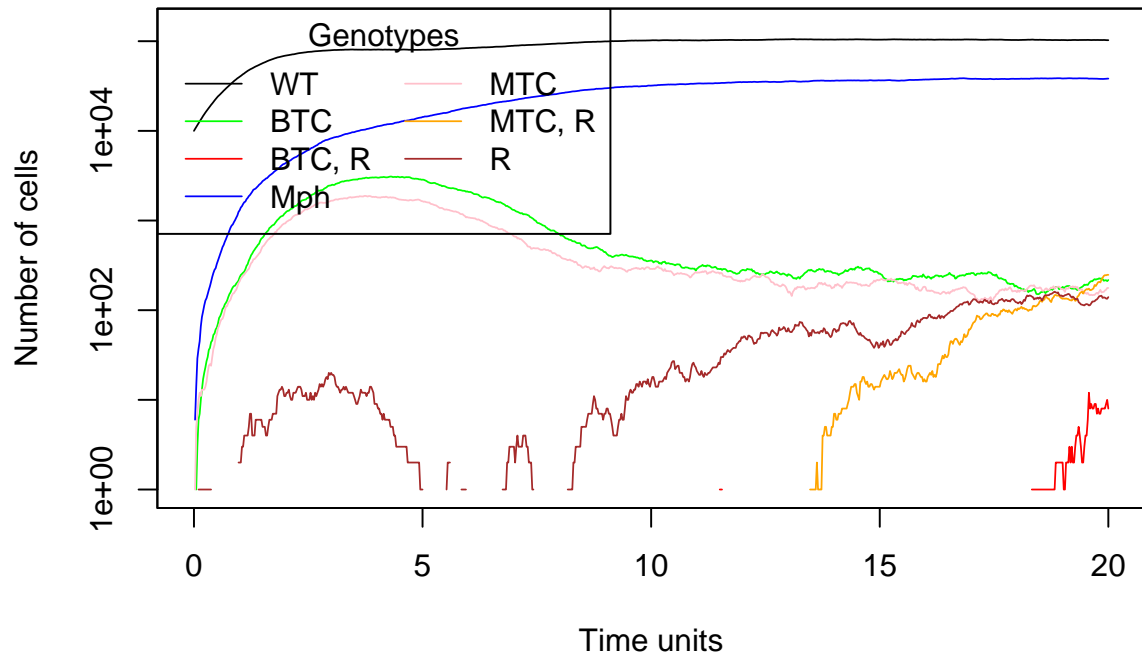
12 Death and Birth specification

With the changes introduced in mid 2021 by Alberto González Klein, the user is allowed to specify arbitrary death rates in the same way fitness could be specified, instead of using the fixed death rate given by the chosen model. However, the introduction of explicit death rates makes the fitness nomenclature confusing and inconsistent. Because of this, fitness has been renamed to birth.

12.1 Changes in nomenclature

The changes only concern the explicit mapping. The `Fitness` column in the mapping has been renamed to `Birth`. Also, the parameter `frequencyDependentFitness` in the function `allFitnessEffects` has been renamed to `frequencyDependentBirth`. The next example will illustrate the new nomentaclature.

```
m4 <- data.frame(Genotype = c("WT", "A", "B", "A, B"), Birth = c(1, 2, 3, 4))

fem4 <- allFitnessEffects(genotFitness = m4, frequencyDependentBirth = FALSE)

evalAllGenotypes(fem4)
##    Genotype Birth
## 1         A     2
## 2         B     3
## 3      A, B     4
```

Note that the parameter `frequencyDependentBirth` has been specified in order to show the changes. However, it continues to default as FALSE.

Despite this change, the old nomenclature continues to be valid and it will work in every function of the package. Users can use already created objects using the fitness nomenclature and can also create new objects with the old nomenclature.

12.2 Explicit mapping of genotypes to death rates

The mapping is done analogously to the genotypes to birth rates. However, death rates cannot be mapped if birth rates are not being mapped as well. This is to make it compatible with the pre-existing behavior. Either we specify the death rates ourselves or the model we choose will determine the death rates. A new parameter has been introduced to indicate the `allFitnessEffects` function that death rates are being specified. This parameter is `deathSpec`.

Also, the death rates can be frequency dependent or not. This is determined by the new parameter `frequencyDependentDeath` in the function 'allFitnessEffects. The following example shows how to explicitly map genotypes to death rates.

```
m4 <- data.frame(Genotype = c("WT", "A", "B", "A, B"),
                 Birth = c(1, 2, 3, 4),
                 Death = c(1, 2, 3, 4))
```

```
fem4 <- allFitnessEffects(genotFitness = m4,
                          frequencyDependentBirth = FALSE,
                          frequencyDependentDeath = FALSE,
                          deathSpec = TRUE)


evalAllGenotypes(fem4)
## Warning in evalAllGenotypesORMut(fmEffects = fitnessEffects, order
## = order, : Death is specified in fitnessLandscape. Assuming
## arbitrary model
##   Genotype Birth Death
## 1        A     2     2
## 2        B     3     3
## 3     A, B     4     4
```

In order to simulate using `oncoSimulIndiv`, we must specify a model. A new model has been introduced to indicate the simulation that the death rates are present in the mapping and that we do not use any of the already existing models. The new model is `Arb`. The next example will show a simulation using the `Arb` model.

```
G_fe_LVm <- function(r1, r2, K1, K2, a_12, a_21, gt = c("S1", "S2")) {
    data.frame(Genotype = gt,
               Birth = c(paste0(r1, "-", r1, "*(", a_12, "*n_", gt[2], ")/", K1), r
               Death = c(paste0(r1, "*(n_", gt[1], ")/", K1),
                         paste0(r2, "*(n_", gt[2], "+", a_21, "*n_", gt[1], ")/", K
}
fe_pred_prey <- allFitnessEffects(
        genotFitness = G_fe_LVm(1.4, 1.5, 4000, 10000, -0.5, 1.1, gt = c("Predator"
        frequencyDependentBirth = TRUE,
        frequencyDependentDeath = TRUE,
        deathSpec = TRUE)
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
```

```
s_pred_preym <- oncoSimulIndiv(fe_pred_prey, model = "Arb",
                               initMutant = c("Predator", "Prey"),
                               initSize = c(1000, 1000),
                               onlyCancer = FALSE,
                               finalTime = 75, mu = 1e-3,
                               keepPhylog = FALSE, seed = NULL,
                               errorHitMaxTries = FALSE,
                               errorHitWallTime = FALSE,
                               keepEvery = 1)
```

```
plot(s_pred_preym, show="genotypes", type="line", log = "")
```
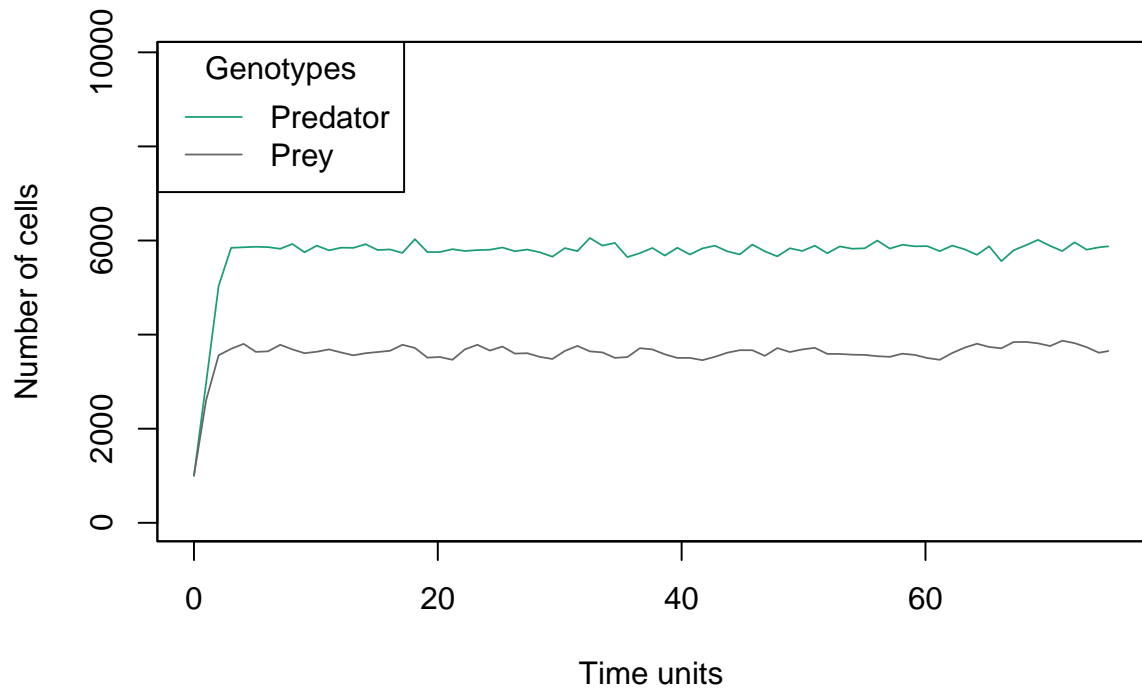
13 Simulating with constant total population size

There are some models in which it is necessary to maintain the total size of the population constant, such as Moran models Moran (1962). In order to allow this types of models to be simulated, the `Const` model must be used, as shown in the example below.

```
H_D_fitness <- function(c, v, gt = c("H", "D")) {
                data.frame(Genotype = gt,
                           Birth = c(
                               paste0("max(1e-5, f_H *", (v-c)/2, "+ f_D *", v
                           paste0("f_D *", v/2)))
}


HD_eq <- allFitnessEffects(
                genotFitness = H_D_fitness(10, 4, gt = c("H", "D")),
                frequencyDependentBirth = TRUE,
                frequencyType = "rel")
## All single-gene genotypes as input to to_genotFitness_std
```

```
osi_eq <- oncoSimulIndiv(HD_eq, model = "Const",
                          onlyCancer = FALSE, finalTime = 50,
                          mu = 1e-6, initSize = c(2000, 2000),
                          initMutant = c("H", "D"), keepPhylog = FALSE,
                          seed = NULL, errorHitMaxTries = FALSE,
                          errorHitWallTime = FALSE)
```

```
osi_eq
##
## Individual OncoSimul trajectory with call:
##  oncoSimulIndiv(fp = HD_eq, model = "Const", mu = 1e-06, initSize = c(2000,
##     2000), finalTime = 50, onlyCancer = FALSE, keepPhylog = FALSE,
##     errorHitWallTime = FALSE, errorHitMaxTries = FALSE, initMutant = c("H",
##        "D"), seed = NULL)
##
##   NumClones TotalPopSize LargestClone MaxNumDrivers MaxDriversLast
## 1         2         4021         2402             0              0
##   NumDriversLargestPop TotalPresentDrivers FinalTime NumIter
## 1                    0                   0        50    2001
##   HittedWallTime HittedMaxTries errorMF minDMratio minBMratio
## 1          FALSE          FALSE      NA     612244    1004002
##   OccurringDrivers
## 1
##
## Final population composition:
##   Genotype    N
## 1        D 2402
```

322

```
## 2          H 1619
```

```
## This try should not be necessary, except
## the code above seems to produce an empty object
## in the BioC kjohnson3 (maOS 13.6.5, arm64) machine.
## See below, "Help debugging"
try(plot(osi_eq, show="genotypes", ylim=c(1, 5000)))
```
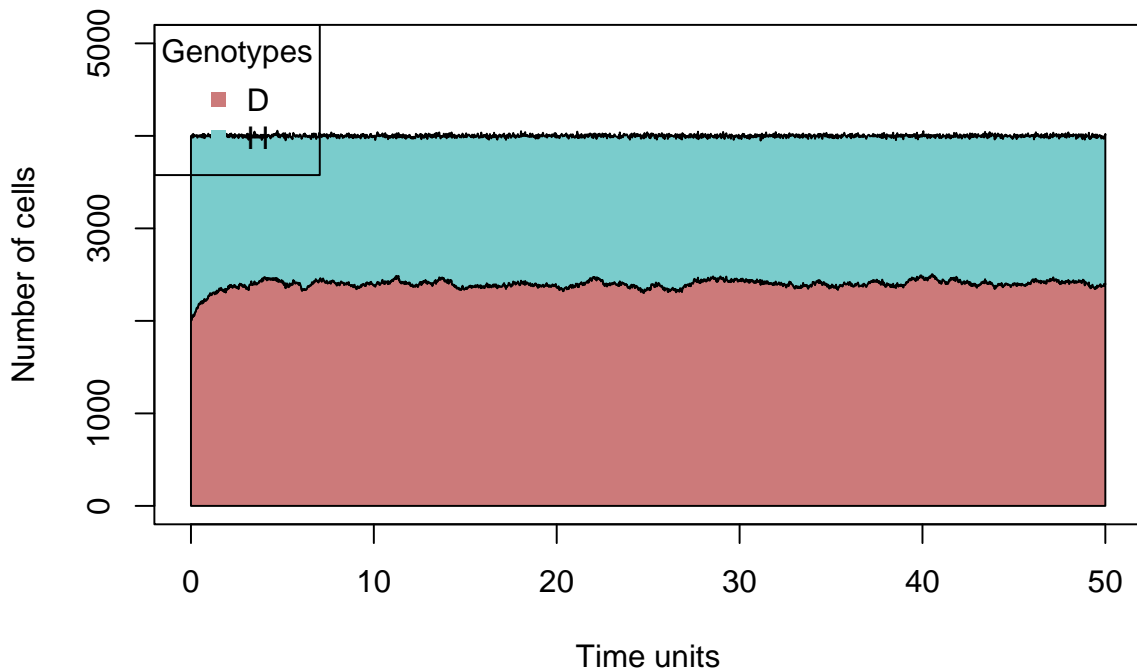

(Help debugging: If you are running this in a macOS 13.6.5 with arm64 or, more generally, if you see the above plot produce nothing, I'd appreciate if you can let me know. Email me with the output of the run, specially what `print(osi_eq)` and `summary(osi_eq)` yield. And try running the code again, to see if this happens consistently regardless of the seed.)

When using this model, we can either specify the death rates or not. In the case that we specify them, the death rates will be corrected in order to maintain the desired size of population (which is the initial size of the population). On the other hand, if we do not specify the death rates they will be set to the required value.

14 Simulating therapeutic interventions and adaptive therapy, and using user-defined variables

(Most of the code that implements this functionality has been added by Javier Muñoz Haro and Javier López Cano. Authors for specific examples are listed in the corresponding places.)

14.1 Interventions

(*Note that the examples below are not used because of their biological realism, but rather to show some key features of the software*)

OncoSimulR also allows the user to specify interventions within the simulation. Interventions will allow the user to manipulate different scenarios in the simulation, by reducing the population of a specific genotype or the total population. In R-terms a intervention a list of lists, where each element of the list must have the following attributes:

- **ID**: This will be the identifier of the intervention. This attribute must be unique in the list.
- **Trigger**: A condition that "triggers" an action over the simulation. It is a boolean evaluation, meaning this will give a TRUE or FALSE result.
- **WhatHappens**: The action that will affect the simulation.
- **Repetitions**: How many times do we want the intervention to affect the population (Trigger must be activated in order to execute a Repetition).
- **Periodicity**: How much time must pass between to repeat the intervention.

Here we have an example of how the user can specify a list of interventions:

```
interventions <- list(
    list(ID           = "i2",
        Trigger       = "(N > 1e6) & (T > 100)",
        WhatHappens   = "N = 0.001 * N",
        Repetitions   = 7,
        Periodicity    = Inf
    ),
    list(ID           = "i1",
        Trigger       = "(T > 10)",
        WhatHappens   = "N = 0.3 * N",
        Periodicity   = 10,
        Repetitions   = 0
    ),
    list(ID           = "i3",
        Trigger       = "(T > 1) & (T < 200)",
        WhatHappens   = "n_A = n_A * 0,3 / n_C",
        Repetitions   = Inf,
        Periodicity    = 10
    ),
    list(ID           = "i5",
        Trigger       = "(N > 1e8) & (T> 1.2)",
        WhatHappens   = "n_A_B = n_B * 0,3 / n_SRL",
        Repetitions   = 0,
        Periodicity    = Inf
    )
)
```

(*Note: In a intervention, if Periodicity is specified as Inf, then the intervention will only execute once. An intervention that specifies will only execute if this expression:* **(T − (Last time executed) >= Periodicity** *turns out to be TRUE.*)

As is seen in the example above, interventions can depend on the current time of The simulation (T), the current total population (N) or some genotype population n_(genotype name), and can be combined as the user might want, depending on the mean of the intervention. This is possible thanks to [Exprtk library] (http://www.partow.net/programming/exprtk/), this library allow complex expressions to be specified.

In order to specify interventions, the user must call `createInterventions` function. This function will adapt the different attributes specified to something C++ will understand. But first, a `fitnessEffects` object must be defined, for example:

```
fa1 <- data.frame(Genotype = c("WT", "A", "B"),
                  Fitness = c("n_*0",
                              "1.5",
                              "1"))


afd3 <- allFitnessEffects(genotFitness = fa1,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = fa1,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

Note that, for now, the fitness must have `frequencyDependentFitness` as TRUE and `frequencyType`as "abs" (it can be).

Once this is specified, we call `createInterventions`:

```
interventions <- createInterventions(interventions, afd3)
## [1] "Checking intervention: i2"
## [1] "Checking intervention: i1"
## [1] "Checking intervention: i3"
## [1] "Checking intervention: i5"
```

Where the first argument is the list of lists previously defined, and the second argument is the `fitnessEffects` object.

Finally, once the intervention is "created" by the `createInterventions` function, the object returned can be passed as an argument to the `oncoSimul*` function, like this example below. More detailed examples are shown next (and, for speed creating the vignette, we do not execute the next code chunk).

```
ep2 <- oncoSimulIndiv(
    afd3,
    model = "Exp",
```

```
    mu = 1e-4,
    sampleEvery = 0.001,
    initSize = c(20000, 20000),
    initMutant = c("A", "B"),
    finalTime = 5.2,
    onlyCancer = FALSE,
    interventions = interventions
)
```

14.2   A first example with interventions

As is stated before, the examples provided will show that the software works. In this example it will be shown how a simple intervention can affect the way the simulation develops. First, we define (as usual) the dataframe that associates the genotypes with their fitness. In this case in particular, we will define an scenario where the genotype B has a higher fitness than the other genotypes in the population:

```
df3x <- data.frame(Genotype = c("WT", "B", "A", "B, A", "C, A"),
                   Fitness = c("0*n_",
                               "1.5",
                               "1.002",
                               "1.003",
                               "1.004"))


afd3 <- allFitnessEffects(genotFitness = df3x,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = df3x,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
ex1 <- oncoSimulIndiv(
                 afd3,
                 model = "McFLD",
                 mu = 1e-4,
                 sampleEvery = 0.01,
                 initSize = c(20000, 20000),
                 initMutant = c("A", "B"),
                 finalTime = 10,
                 onlyCancer = FALSE
                 )
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
```
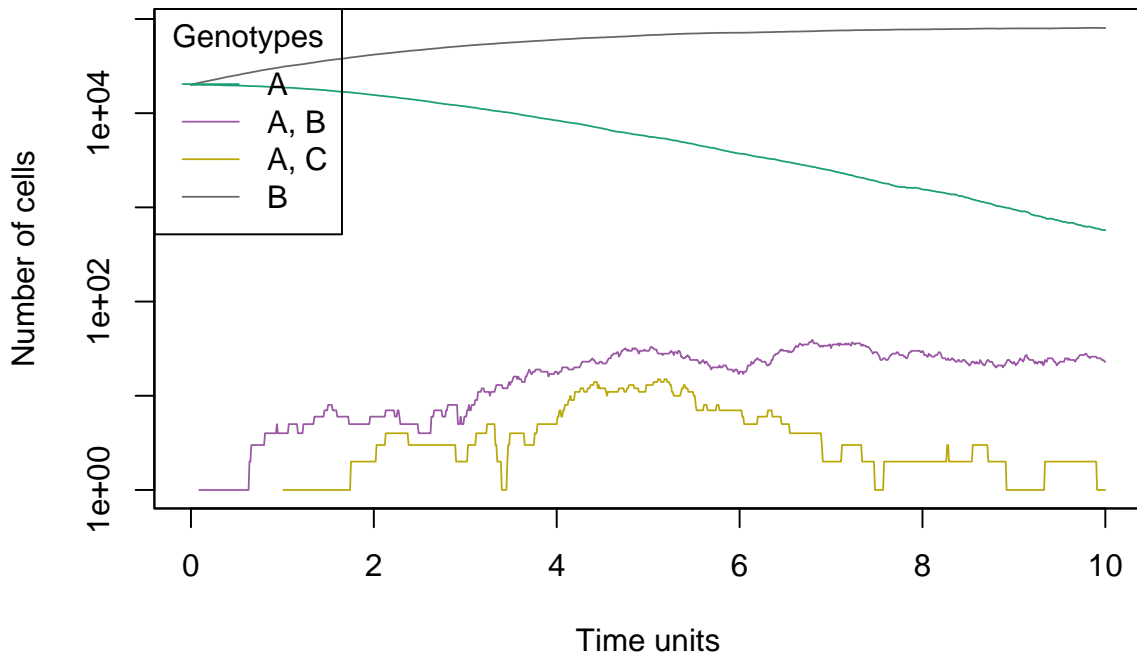
```
plot(ex1, show="genotypes", type = "line")
```


As expected, the B genotype dominates the population but ¿what if we periodically we start to decrement the dominant genotype?

First, we define that intervention, in this case, the intervention will decrement the B population by 88% each 0.07 time units:

```
interventions <- list(
    list(ID           = "intOverB",
         Trigger       = "(T >= 5)",
         WhatHappens   = "n_B = n_B * 0.88",
         Repetitions   = Inf,
         Periodicity   = 0.07
    ))


interventions <- createInterventions(interventions, afd3)
## [1] "Checking intervention: intOverB"
```

Note that we specify as `Inf` the amount of repetitions that this interventions will execute. This means that, until `finalTime` is reached, the intervention will be executing each 0.5 time units. Another detail that is worth noticing is the `Trigger` attribute, that defines in this case, that until the current time of the simulation T reaches 5 time units, the intervention will not execute over the simulation.

Then, we run the simulation again, but this time with the interventions specified:

```
ex1_with_ints <- oncoSimulIndiv(
                  afd3,
                  model = "McFLD",
```

327

```
                    mu = 1e-4,
                    sampleEvery = 0.01,
                    initSize = c(20000, 20000),
                    initMutant = c("A", "B"),
                    finalTime = 10,
                    onlyCancer = FALSE,
                    interventions = interventions)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```
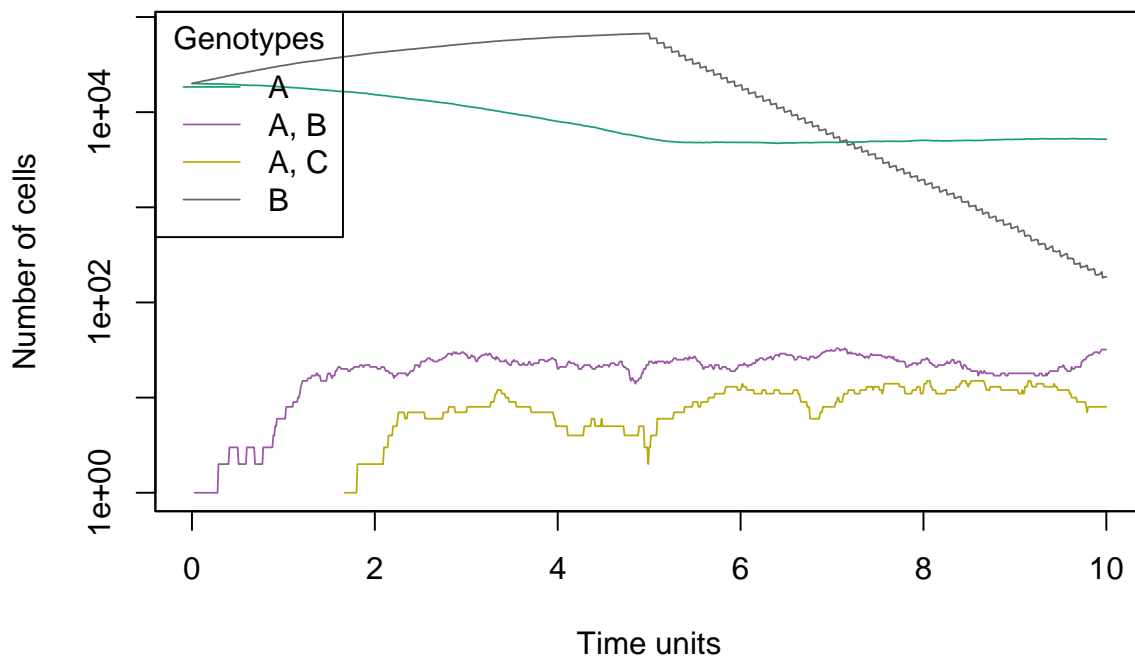
```
## This try should not be necessary, except
## the code above seems to produce an empty object
## in the BioC kjohnson3 (maOS 13.6.5, arm64) machine.
## See below, "Help debugging"
try(plot(ex1_with_ints, show="genotypes", type = "line"))
```


(Help debugging: If you are running this in a macOS 13.6.5 with arm64 or, more generally, if you see the above plot produce nothing, I'd appreciate if you can let me know. Email me with the output of the run, specially what `print(ex1_with_ints)` and `summary(ex1_with_ints)` yield. And try running the code again, to see if this happens consistently regardless of the seed.)

Where it is shown that, once we reach **T=5**, the population of B decreases, and, as a result of that, since genotype A does not encounter any other competition, starts to expanding the population.

14.3 Intervening over the total population

In the previous example, the intervention is specified so only one genotype is affected by it, but ¿can we define interventions that affect the total population?

First, we define the scenario where the intervention will operate. In these case we take the example from the example @**??**fdfabs), where genotype B has a birth rate less than 1, unless genotype A exists in the population. Where create the scenario:

```
gffd3 <- data.frame(Genotype = c("WT", "A", "B"),
                    Fitness = c("1",
                    "1 + 0.25 * (n_B > 0)",
                    ".9 + 0.4 * (n_A > 0)"
                    ))
afd3 <- allFitnessEffects(genotFitness = gffd3,
                            frequencyDependentFitness = TRUE,
                            frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = gffd3,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below
data(osi)

if (FALSE) {
osi <- oncoSimulIndiv( afd3,
                        model = "McFLD",
                        onlyCancer = FALSE,
                        finalTime = 200,
                        mu = 1e-4,
                        initSize = 5000,
                        sampleEvery = 0.001,
                        keepEvery = 1)
}
```

Then, we plot the result:

```
plot(osi, show = "genotypes", type = "line")
```
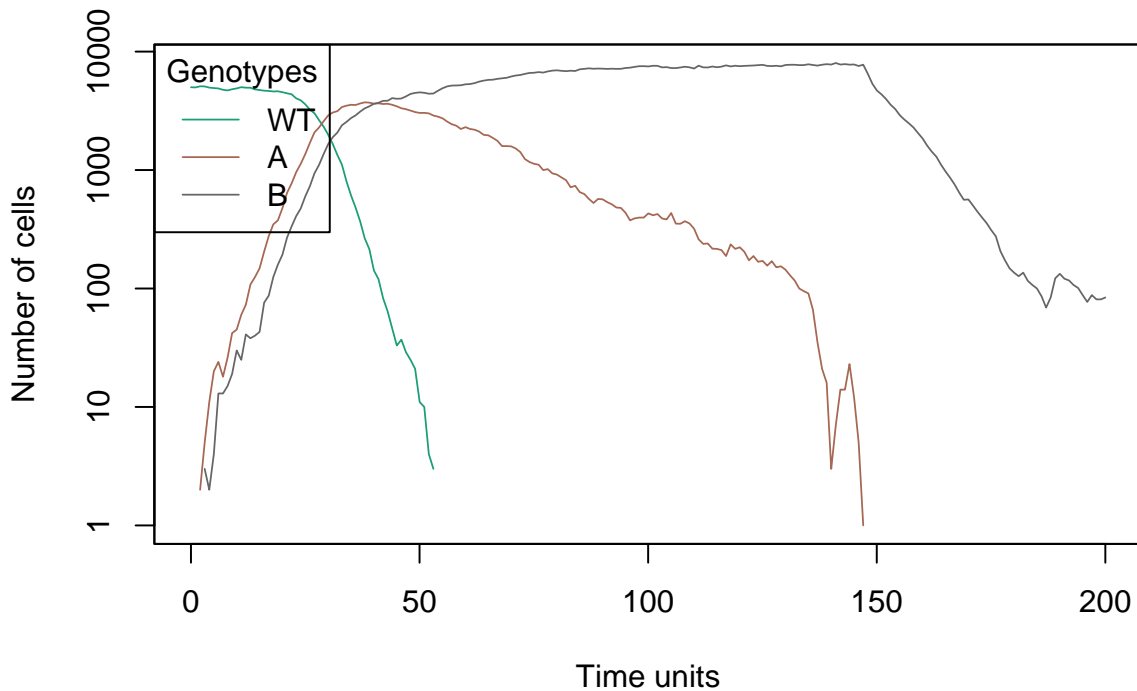
Now, we want to intervene over the total population, so we define the following intervention:

```r
intervention_tot_pop = list(
        list(
            ID           = "intOverTotPop",
            Trigger      = "T > 40",
            WhatHappens  = "N = N * 0.2",
            Repetitions  = 2,
            Periodicity  = 20
        )
    )

    intervention_tot_pop <- createInterventions(intervention_tot_pop, afd3)
## [1] "Checking intervention: intOverTotPop"
```

Where the intervention will start at T > 40, reducing the total population to the 20% of the original value. This intervention will be executed with a maximum of 2 repetitions (3 in total if the conditions are given) with a periodicity of 20 time units. Now, we re-run the simulation, but this time specifying the interventions:

```r
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below
data(osi_with_ints)

if(FALSE) {
osi_with_ints <- oncoSimulIndiv( afd3,
                        model = "McFLD",
                        onlyCancer = FALSE,
```
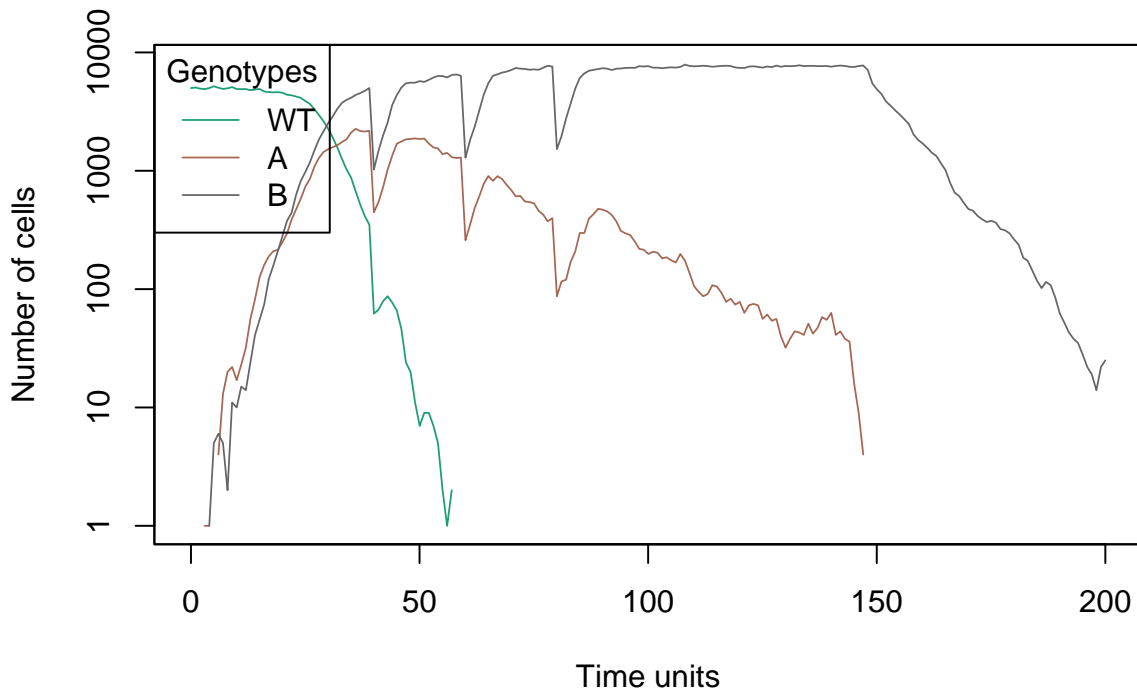
330

```
                    finalTime = 200,
                    mu = 1e-4,
                    initSize = 5000,
                    sampleEvery = 0.001,
                    interventions = intervention_tot_pop)
}

plot(osi_with_ints, show = "genotypes", type = "line")
```


(*Note that, this simulation will take long to run, since* `sampleEvery` *is quite small.*)

As it can be seen, when T=40, T=60 and T=80, the total population decrements, but let's take a closer look and see what `pops.by.time` in those instants:

```
osi_with_ints$pops.by.time[39:42, ]
##       [,1] [,2] [,3] [,4]
## [1,] 38.02  415 2147 4727
## [2,] 39.02  350 2174 5002
## [3,] 40.02   62  446 1024
## [4,] 41.02   67  544 1475
```

Here, the first column represents the time units, the second represents the population of the wild-type (WT) genotype for a given time, the third column represents the population of genotype A for a given time and the forth column represents the population for the B genotype. As it can be seen, when the current time of the simulation is really close to 40, the intervention happens, reducing the population to the 20%.

331

```
pre_int_tot_pop = osi_with_ints$PerSampleStats[40, 1]
post_int_tot_pop = osi_with_ints$PerSampleStats[41, 1]

## If you did not remember about PerSampleStats
## you could add all except the first column of pops.by.time
## to get the total population sizes.
percentage_eliminated = (post_int_tot_pop/pre_int_tot_pop)*100

paste0("The percentage of population has decreased by ", percentage_eliminated, "%"
## [1] "The percentage of population has decreased by 20.3560988572947%"
```

14.3.1 Differences between intervening on the total population or over specific genotypes: when do each occur?

Suppose an intervention that happens at time unit 10 (and, for the sake of simplicity, suppose we have set `sampleEvery = 1`). When, in the "What Happens" you specify something like

```
N = 0.2 * N
```

the total population size at time unit 10 is 0.2 times the population you had at the immediately previous sampling period; in this case, total population size at time 10 will be 0.2 the total population size at time 9. You can easily check this looking at the `pops.by.time` object (beware if you are not keeping all the sampling period; in case of doubt, and if you want to check this, make sure `keepEvery` is set to `sampleEvery`).

If you do, instead,

```
n_A = 0.2 * n_A
```

you will not see that `n_A` at time 10 is 0.2 `n_A` at time 9. The way the code works is: after we have done all the updates, etc, we change the `n_A` by the requested one. Thus, `n_A` at time 10 is not 0.2 the `n_A` at time 9, but 0.2 the `n_A` that you would have seen at time 10 had you not done an intervention.

14.4 Intervening in Rock-Paper-Scissors model in bacterial comunity

This example is taken from the example @**??**rockscissors) inspired by Kerr et al. (2002). Here it is described the relationship between 3 strains of *Escherichia coli* bacteria, that turns out to be very similar to a rock-paper-scissors game. (It is strongly recommended to visit that example before trying to understand this one.)

We know that the equations that model the growth of the different strains are these:

$$W\left(WT\right) = 1 + af_R - bf_C \tag{20}$$

$$W\left(C\right) = 1 + bf_{WT} - cf_R \tag{21}$$

$$W\left(R\right) = 1 + cf_C - af_{WT} \tag{22}$$

$$\tag{23}$$

where f_{WT}, f_C and f_R are the frequencies of WT, C and R, respectively. Being WT the wild-type, C the strain that produces colicin and R the strain that is resistant to the colicin.

We create the equations and run the simulation:

```
crs <- function (a, b, c){
      data.frame(Genotype = c("WT", "C", "R"),
      Fitness = c(paste0("1 + ", a, " * n_R/N - ", b, " * n_C/N"),
      paste0("1 + ", b, " * n_/N - ", c, " * n_R/N"),
      paste0("1 + ", c, " * n_C/N - ", a, " * n_/N")
      ))
   }


afcrs1 <- allFitnessEffects(genotFitness = crs(1, 1, 1),
frequencyDependentFitness = TRUE,
frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = crs(1, 1, 1),
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
resultscrs1_noints <- oncoSimulIndiv(afcrs1,
                       model = "McFL",
                       finalTime = 25,
                       mu = 1e-2,
                       initSize = 4000,
                       onlyCancer = FALSE,
                       keepPhylog = FALSE,
                       seed = NULL)
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
```

```
plot(resultscrs1_noints, show="genotypes", type = "line")
```
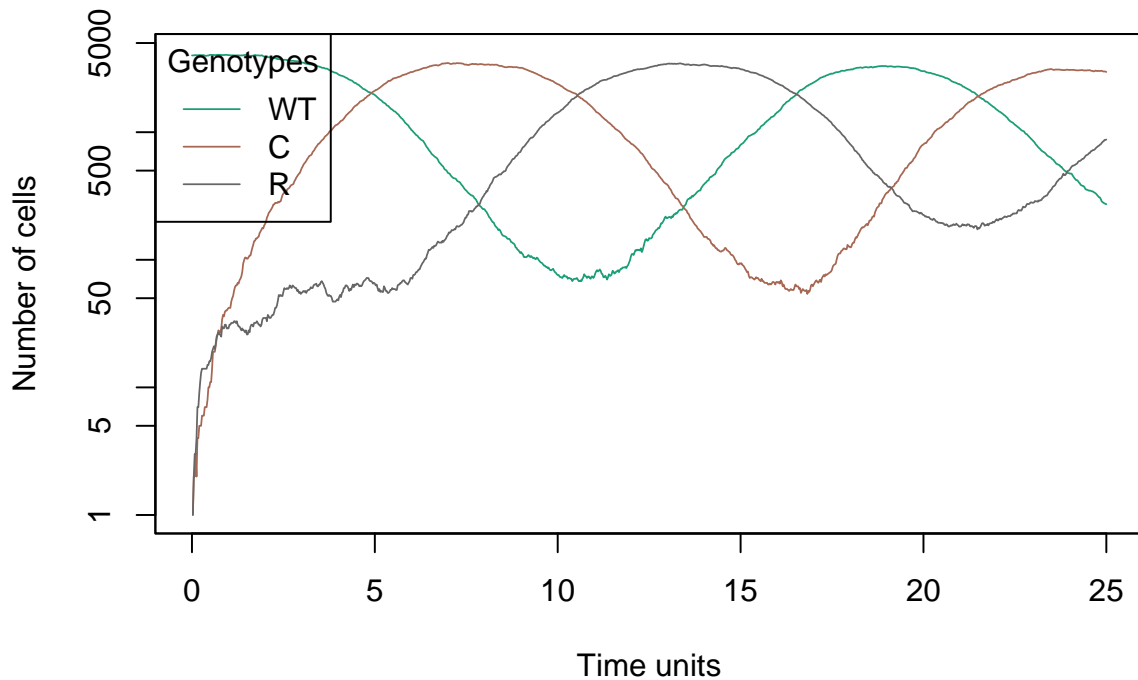
The WT genotype initially dominates the population, but, when the mutation starts, and the genotype C appears, it grows quite rapidly, since C produces the toxin (colicin) that WT is sensible to. This causes to the genotype WT to decrease, allowing the resistant strain (R) to proliferate. Once populations of the genotypes C and R grow, C strain will lose to R strain, since R resists to the toxin. Once R is dominating the total population, WT has an advantage over R, since R loses some capacities by loosing the receptor to the toxin that makes it inmune, so the WT genotype will dominate (again the population), but ¿what happens if we decide to intervene over this loop?

Let's say for example, that we do not want the R strain to proliferate, since the equations define that the R genotype needs the C genotype to exist and grow, if we cut the progression of C, R will never dominate the population. First, we define the intervention:

```
int_over_C <- list(
    list(
        ID = "Bothering R strain, by reducing C",
        Trigger = "n_C >= 500",
        WhatHappens = "n_C = n_C * 0.1",
        Periodicity = 3,
        Repetitions = Inf
    )
)


final_int_over_C <-
    createInterventions(interventions = int_over_C,
                        genotFitness = afcrs1)
## [1] "Checking intervention: Bothering R strain, by reducing C"
```

Where it is controled that, if the C genotype exceeds 500 individuals of population its population is reduced to the 1%. This will happen with a periodicity of 3 time units and with no limit of repetitions.

Running the simulations again, but this time with the interventions specified:

```
resultscrs1_noints <- oncoSimulIndiv(afcrs1,
                        model = "McFL",
                        finalTime = 25,
                        mu = 1e-2,
                        initSize = 4000,
                        onlyCancer = FALSE,
                        keepPhylog = FALSE,
                        seed = NULL,
                        interventions = final_int_over_C)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

```
plot(resultscrs1_noints, show="genotypes", type = "line")
```
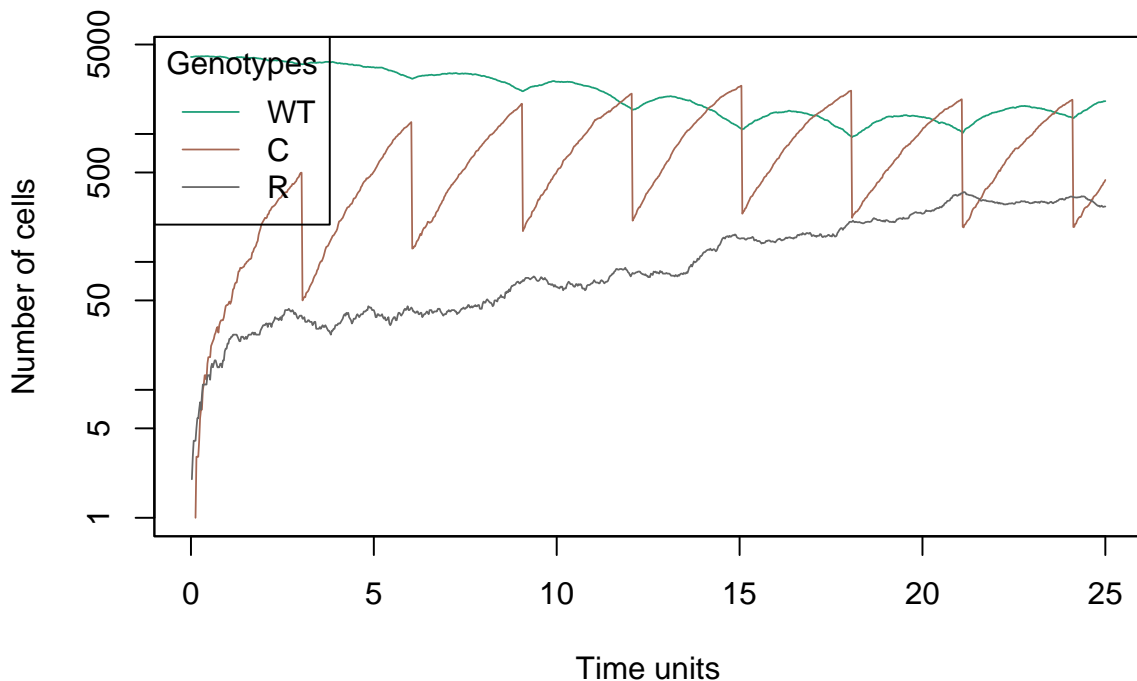

The peaks seen on the graphic representation of the population of strain C, are an effect of the intervention acting on strain C, reducing it to 10% of the initial population. As it is logical, since the WT strain continues to dominate the simulation, in the time interval that is created between intervention and intervention, the C strain invades the WT strain, in such a way that the population of the WT strain is reduced . Once the intervention happens, WT grows back, but not for long, as the growth of strain C reoccurs. On the other hand, the population of the R strain can never take off, because C cannot proliferate.

It can be seen in the graph that, from the intervention on the population of C, the population of R decreases, but, on the other hand, when the population C grows, R also does, since the strain C is not allowed to proliferating at a frequency greater than 500 never dominates the simulation.

As it can be seen, interventions are flexible, and as complex as the user wants them to be. They can also be scheduled as the user wants. so they can be executed in key parts of the simulation when certain conditions are given. In conclusion, by using interventions the user can affect how the outcome of a simulation might be.

14.5 User variables

OncoSimulR grants the user the possibilty of defining some arbitrarily complex user variables that depend on the data of the simulation. This variables will be calculated durig the simulation according to the definition provided by the user, and can be then checked as a return from the program, or can be used in the definition of interventions, allowing to emulate adaptive therapy. Thsese user variables will be defined in a list with the following parameters:

- **Name**: This will be name that will identify the variable. This attribute must be unique. in the list.
- **Value**: The value that the variable has in a given moment. When defining the variable, the user will set the initial valiue for the variable.

Therefore, the user should define a list of user variables in the following way:

```r
userVars <- list(
    list(Name         = "user_var1",
        Value       = 0
    ),
    list(Name         = "user_var2",
        Value       = 3
    ),
    list(Name         = "user_var3",
        Value       = 2.5
    )
)
```

The user can also define a list of rules that will determine when and how the user varsiables will be modified during the simulation. This is what makes the user variables be arbitrarily complex, and gives the user complete freedom when defining the variables. These rules must be defined using the following attributes: * **ID**: A parameter to identify the rule. Tis attribute must be unique. * **Condition**: The condition (must be true/false) that determines when the rule will be executed. * **Action**: The action that will take place when the rule executes, it defines what variables will be modified, and what will be their value.

Similarly to the user variables, the user can define a list of rules:

```r
rules <- list(
        list(ID = "rule_1",
            Condition = "T > 20",
            Action = "user_var_1 = 1"
        ),list(ID = "rule_2",
            Condition = "T > 30",
            Action = "user_var_2 = 2; user_var3 = 2*N"
        ),list(ID = "rule_3",
            Condition = "T > 40",
            Action = "user_var_3 = 3;user_var_2 = n_A*n_B"
```

```
        )
    )
```

As is seen in the example above, rules can depend on the current time of the simulation (T), the current total population (N) or some genotype population n_(genotype name), they can also depend on birth, death or mutation rates of genotypes (b_, d_ and m_ respectively) or even on some of the defined user variables; and can be combined as the user might want. This is possible thanks to [Exprtk library] (http://www.partow.net/programming/exprtk/), this library allow complex expressions to be specified.

In order to create the user variables and rules, the user must use `createUserVars` and `createRules` function, which will check that these are correctly specified and will adapt them so that they can be sucessfully transferred to C++. The correct way to do so is the following:

```
dfuv <- data.frame(Genotype = c("WT", "A", "B"),
                   Fitness = c("1",
                   "1 + 0.2 * (n_B > 0)",
                   ".9 + 0.4 * (n_A > 0)"
                   ))
afuv <- allFitnessEffects(genotFitness = dfuv,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = dfuv,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

Note that, for now, the fitness must have `frequencyDependentFitness` as TRUE and `frequencyType`as "abs".

After tjis, we can call both `createUserVars` and `createRules`:

```
userVars <- createUserVars(userVars)
## [1] "Checking user variable: user_var1"
## [1] "Checking user variable: user_var2"
## [1] "Checking user variable: user_var3"
```

```
rules <- createRules(rules, afuv)
## [1] "Checking rule: rule_1"
## [1] "Checking rule: rule_2"
## [1] "Checking rule: rule_3"
```

In `createUserVars` the argument is the previously defined list of user variables. In `createRules`, the first argument is the previously defined list of rules, and the second one is the `fitnessEffects` object.

Finally, once both objects are created, they can be passed as an argument to the

`oncoSimul*` function (for speed, we do not run the example below, as we have more detailed examples next).

```
uvex <- oncoSimulIndiv(
    afuv,
    model = "McFLD",
    mu = 1e-4,
    sampleEvery = 0.001,
    initSize = c(20000, 20000),
    initMutant = c("A", "B"),
    finalTime = 5.2,
    onlyCancer = FALSE,
    userVars = userVars,
    rules = rules
)
```

14.6   Basic example with user variables

The examples that will be provided in the following sections will show that the software works. In this example it will be shown how a user variable vary during the simularion time, we will use a similar example to the one used to first illustrate the intervention funtionality, for this example we will make the user variable be the proportion of genotype B cells in the total population (n_B/N), and we will check its value every second (T%1 == 0):

```
dfuv2 <- data.frame(Genotype = c("WT", "B", "A", "B, A", "C, A"),
                    Fitness = c("0*n_",
                                "1.5",
                                "1.002",
                                "1.003",
                                "1.004"))

afuv2 <- allFitnessEffects(genotFitness = dfuv2,
                           frequencyDependentFitness = TRUE,
                           frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = dfuv2,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```
userVars <- list(
    list(Name        = "genAProp",
        Value       = 0.5
    ),
    list(Name        = "genBProp",
        Value       = 0.5
    ),
```

```
        list(Name            = "genABProp",
             Value      = 0.0
        ),
        list(Name            = "genACProp",
             Value      = 0.0
        )
)

userVars <- createUserVars(userVars)
## [1] "Checking user variable: genAProp"
## [1] "Checking user variable: genBProp"
## [1] "Checking user variable: genABProp"
## [1] "Checking user variable: genACProp"
```

```
rules <- list(
        list(ID = "rule_1",
            Condition = "TRUE",
            Action = "genBProp = n_B/N"
        ),
        list(ID = "rule_2",
            Condition = "TRUE",
            Action = "genAProp = n_A/N"
        ),
        list(ID = "rule_3",
            Condition = "TRUE",
            Action = "genABProp = n_A_B/N"
        ),
        list(ID = "rule_4",
            Condition = "TRUE",
            Action = "genACProp = n_A_C/N"
        )
    )

rules <- createRules(rules, afuv2)
## [1] "Checking rule: rule_1"
## [1] "Checking rule: rule_2"
## [1] "Checking rule: rule_3"
## [1] "Checking rule: rule_4"
```

```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below
data(uvex2)

if(FALSE) {
set.seed(1)
```

```
uvex2 <- oncoSimulIndiv(
                    afuv2,
                    model = "McFLD",
                    mu = 1e-4,
                    sampleEvery = 0.01,
                    initSize = c(20000, 20000),
                    initMutant = c("A", "B"),
                    finalTime = 10,
                    onlyCancer = FALSE,
                    userVars = userVars,
                    rules = rules,
                    keepEvery = 0.1
                    )
}

plot(
    unlist(uvex2$other$userVarValues) [c(FALSE, FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex2$other$userVarValues) [c(TRUE, FALSE, FALSE, FALSE, FALSE)],
    xlab="Time", ylab="Proportion", ylim=c(0,1), type="l", col="purple")

lines(
    unlist(uvex2$other$userVarValues) [c(FALSE, FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex2$other$userVarValues) [c(FALSE, TRUE, FALSE, FALSE, FALSE)], type="

lines(
    unlist(uvex2$other$userVarValues) [c(FALSE, FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex2$other$userVarValues) [c(FALSE, FALSE, TRUE, FALSE, FALSE)], type="

lines(
    unlist(uvex2$other$userVarValues) [c(FALSE, FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex2$other$userVarValues) [c(FALSE, FALSE, FALSE, TRUE, FALSE)], type="

legend(0,1,
        legend=c("genABProp", "genACProp", "genAProp", "genBProp"), col=c("purple",
```
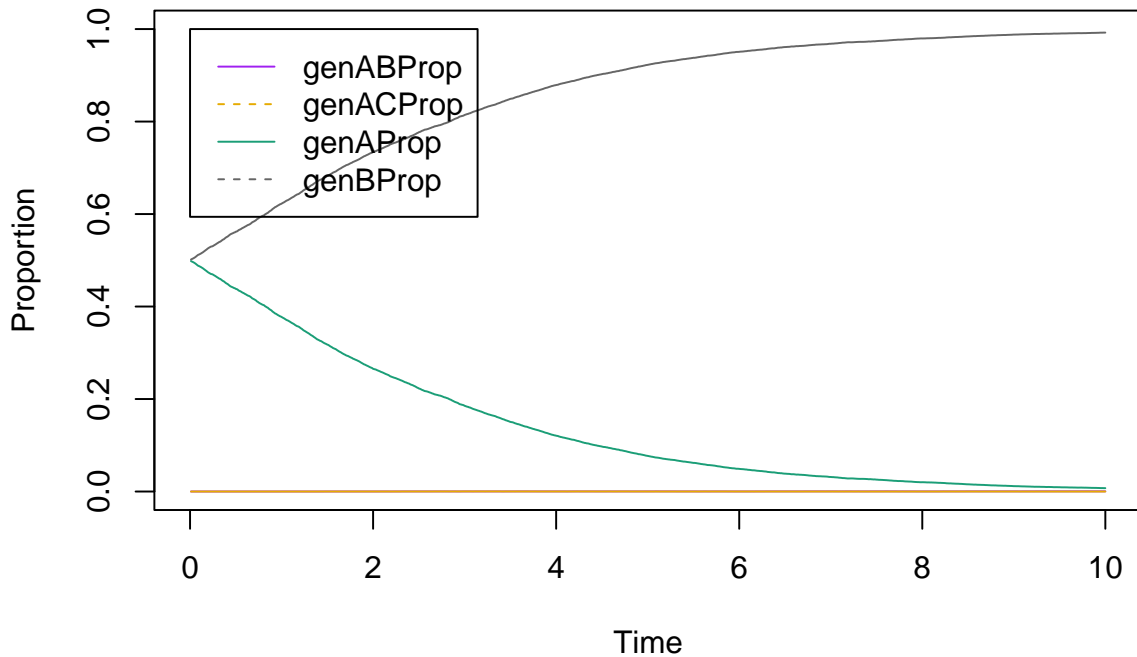
We can see, just as we saw in the interventions example that genotype B dominates the population, but, as we used user variables that show us the proportions intead of directly the populatios, we can se the fraction of the total represented by each genotype.

14.7 User Variables Example 2

In this example we will take a look at the difference between borth and death rates to better understand the evolution of the genotype populations.

```
dfuv3 <- data.frame(Genotype = c("WT", "A", "B"),
                    Fitness = c("1",
                                "1 + 0.2 * (n_B > 10)",
                                ".9 + 0.4 * (n_A > 10)"
                                ))
afuv3 <- allFitnessEffects(genotFitness = dfuv3,
                           frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = dfuv3,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std


userVars <- list(
    list(Name        = "genWTRateDiff",
        Value       = 0.5
    ),list(Name        = "genARateDiff",
        Value       = 0.5
    ),list(Name        = "genBRateDiff",
        Value       = 0.0
```

342

```
        )
)

userVars <- createUserVars(userVars)
## [1] "Checking user variable: genWTRateDiff"
## [1] "Checking user variable: genARateDiff"
## [1] "Checking user variable: genBRateDiff"
```

```
rules <- list(
        list(ID = "rule_1",
            Condition = "TRUE",
            Action = "genWTRateDiff = b_-d_"
        ),list(ID = "rule_2",
            Condition = "TRUE",
            Action = "genARateDiff = b_1-d_1"
        ),list(ID = "rule_3",
            Condition = "TRUE",
            Action = "genBRateDiff = b_2-d_2"
        )
    )

rules <- createRules(rules, afuv3)
## [1] "Checking rule: rule_1"
## [1] "Checking rule: rule_2"
## [1] "Checking rule: rule_3"
```

```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(uvex3)

if(FALSE) {
set.seed(1)

uvex3 <- oncoSimulIndiv(afuv3,
                    model = "McFLD",
                    onlyCancer = FALSE,
                    finalTime = 105,
                    mu = 1e-4,
                    initSize = 5000,
                    keepPhylog = FALSE,
                    seed = NULL,
                    errorHitMaxTries = FALSE,
                    errorHitWallTime = FALSE,
```

```
                        userVars = userVars,
                        rules = rules,
                        keepEvery = 1)
}

plot(uvex3, show = "genotypes", type = "line")
```
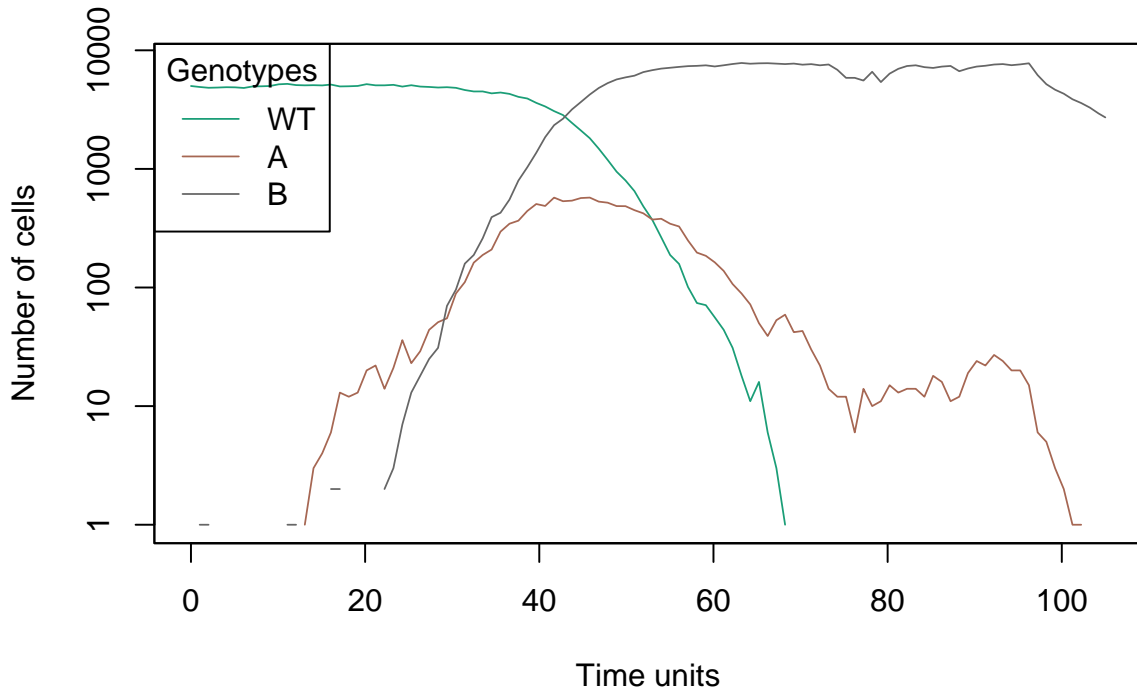

We now plot the difference between birth and death rate

```
plot(
    unlist(uvex3$other$userVarValues) [c(FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex3$other$userVarValues) [c(FALSE, FALSE, TRUE, FALSE)], xlab="Time",
    xlim=c(0, 105),
    ylim=c(-0.75,0.75), type="l", col="#1B9E77")

lines(
    unlist(uvex3$other$userVarValues) [c(FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex3$other$userVarValues) [c(TRUE, FALSE, FALSE, FALSE)], type="l", col

lines(
    unlist(uvex3$other$userVarValues) [c(FALSE, FALSE, FALSE, TRUE)],
    unlist(uvex3$other$userVarValues) [c(FALSE, TRUE, FALSE, FALSE)], type="l", col

legend(0, 0.75,
        legend=c("genWTRateDiff", "genARateDiff", "genBRateDiff"), col=c("#1B9E77",
```
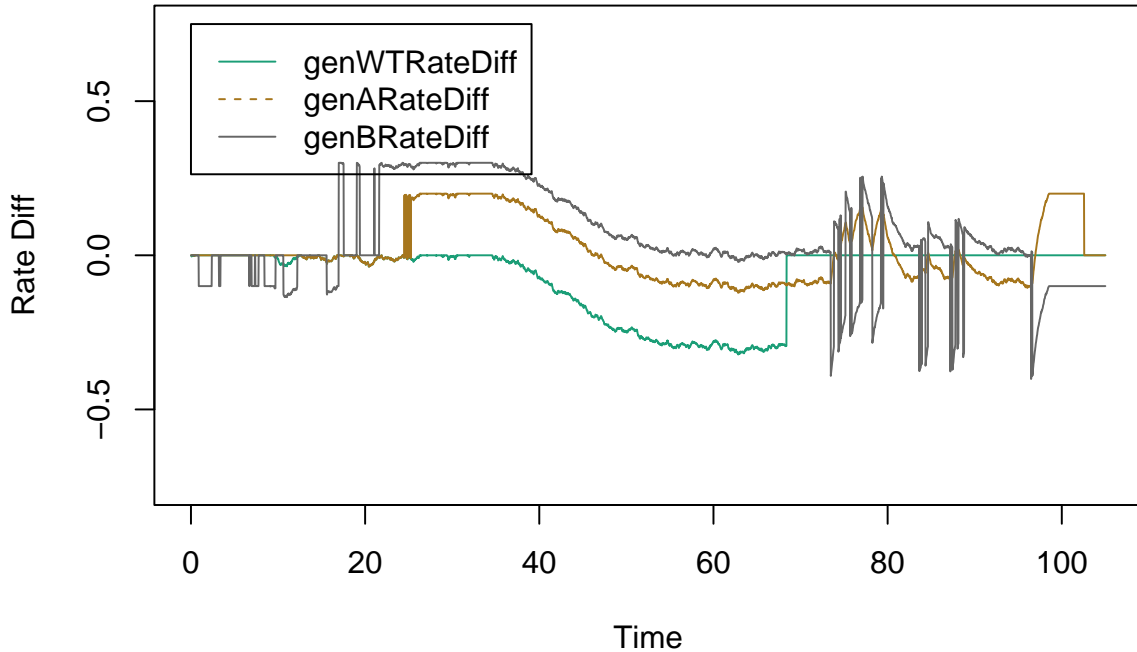
As expected we see that the points in time when the population of some genotypes decline match the points where the difference between birth and death rate is negative.

14.8 Adaptive therapy. Interventions using user variables

OncoSimulR can now emilate the effects of adaptive therapy (Hansen & Read (2020b); Hansen & Read (2020a)) during the simulations by using the implemented user variables when defining interventions. When doing so, we can create a set of interventions that vary according to the current state of the tumor in the simulation, as the user variables are constantly changing according to the simulaition parameters.

In order to achieve this, we must first define the user variables and rules to calculate their value, as explained in the user variables section:

```
dfat <- data.frame(Genotype = c("WT", "B", "A", "B, A", "C, A"),
                   Fitness = c("0*n_",
                               "1.5",
                               "1.002",
                               "1.003",
                               "1.004"))


adat <- allFitnessEffects(genotFitness = dfat,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = dfat,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

345

```
userVars <- list(
    list(Name          = "user_var1",
        Value       = 0
    )
)

userVars <- createUserVars(userVars)
## [1] "Checking user variable: user_var1"
```

```
rules <- list(
        list(ID = "rule_1",
            Condition = "n_B > n_A",
            Action = "user_var1 = n_A-n_B"
        ),list(ID = "rule_2",
            Condition = "n_B > n_A",
            Action = "user_var1 = n_B-n_A"
        )
    )

rules <- createRules(rules, adat)
## [1] "Checking rule: rule_1"
## [1] "Checking rule: rule_2"
```

After this, we proceed to define the interventions as explained in the interventions section. When doing so, we can now use thedefined user variables in order to achieve the adaptive therapy.

```
interventions <- list(
    list(ID           = "i1",
        Trigger       = "N > 1000",
        WhatHappens   = "N = user_var1*0.8",
        Periodicity   = 1,
        Repetitions   = 5
    )
)

interventions <- createInterventions(interventions, adat)
## [1] "Checking intervention: i1"
```

We now proceed to the call to OncoSimulIndiv passing userVars, rules and interventions as arguments and we plot the result populations.

```
atex <- oncoSimulIndiv(
                adat,
                model = "McFLD",
                mu = 1e-4,
                sampleEvery = 0.01,
```
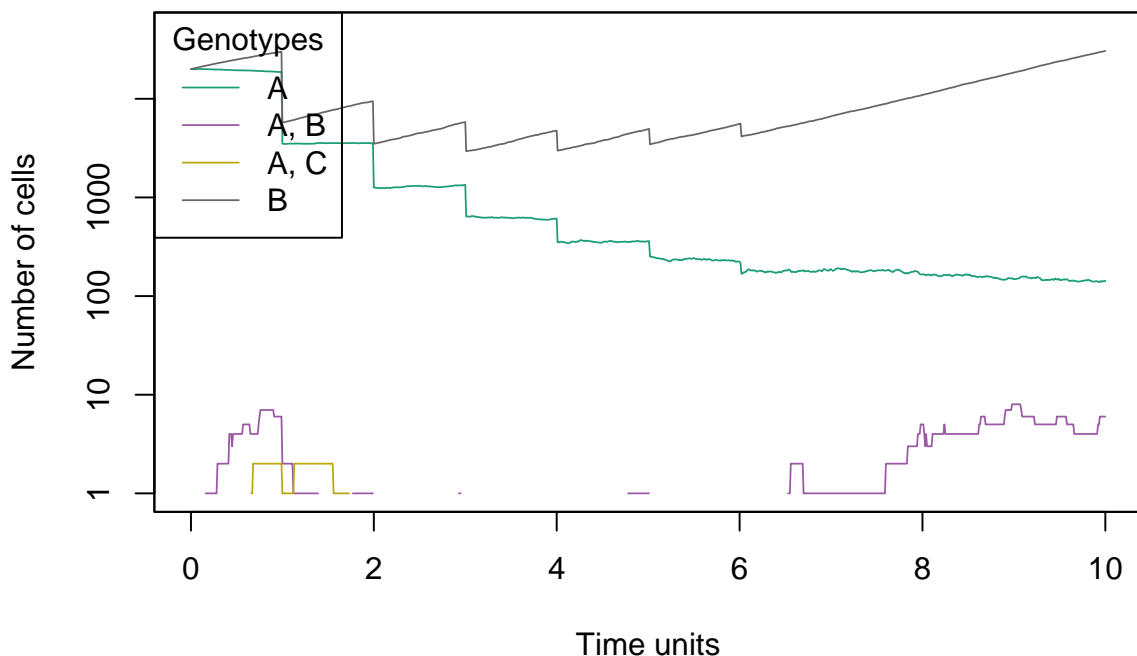
```
                         initSize = c(20000, 20000),
                         initMutant = c("A", "B"),
                         finalTime = 10,
                         onlyCancer = FALSE,
                         userVars = userVars,
                         rules = rules,
                         interventions = interventions
                         )
```
```
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

```
plot(atex, show = "genotypes", type = "line")
```


14.9   Another example of adaptive therapy

Finally we will try and examplify a more real scenario where we have a continuous specific treatment to keep the tumor at bay, if the mutated cells are enough to be detected, but if we detect a sudden increase in some mutated genotype we switch the treatment to focus that growth, also, if the tumor grows too big, we simulate a quirurgic intervention.

```
dfat3 <- data.frame(Genotype = c("WT", "A", "B"),
                    Fitness = c("1",
                                "0.8 + 0.2 * (n_B > 10) + 0.1 (n_A > 10)",
                                "0.8 + 0.25 * (n_B > 10)"
                                ))
afat3 <- allFitnessEffects(genotFitness = dfat3,
```

347

```
                              frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = dfat3,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std


userVars <- list(
    list(Name         = "lastMeasuredA",
        Value       = 0
    ),
    list(Name         = "lastMeasuredB",
        Value       = 0
    ),
    list(Name         = "previousA",
        Value       = 0
    ),
    list(Name         = "previousB",
        Value       = 0
    ),
    list(Name         = "lastTime",
        Value       = 0
    ),
    list(Name         = "measure",
        Value       = 0
    ),
    list(Name         = "treatment",
        Value       = 0
    )
)

userVars <- createUserVars(userVars)
## [1] "Checking user variable: lastMeasuredA"
## [1] "Checking user variable: lastMeasuredB"
## [1] "Checking user variable: previousA"
## [1] "Checking user variable: previousB"
## [1] "Checking user variable: lastTime"
## [1] "Checking user variable: measure"
## [1] "Checking user variable: treatment"

rules <- list(
      list(ID = "rule_1",
          Condition = "T - lastTime < 10",
          Action = "measure = 0"
```

```r
        ),
        list(ID = "rule_2",
            Condition = "T - lastTime >= 10",
            Action = "measure = 1;lastTime = T"
        ),
        list(ID = "rule_3",
            Condition = "measure == 1",
            Action = "previousA = lastMeasuredA;previousB = lastMeasuredB;lastMeasu
        ),
        list(ID = "rule_4",
            Condition = "TRUE",
            Action = "treatment = 0"
        ),
        list(ID = "rule_5",
            Condition = "lastMeasuredA + lastMeasuredB > 100",
            Action = "treatment = 1"
        ),
        list(ID = "rule_6",
            Condition = "lastMeasuredA - PreviousA > 500",
            Action = "treatment = 2"
        ),
        list(ID = "rule_7",
            Condition = "lastMeasuredB - PreviousB > 500",
            Action = "treatment = 3"
        ),
        list(ID = "rule_8",
            Condition = "lastMeasuredA - PreviousA > 500 and lastMeasuredB - Previo
            Action = "treatment = 4"
        )
    )

rules <- createRules(rules, afat3)
## [1] "Checking rule: rule_1"
## [1] "Checking rule: rule_2"
## [1] "Checking rule: rule_3"
## [1] "Checking rule: rule_4"
## [1] "Checking rule: rule_5"
## [1] "Checking rule: rule_6"
## [1] "Checking rule: rule_7"
## [1] "Checking rule: rule_8"


interventions <- list(
    list(ID          = "basicTreatment",
        Trigger      = "treatment == 1",
```

```
          WhatHappens   = "N = 0.8*N",
          Periodicity   = 10,
          Repetitions   = Inf
     ),
     list(ID           = "treatmentOverA",
          Trigger      = "treatment == 2 or treatment == 4",
          WhatHappens  = "n_B = n_B*0.3",
          Periodicity  = 20,
          Repetitions  = Inf
     ),
     list(ID           = "treatmentOverB",
          Trigger      = "treatment == 3 or treatment == 4",
          WhatHappens  = "n_B = n_B*0.3",
          Periodicity  = 20,
          Repetitions  = Inf
     ),
     list(ID           = "intervention",
          Trigger      = "lastMeasuredA+lastMeasuredB > 5000",
          WhatHappens  = "N = 0.1*N",
          Periodicity  = 70,
          Repetitions  = Inf
     )
)

interventions <- createInterventions(interventions, afat3)
## [1] "Checking intervention: basicTreatment"
## [1] "Checking intervention: treatmentOverA"
## [1] "Checking intervention: treatmentOverB"
## [1] "Checking intervention: intervention"
```

```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(atex2b)

if (FALSE) {

set.seed(1) ## for reproducibility
atex2b <- oncoSimulIndiv(afat3,
                      model = "McFLD",
                      onlyCancer = FALSE,
                      finalTime = 200,
                      mu = 1e-4,
                      initSize = 5000,
```

```
                    keepPhylog = FALSE,
                    seed = NULL,
                    errorHitMaxTries = FALSE,
                    errorHitWallTime = FALSE,
                    userVars = userVars,
                    rules = rules,
                    interventions = interventions,
                    keepEvery = 1)
}

plot(atex2b, show = "genotypes", type = "line")
```
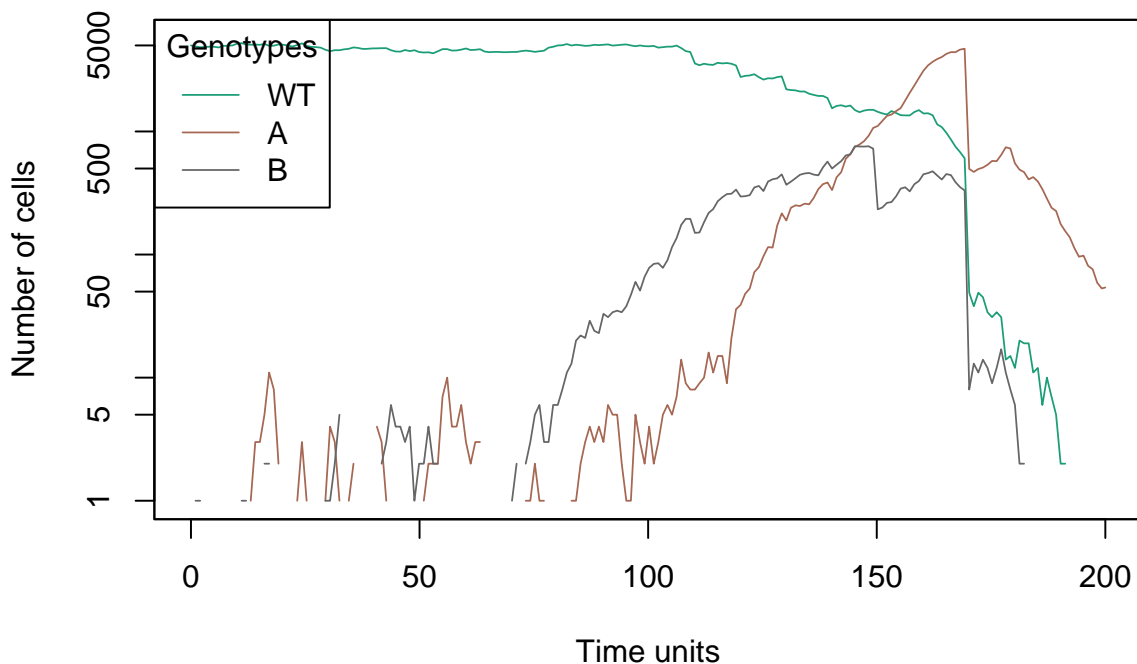

We clearley see here how the therapy adapts to the circumstances cutting genotype B when it starts to grow rapidly, and then making a major intervention once A grows so big that the total mutated cell exceed the set amount.

14.10   Adaptive therapy: a canonical example

We now simulate a canonical adaptive therapy example such as the one shown in examples b and c of Figure 1 in Hansen & Read (2020b).

We start by creating the scenario with 2 types of cells, A will be treatment-resistant and slow growing whereas B will be faster growing but treatment susceptible.

```
dfat4 <- data.frame(Genotype = c("WT", "A", "B"),
                Fitness = c("n_/n_",
                            "1.005",
                            "1.1"
                            ))
afat4 <- allFitnessEffects(genotFitness = dfat4,
```

```
                        frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = dfat4,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
```

We first execute this scenario without adaptive therapy to check what the outcome would be. We therefore set the standard treatment by defining the following intervention:

```
interventions <- list(
    list(ID          = "i1",
        Trigger      = "T > 10",
        WhatHappens  = "n_B = n_B*0.8",
        Periodicity  = 1,
        Repetitions  = Inf
    )
)

interventions <- createInterventions(interventions, afat4)
## [1] "Checking intervention: i1"
```

We run the simulation and plot the results.

```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(atex4)

if(FALSE) {
set.seed(1) ## for reproducibility
atex4 <- oncoSimulIndiv(afat4,
                        model = "McFLD",
                        onlyCancer = FALSE,
                        finalTime = 2000,
                        mu = 1e-4,
                        initSize = c(10000, 50, 1000),
                        initMutant = c("WT", "A", "B"),
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE,
                        interventions = interventions,
                        keepEvery = 1)
}
```
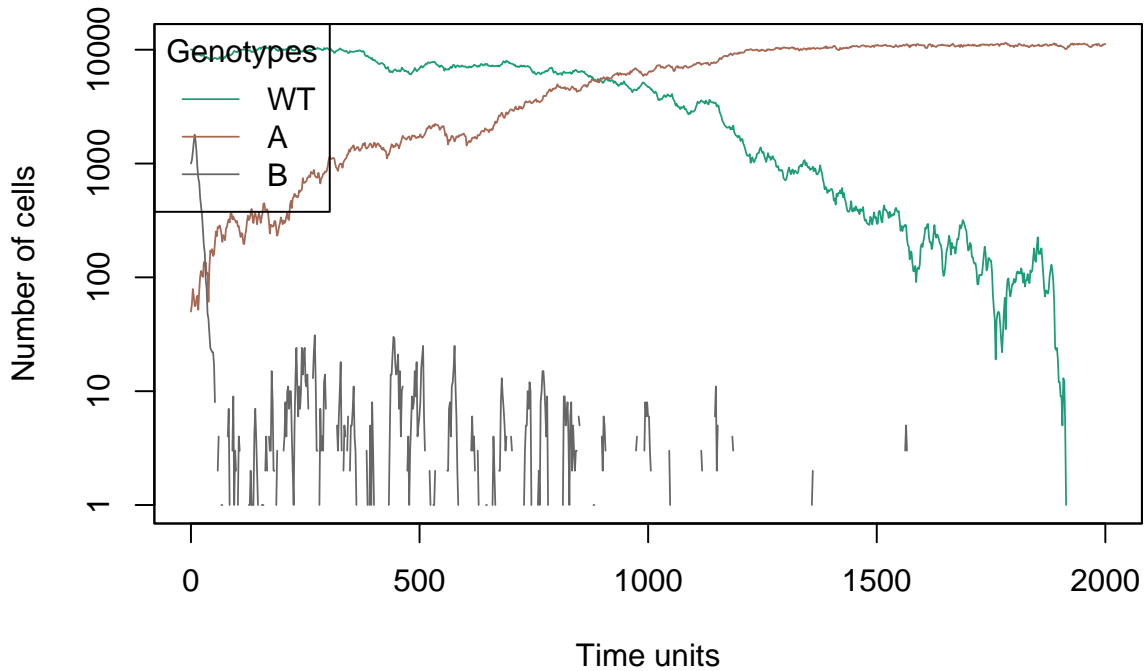
```r
plot(atex4, show = "genotypes", type = "line")
```


We now set the same initial scenario but this time we apply adaptive therapy.

```r
dfat5 <- data.frame(Genotype = c("WT", "A", "B"),
                    Fitness = c("n_/n_",
                                "1.005",
                                "1.1"
                                ))
afat5 <- allFitnessEffects(genotFitness = dfat5,
                           frequencyDependentFitness = TRUE)
## Warning in allFitnessEffects(genotFitness = dfat5,
## frequencyDependentFitness = TRUE): v2 functionality detected.
## Adapting to v3 functionality.
## frequencyType set to 'auto'
## All single-gene genotypes as input to to_genotFitness_std
```

In order to apply adaptive therapy we set the following variables and rules

```r
userVars <- list(
    list(Name           = "measure",
         Value       = 0
    ),list(Name           = "lastTime",
         Value       = 0
    ),list(Name           = "treatment",
         Value       = 0
    ),list(Name           = "totalPopMeasured",
         Value       = 0
    )
```

```
)

userVars <- createUserVars(userVars)
## [1] "Checking user variable: measure"
## [1] "Checking user variable: lastTime"
## [1] "Checking user variable: treatment"
## [1] "Checking user variable: totalPopMeasured"
```

```
rules <- list(
        list(ID = "rule_1",
            Condition = "T - lastTime < 10",
            Action = "measure = 0"
        ),list(ID = "rule_2",
            Condition = "T - lastTime >= 10",
            Action = "measure = 1;lastTime = T"
        ),list(ID = "rule_3",
            Condition = "measure == 1",
            Action = "totalPopMeasured = n_A + n_B"
        ),list(ID = "rule_4",
            Condition = "totalPopMeasured < 2000",
            Action = "treatment = 0"
        ),list(ID = "rule_5",
            Condition = "totalPopMeasured >= 2000",
            Action = "treatment = 1"
        )
    )

rules <- createRules(rules, afat5)
## [1] "Checking rule: rule_1"
## [1] "Checking rule: rule_2"
## [1] "Checking rule: rule_3"
## [1] "Checking rule: rule_4"
## [1] "Checking rule: rule_5"
```

We define the same intervention as before but we apply it only when the adaptive therapy determines so.

```
interventions <- list(
    list(ID          = "i1",
        Trigger      = "treatment == 1",
        WhatHappens  = "n_B = n_B*0.8",
        Periodicity  = 1,
        Repetitions  = Inf
    )
)
```

```
interventions <- createInterventions(interventions, afat5)
## [1] "Checking intervention: i1"
```

We run the simulation and plot the results.

```
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(atex5)

if(FALSE) {
set.seed(1) ## for reproducibility
atex5 <- oncoSimulIndiv(afat5,
                        model = "McFLD",
                        onlyCancer = FALSE,
                        finalTime = 1500,
                        mu = 1e-4,
                        initSize = c(10000, 50, 1000),
                        initMutant = c("WT", "A", "B"),
                        keepPhylog = FALSE,
                        seed = NULL,
                        errorHitMaxTries = FALSE,
                        errorHitWallTime = FALSE,
                        userVars = userVars,
                        rules = rules,
                        interventions = interventions)
}

plot(atex5, show = "genotypes", type = "line")
```
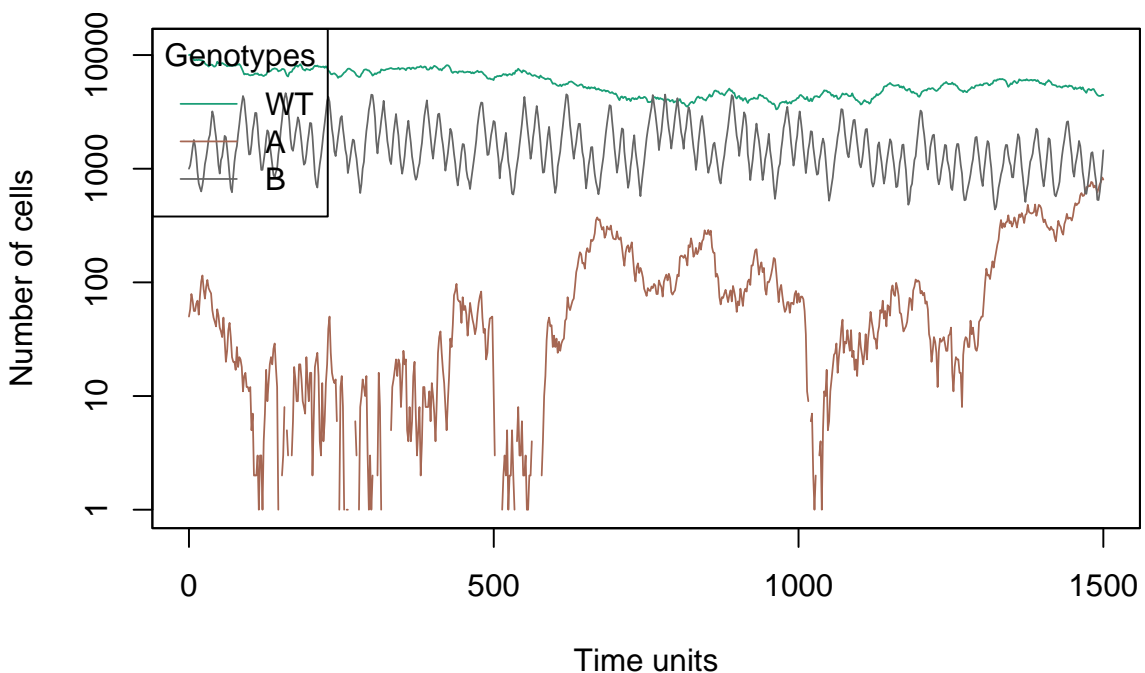
Comparing these results with the ones without adaptive therapy we see that by interrupting the treatment before the complete elimination of the susceptible treatment we achieve the desired effect, that is the control of the resistant genotype by maintaining certain degree of cellular competition between genotypes.

15 Simulating therapeutic interventions that depend on time

In this section we show some examples using the time dependent functionality; with it, fitness can be made to depend on T, the current time defined in the simulation. These examples were originally prepared by Niklas Endres, Rafael Barrero Rodríguez, Rosalía Palomino Cabrera and Silvia Talavera Marcos, as an excercisse for the course Programming and Statistics with R (Master's Degree in Bioinformatics and Computational Biology, Universidad Autónoma de Madrid), course 2019-20; Niklas Endres had the idea of accessing T from exprTk.

This first example is an artificial simulation, but it shows how the fitness of a genotype can suddenly increase at a certain given timepoint.

```r
## Fitness definition
fl <- data.frame(
  Genotype = c("WT", "A", "B"),
  Fitness = c("1",                          #WT
              "if (T>50) 1.5; else 0;",    #A
              "0*f_") ,                      #B
  stringsAsFactors = FALSE
)


fe <- allFitnessEffects(genotFitness = fl,
                        frequencyDependentFitness = TRUE,
                        frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = fl,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```r
## Evaluate the fitness before and after the specified currentTime
evalAllGenotypes(fe, spPopSizes = c(100, 100, 100))
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
##    Genotype Fitness
## 1       WT        1
## 2        A        0
## 3        B        0
## 4     A, B        0
```

```r
evalAllGenotypes(fe, spPopSizes = c(100, 100, 100), currentTime = 80)
## Using old version of fitnessEffects. Transforming fitnessEffects
```

```
##              to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
##    Genotype Fitness
## 1        WT     1.0
## 2         A     1.5
## 3         B     0.0
## 4      A, B     0.0
```

```r
## Simulation
sim <- oncoSimulIndiv(fe,
                      model = "McFL",
                      onlyCancer = FALSE,
                      finalTime = 100,
                      mu = 0.01,
                      initSize = 5000,
                      keepPhylog = FALSE,
                      seed = NULL,
                      errorHitMaxTries = FALSE,
                      errorHitWallTime = FALSE)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

```r
## Plot the results
plot(sim, show = "genotypes")
```
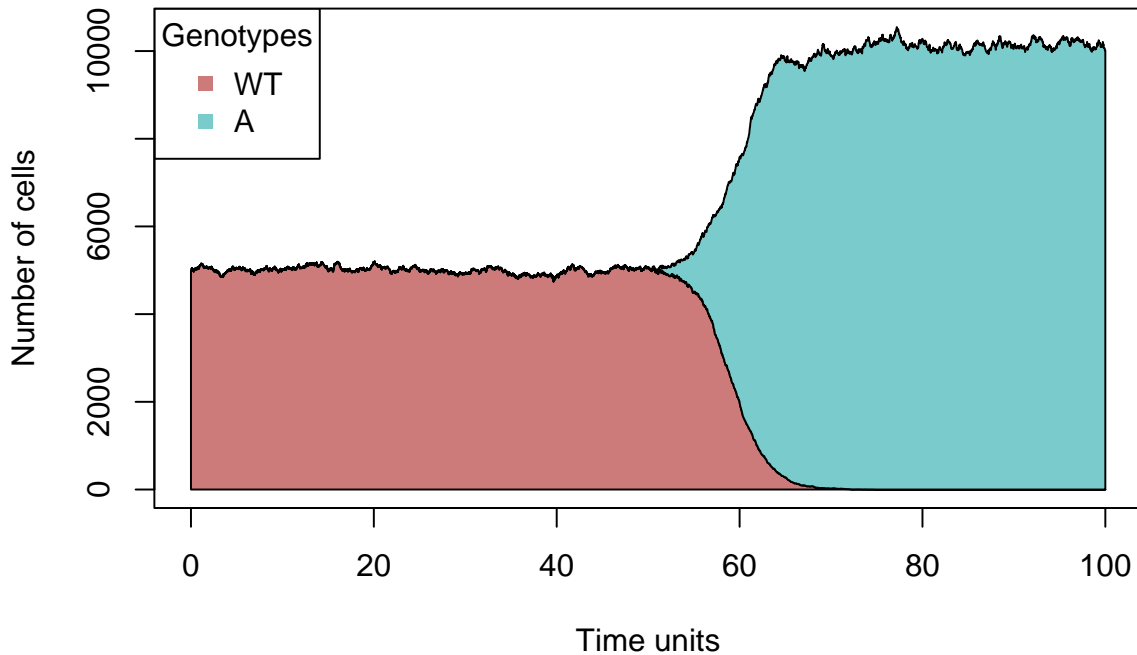
15.1 Adaptive control of competitive release and chemotherapeutic resistance

The code structure of the previous example can be used, for instance, to simulate the effect of different chemotherapy treatment protocols.

An example of using these game theory concepts is the adaptive theory. The primary goal is to maximize the time of tumor control by using the tumor cells that are sensitive to treatment as agents that can supress the proliferation of the resistant cells. Thus, a significant residual populations of tumor cells can be under control to inhibit the growth of cells that otherwise cannot control: killed vs resistant.

Newton and Ma Newton & Ma (2019) built simulations for a tumor consisting in two types of cells: resistant to chemotherapy and sensitive to chemotherapy. The idea is to promote competition among tumor cells in order to prevent tumor growth.

To do this, they developed a **three-component Prisoner's Dilemma scenario** including healthy (**H**) cells as well as chemoresistant (**R**) and chemosensitive (**S**) cancer cells. Healthy cells are cooperators, while cancer cells are defectors.

This is the system's payoff matrix:

```
a <- 1;     b <- 0.5;  c <- 0.5    ## a b c
d <- 1;     e <- 1.25; f <- 0.7    ## d e f
g <- 0.975; h <- -0.5; i <- 0.75   ## g h i


payoff_m <- matrix(c(a,b,c,d,e,f,g,h,i), ncol=3, byrow=TRUE)
colnames(payoff_m) <- c("Healthy", "Chemo-sensitive", "Chemo-resistant")
rownames(payoff_m) <- c("Healthy", "Chemo-sensitive", "Chemo-resistant")
print(payoff_m <- as.table(payoff_m))
##                Healthy Chemo-sensitive Chemo-resistant
```

```
## Healthy          1.000              0.500              0.500
## Chemo-sensitive   1.000              1.250              0.700
## Chemo-resistant   0.975             -0.500              0.750
```

The numerical values in the matrix are selected to satisfy the following theoretical constraints:

g > a > i > c; d > a > e > b; f > i > e > h and d > g (cost to resistance).

Thus, the fitness definitions for the three types of cells could be written as in the following data frame:

```r
print( df <- data.frame(
  CellType = c("H", "S", "R"),
  Fitness = c("F(H) = ax(H) + bx(S) + cx(R)",     #Healthy
              "F(S) = dx(H) + ex(S) + fx(R)",     #Sensitive
              "F(R) = gx(H) + hx(S) + ix(R)")),   #Resistant
  row.names = FALSE )
##  CellType                        Fitness
##         H F(H) = ax(H) + bx(S) + cx(R)
##         S F(S) = dx(H) + ex(S) + fx(R)
##         R F(R) = gx(H) + hx(S) + ix(R)
```

In summary:

- The fitness of H cells is lower than the chemosensitive cells.
- Without any therapy, S cells have a higher fitness value than R cells, because of the cost to resistance
- In the presence of therapy, R cells take over the S cells

One of the challenges is to optimize the drug dosage intervals. In practice, this would mean to infer the growth rates of the different cell types from a frequent monitoring of the tumor environment. With OncoSimulR is possible to try out different intervals on simulations of the collected data.

15.1.1 Scenario without chemotherapy

First of all, we can simulate the growth of a tumor from H cells without any treatment. We can consider that R tumor cells are generated from S and WT cells. Thus, as we expected, we can observe in the simulation results that S cells grow (tumor) and R cells cannot subsist because of their fitness disadvantage: the cost of being resistant.

```r
set.seed(2)
RNGkind("L'Ecuyer-CMRG")


## Coefficients
# Healthy  Sensitive  Resistant
a=3;       b=1.5;     c=1.5      # Healthy
d=4;       e=5;       f=2.8      # Sensitive
```

```r
g=3.9;        h=-2;        i=2.2        # Resistant



# Here we divide coefficients to reduce the amount of cells obtained in the simul
# We have divided a, b and c by 3, and d, e and i by 4.

# Healthy      Sensitive    Resistant
a <- 1;        b <- 0.5;    c <- 0.5      # Healthy
d <- 1;        e <- 1.25;   f <- 0.7      # Sensitive
g <- 0.975;    h <- -0.5;   i <- 0.75     # Resistant


## Fitness definition
players <- data.frame(Genotype = c("WT","S","R","S,R"),
                      Fitness = c(paste0(a, "*f_+", b, "*f_S+", c, "*f_S_R"), #WT
                                  paste0(d,"*f_+",e,"*f_S+",f,"*f_S_R"),        #S
                                  "0",                                          #R
                                  paste0(g,"*f_+",h,"*f_S+",i,"*f_S_R")),      #S,R
                      stringsAsFactors = FALSE)


game <- allFitnessEffects(genotFitness = players,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = players,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```
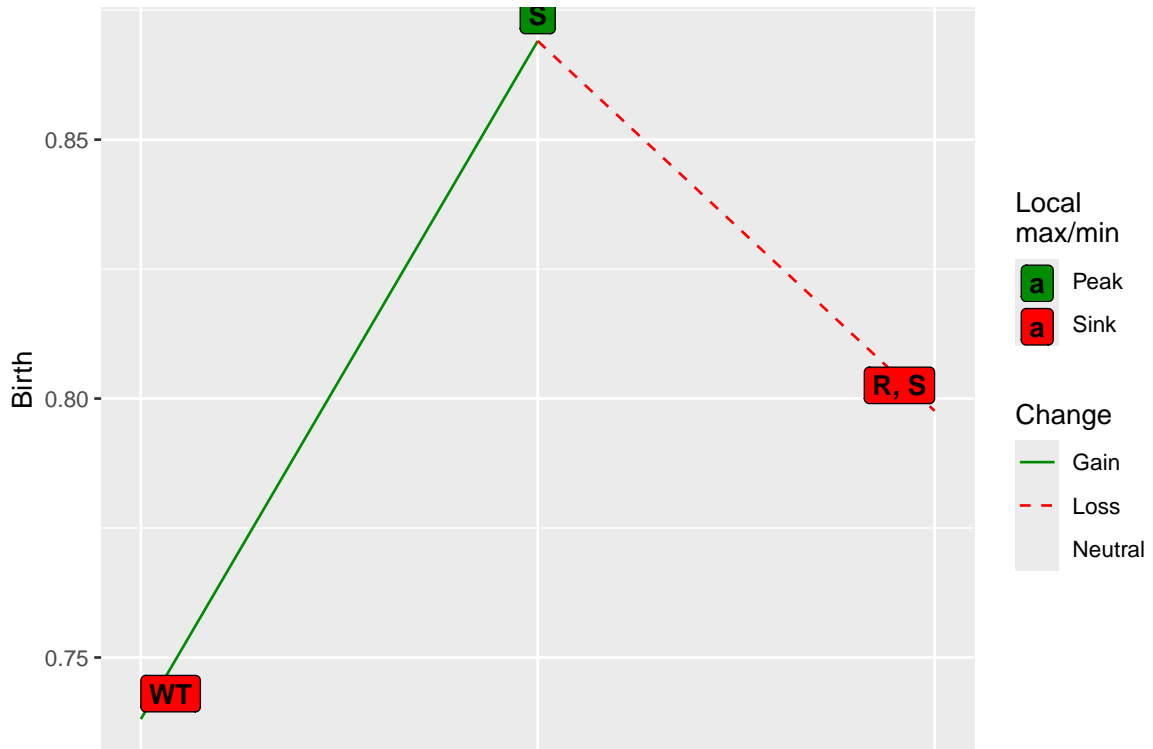
```r
## Plot the first scenario
eag <- evalAllGenotypes(game, spPopSizes = c(10,1,0,10))[c(1, 3, 4),]
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
```

```r
plot(eag)
```

```
## Simulation
gamesimul <- oncoSimulIndiv(game,
                            model = "McFL",
                            onlyCancer = FALSE,
                            finalTime = 40,
                            mu = 0.01,
                            initSize = 5000,
                            keepPhylog = FALSE,
                            seed = NULL)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```
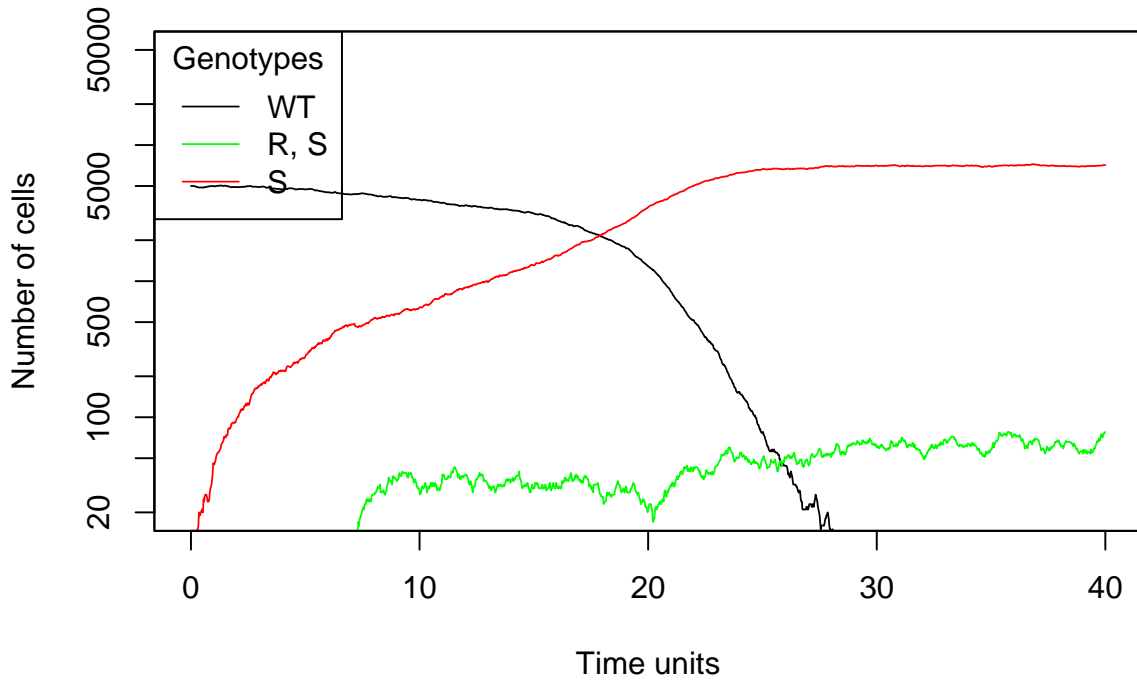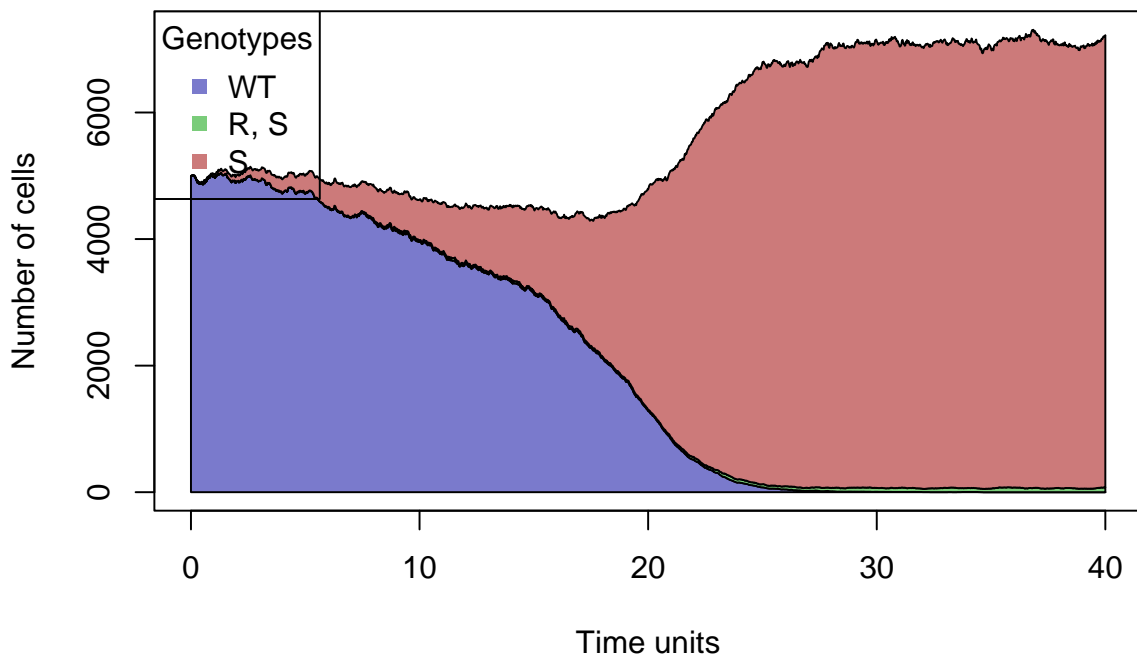
```
## Plot 2
plot(gamesimul, show = "genotypes", type = "line",
     col = c("black", "green", "red"), ylim = c(20, 50000))
```

```r
plot(gamesimul, show = "genotypes")
```


15.1.2 Scenario with continuous chemotherapy: fixed dose

For this simulation, we add the effect of chemotherapy as a fixed coefficient (**drug_eff**), representing a fixed dose. The dose is delivered only when a tumor has grown up, so we perform the drug effect after starting the simulation. For this, we apply the time funcionality used in the first section of this chapter.

```r
# Effect of drug on fitness sensible tumor cells
drug_eff <- 0.01
```

```r
wt_fitness <- paste0(a, "*f_+", b, "*f_S+", c, "*f_S_R")
sens_fitness <- paste0(d, "*f_+", e, "*f_S+", f, "*f_S_R")
res_fitness <- paste0(g, "*f_+", h, "*f_S+", i, "*f_S_R")

players_1 <- data.frame(Genotype = c("WT", "S", "R", "S, R"),
                        Fitness = c(wt_fitness,
                                    paste0("if (T>50) ", drug_eff, "*(",sens_fitnes
                                        else ", sens_fitness, ";"),
                                    "0",
                                    res_fitness),
                        stringsAsFactors = FALSE)

period_1 <- allFitnessEffects(genotFitness = players_1,
                              frequencyDependentFitness = TRUE,
                              frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = players_1,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```r
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(simul_period_1)

if (FALSE) {
set.seed(2)

final_time <- 170 ## for speed
simul_period_1 <- oncoSimulIndiv(period_1,
                                 model = "McFL",
                                 onlyCancer = FALSE,
                                 finalTime = final_time,
                                 mu = 0.01,
                                 initSize = 5000,
                                 keepPhylog = FALSE,
                                 seed = NULL)
}
# ylim has been adapted to number of cells
plot(simul_period_1, show = "genotypes", type = "line",
     col = c("black", "green", "red"), ylim = c(20, 300000),
     thinData = TRUE)
```
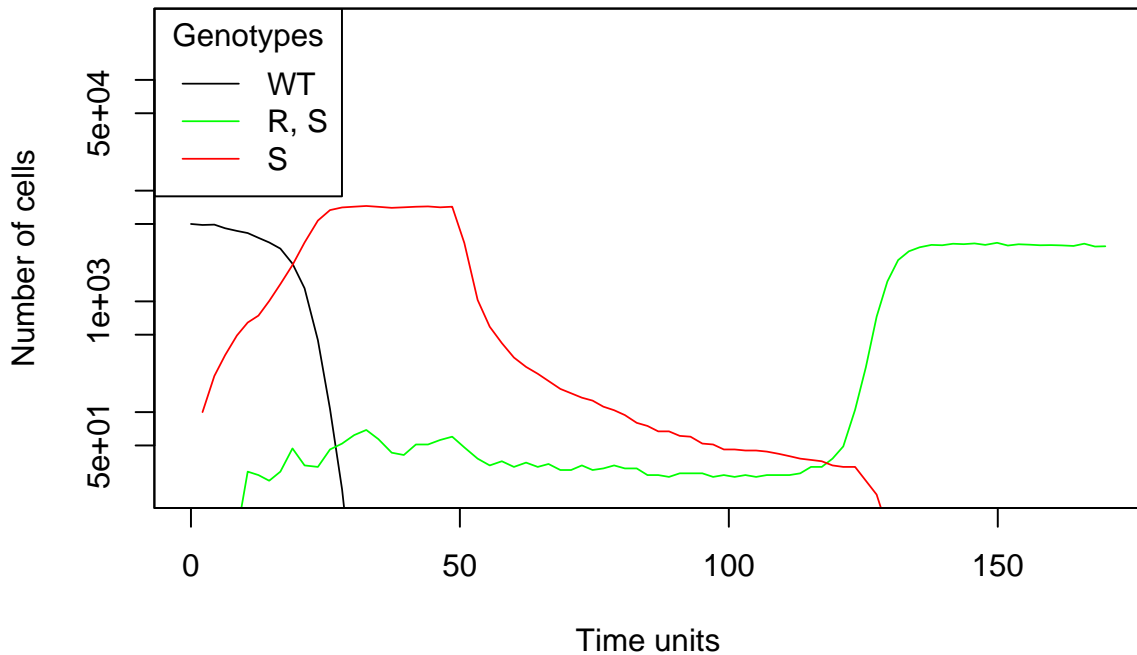
```
## plot(simul_period_1, show = "genotypes", ylim = c(20, 12000))
```

As expected, the simulation results show how sensitive cells suddenly decrease when the current time of the simulation reach a value of 50, which is the consequence of the chemotherapy. In this regard, the code illustrates how sensitive cells fitness is multiplied by *drug_eff* variable after 50 units of time, and then resistant cells start to grow exponentially until reaching an equilibrium where chemoterapy does not affect them anymore.

With this example, we can show how chemotherapy usage could be counterproductive under certain situations, especially in those cases in which resistant tumor cells are more aggressive than sensitive cells.

15.1.3   Scenario with switching doses of chemotherapy

The original model by Newton and Ma Newton & Ma (2019) includes chemotherapeutic dosage as a time-dependent controller. The main idea is to increase or decrease the dose according to a periodic cancer growth in order to avoid the fixation of both sensitive and resistant cells, and keep the tumor trajectory enclosed in a loop.

The model developed by this group is a is a cubic nonlinear system based on Hamiltonian orbits. They use time-dependent chemotherapeutic parameters *w*, whose values are different in each one of the several and carefully chosen intervals that depends on *C(t)*, which is the chemo-concentration parameter. For simplicity's sake, we will just define the fitness of sensitive cells as dependent on a sine time function.

During this simulation, we will see in our results small oscilations doses that keep sensitive cells population at a minimum value and the R cells progress is prevented.

```r
set.seed(2)
RNGkind("L'Ecuyer-CMRG")

# Healthy      Sensitive    Resistant
a <- 1;        b <- 0.5;    c <- 0.5     # Healthy
d <- 1;        e <- 1.25;   f <- 0.7     # Sensitive
g <- 0.975;    h <- -0.5;   i <- 0.75    # Resistant

wt_fitness   <- paste0(a, "*f_+", b, "*f_S+", c, "*f_S_R")
sens_fitness <- paste0(d, "*f_+", e, "*f_S+", f, "*f_S_R")
res_fitness  <- paste0(g, "*f_+", h, "*f_S+", i, "*f_S_R")

fitness_df <-data.frame(Genotype = c("WT", "S", "R", "S, R"),
                        Fitness = c(wt_fitness,
                                    paste0("if (T>50) (sin(T+2)/10) * (", sens_fitn
                                           "; else ", sens_fitness, ";"),
                                    "0",
                                    res_fitness),
                        stringsAsFactors = FALSE)

afe <- allFitnessEffects(genotFitness = fitness_df,
                         frequencyDependentFitness = TRUE,
                         frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = fitness_df,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```r
switching_sim  <- oncoSimulIndiv(afe,
                                 model = "McFL",
                                 onlyCancer = FALSE,
                                 finalTime = 100,
                                 mu = 0.01,
                                 initSize = 5000,
                                 keepPhylog = FALSE,
                                 seed = NULL)
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
```

```r
plot(switching_sim, show = "genotypes", type = "line",
    col = c("black", "green", "red"), ylim = c(20, 200000))
```
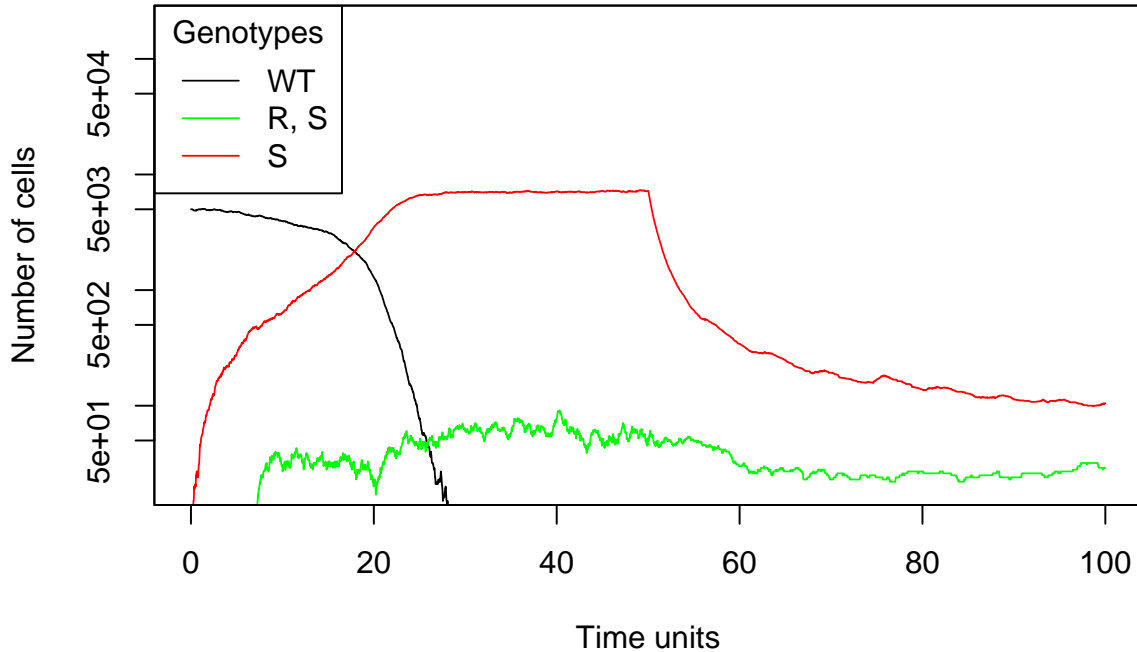
```
## plot(switching_sim, show = "genotypes", ylim = c(20, 1000))
```

15.2 Growth factors as chemotherapy target

It has been reported in the literature and verified in the above simulations, that resistance acquisition is almost unavoidable. A new approach to avoid this evolutionary adaptation, proposes to change the chemotherapy target, from the cell subclones to the growing factors (GF) they produce. These molecules, which are secreted to the medium by **cooperators**, help to grow their own subclone types besides the GF **defectors**. Thus, the fitness of the whole population increases and the tumor grows.

An attended side effect of this type of treatment is the emergence of cooperators that overproduce GF. This will increase their fitness and reduce the impact of GF sequestering agents. However, it would also increase the cost of its production, which will decrease this benefitial impact.

Archetti (2013) develops a series of formulas to relate the impact of cooperation in tumor composition and fitness of subclones, which they divide as cooperators and defectors. Although GFs are not always distributed homogeneously, we will assume they do in the following simulations for the sake of simplicity.

Attending to the theoretical foundations mentioned, we will create a cooperation scenario, including wild-type healthy cells (**WT**) and tumour cells, which are cooperators (**C**), defectors (**D**) and overproducers (**P**).

The system payoff matrix is the following:

```
# WT       Cooperators Defectors Overproducers
a <- 1;    b <- 0.5;   c <- 0.5; m <- 0.75      ## a b c m
d <- 1;    e <- 1.25;  f <- 0.7; o <- 0.185     ## d e f o
g <- 1;    h <- 1.5;   i <- 0.5; p <- 2.5       ## g h i p
```

367

```r
j <- 0.8; k <- 1;     l <- 0.5; q <- 1.5        ## j k l q

payoff_m <- matrix(c(a,b,c,m,d,e,f,o,g,h,i,p,j,k,l,q), ncol=4, byrow=TRUE)
colnames(payoff_m) <- c("WT", "Cooperators", "Defectors", "Overproducers")
rownames(payoff_m) <- c("WT", "Cooperators", "Defectors", "Overproducers")
print(payoff_m <- as.table(payoff_m))
##                  WT Cooperators Defectors Overproducers
## WT            1.000       0.500     0.500         0.750
## Cooperators   1.000       1.250     0.700         0.185
## Defectors     1.000       1.500     0.500         2.500
## Overproducers 0.800       1.000     0.500         1.500
```

The fitness definitions for the four types of cells would be the following:

```r
print( df <- data.frame(
  CellType = c("WT", "C", "D", "P"),
  Fitness = c("F(WT) = ax(WT) + bx(C) + cx(D) + mx(P)",
              "F(C) = dx(WT) + ex(C) + fx(D) + ox(P)",
              "F(D) = gx(WT) + hx(C) + ix(D) + px(P)",
              "F(P) = jx(WT) + kc(C) + lx(D) + qx(P)")),
  row.names = FALSE )
##   CellType                                  Fitness
##         WT F(WT) = ax(WT) + bx(C) + cx(D) + mx(P)
##          C  F(C) = dx(WT) + ex(C) + fx(D) + ox(P)
##          D  F(D) = gx(WT) + hx(C) + ix(D) + px(P)
##          P  F(P) = jx(WT) + kc(C) + lx(D) + qx(P)
```

Ordered by decreasing fitness:

In summary:

- The fitness of WT cells is lower than the tumour cells.
- Without any therapy, C cells fitness is higher than P cells, so the overproducers will not overtake them.
- D subclones benefit from the high concentration of cooperators (C or P).

15.2.1 Scenario without chemotherapy

First, we will study the fitness of the subclones types based on hypothetical frequencies. Then, we will simulate the growth of the tumor without any treatment. For this, we are considering that C cells are the original tumor cells and they can mutate and lose by deletion the GF gene (D cells grow) or duplicate it (P cells grow).

```r
set.seed(2)
RNGkind("L'Ecuyer-CMRG")

## Coefficients
## New coefficients for the interaction with overproducing sensitive:
```

```r
# WT        COOPERATOR  DEFECTOR    OVERPRODUCER
a <- 1;   b <- 0.5;   c <- 0.5;   m <- 0.75   # WT
wt_fitness <- paste0(a, "*f_+", b, "*f_C+", c, "*f_C_D+", m, "*f_C_P")

d <- 1;   e <- 1.25; f <- 0.7;   o <- 1.875  # Cooperator
coop_fitness <- paste0(d, "*f_+", e, "*f_C+", f, "*f_C_D+", o, "*f_C_P")

g <- 1;   h <- 1.5;   i <- 0.5;   p <- 2.5     # Defector
def_fitness <- paste0(g, "*f_+", h, "*f_C+", i, "*f_C_D+", p, "*f_C_P")

j <- 0.8; k <- 1;     l <- 0.5;   q <- 1.5     # Cooperator overproducing
over_fitness <- paste0(j, "*f_+", k, "*f_C+", l, "*f_C_D+", q, "*f_C_P")


## No-chemotherapy
## Fitness definition
coop_no <- data.frame(Genotype = c("WT", "C", "D", "P", "C,D", "C,P", "D,P", "C,D,P
                      Fitness = c(
                        wt_fitness, #WT
                        coop_fitness, #S
                        "0", #D
                        "0", #P
                        def_fitness, #S,D
                        over_fitness, #S,P
                        "0", #D,P
                        "0"  #C,D,P
                        ),
                        stringsAsFactors = FALSE)

game_no <- allFitnessEffects(genotFitness = coop_no,
                        frequencyDependentFitness = TRUE,
                        frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = coop_no,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.


## First plot
eag <- evalAllGenotypes(game_no,
                        spPopSizes = c(WT = 10, C = 10, D = 0, P = 0,
                                       "C, D" = 10, "C, P" = 1,
                                       "D, P" = 0, "C, D, P" = 0))[c(1, 2, 5, 6),]
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```
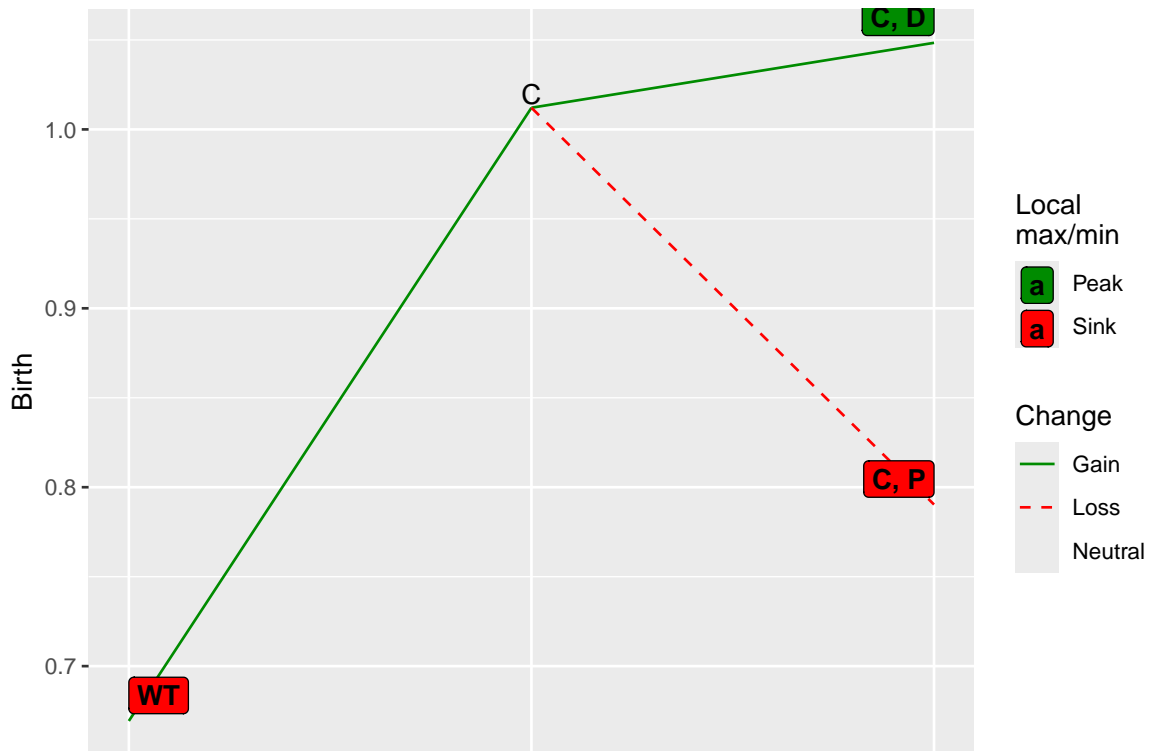
```
plot(eag)
```



```
## Simulation
gamesimul_no <- oncoSimulIndiv(game_no,
                               model = "McFL",
                               onlyCancer = FALSE,
                               finalTime = 35,
                               mu = 0.01,
                               initSize = 5000,
                               keepPhylog = FALSE,
                               seed = NULL)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```
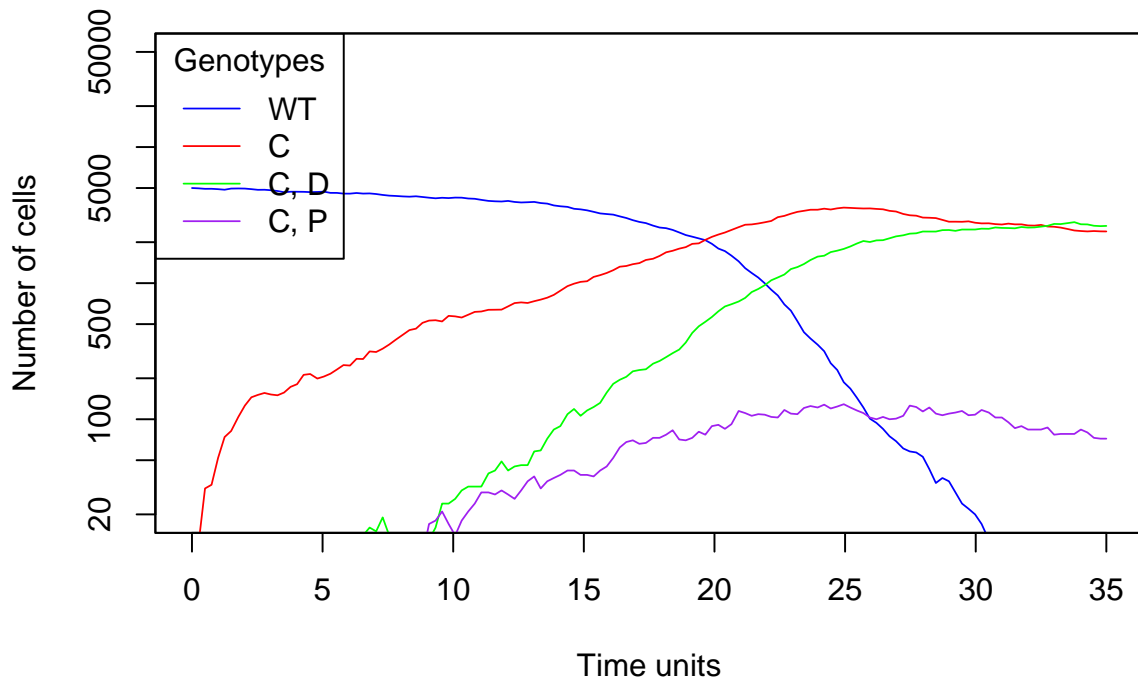
```
## Second plot
plot(gamesimul_no, show = "genotypes", type = "line",
     col = c("blue", "red", "green", "purple"), ylim = c(20, 50000),
     thinData = TRUE)
```

```
# Third plot
## plot(gamesimul_no, show = "genotypes")
```

The resulting plots show that D cells are highly benefited by the second most common subclone activity (C cells), which is a cooperator. On the other hand, subclone P has a lower fitness value because of the GF cost. However, as we can see in the first plot, a small number of clones remain and survive.

15.2.2   Scenario with GF as target for chemotherapy

In this simulation, we will add the drug effect (at time 50) to the fitness of each subclone. We assume that this will cause a reduction of the fitness by the level of GF dependency for each subclone. We now introduce some constants to make it more accurate. As cooperators will keep producing GF, they will be less affected by the sequestration of this molecule, so their coefficients will be greater than 1. In addition, P also will be less affected since its production of GF is greater, so its coefficient will be greater than the C one ($1.5 > 1.1$). On the other hand, as D is a defector, it will be more affected by the treatment, so its coefficient will set to 0.9, smaller than 1.

```
## Chemotherapy - GF Impairing

# Effect of drug on GF availability
# This term is multiplied by the fitness, and reduces the GF available
drug_eff <- 0.25


coop_fix <- data.frame(Genotype = c("WT", "C", "D", "P", "C,D", "C,P", "D,P", "C,D,
                       Fitness = c(
                         wt_fitness,
                         paste0("if (T>50) ", drug_eff, "* 1.2 *(", coop_fitness, ")
```

```r
                              "; else ", coop_fitness, ";"),
                        "0",
                        "0",
                        paste0("if (T>50) ", drug_eff, "*(", def_fitness, ")",
                               "; else ", def_fitness, ";"),
                        paste0("if (T>50) ", drug_eff, "* 1.5 * (", over_fitness, "
                               "; else ", over_fitness, ";"),
                        "0",
                        "0"
                   ),
                   stringsAsFactors = FALSE)

## ** The drug effect is 1.5 times the original because of the overproduction of 
##    full availability of this molecule inside the producing subclone.

period_fix <- allFitnessEffects(genotFitness = coop_fix,
                          frequencyDependentFitness = TRUE,
                          frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = coop_fix,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
```

```r
set.seed(2)
final_time <- 30 ## you'd want this longer; short for speed of vignette
simul_period_fix <- oncoSimulIndiv(period_fix,
                          model = "McFL",
                          onlyCancer = FALSE,
                          finalTime = final_time,
                          mu = 0.01,
                          initSize = 5000,
                          keepPhylog = FALSE,
                          seed = NULL)
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
```

```r
# First plot
plot(simul_period_fix, show = "genotypes", type = "line",
     col = c("blue", "red", "green", "purple"), ylim = c(20, 50000),
     thinData = TRUE)
```
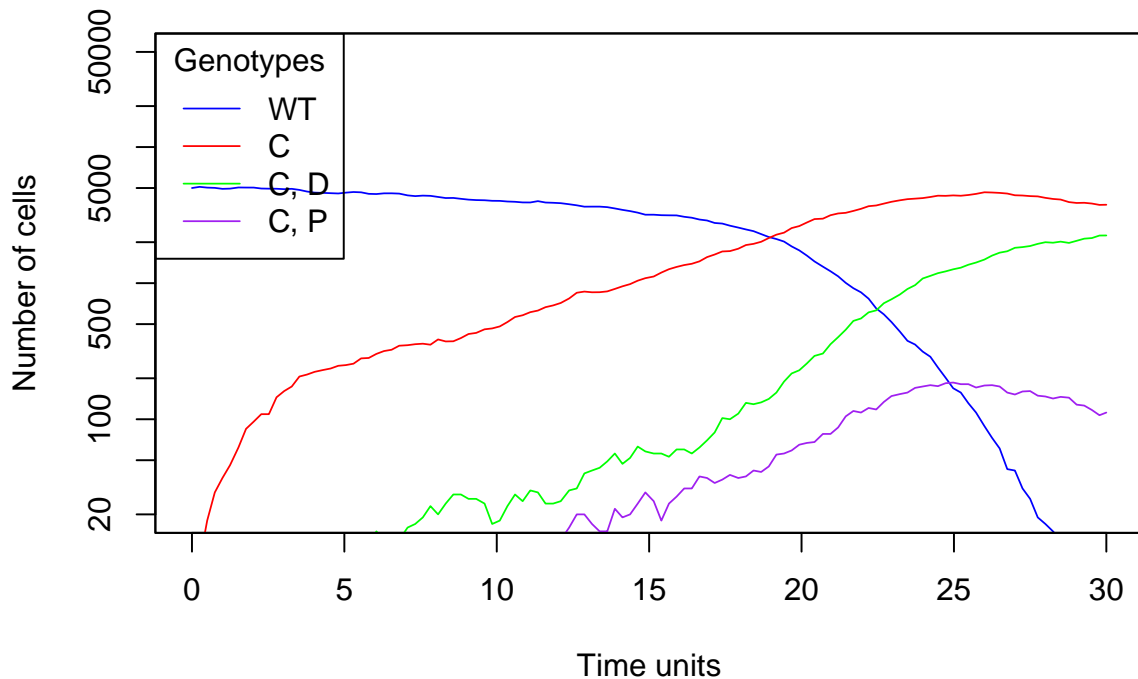
```
# Second plot
## plot(simul_period_fix, show = "genotypes", ylim = c(20, 8000))
```

As expected by Archetti and Pienta Archetti & Pienta (2019), the tumour will be biased towards P mutant. Even if its fitness is lower than C because of its overproduction of GF, it will assure its survival and proliferation when chemotherapy is applied. As a result of this, D does not completely disappear, but reduces dramatically its number.

15.3 Examples using time dependent frequency definition

Here in this chapter we will comment some others approaches that can have this funcionality: increasing or decreasing the fitness as therapeutic interventions, or slow down the collapse of a subpopulation of cells.

15.3.1 Increasing fitness at a certain timepoint

It is possible to increase the fitness value by using T functionality. In the following example we can see how fitness value in genotypes A and B increases when the simulation time reaches a specific value. Since we have used the exponential model, that is the reason why we observe some delay between the specified time and when A or B populations start to grow.

```
dfT1 <- data.frame(Genotype = c("WT", "A", "B"),
              Fitness = c("1",
                          "if (T>50) 1 + 2.35*f_; else 0.50;",
                          "if (T>200) 1 + 0.45*(f_ + f_1); else 0.50;"),
              stringsAsFactors = FALSE)
```

```
afeT1 <- allFitnessEffects(genotFitness = dfT1,
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = dfT1,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```
set.seed(1)
simT1 <- oncoSimulIndiv(afeT1,
                        model = "Exp",
                        mu = 1e-5,
                        initSize = 1000,
                        finalTime = 500,
                        onlyCancer = FALSE,
                        seed = NULL)
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

```
plot(simT1, show = "genotypes", type = "line")
```
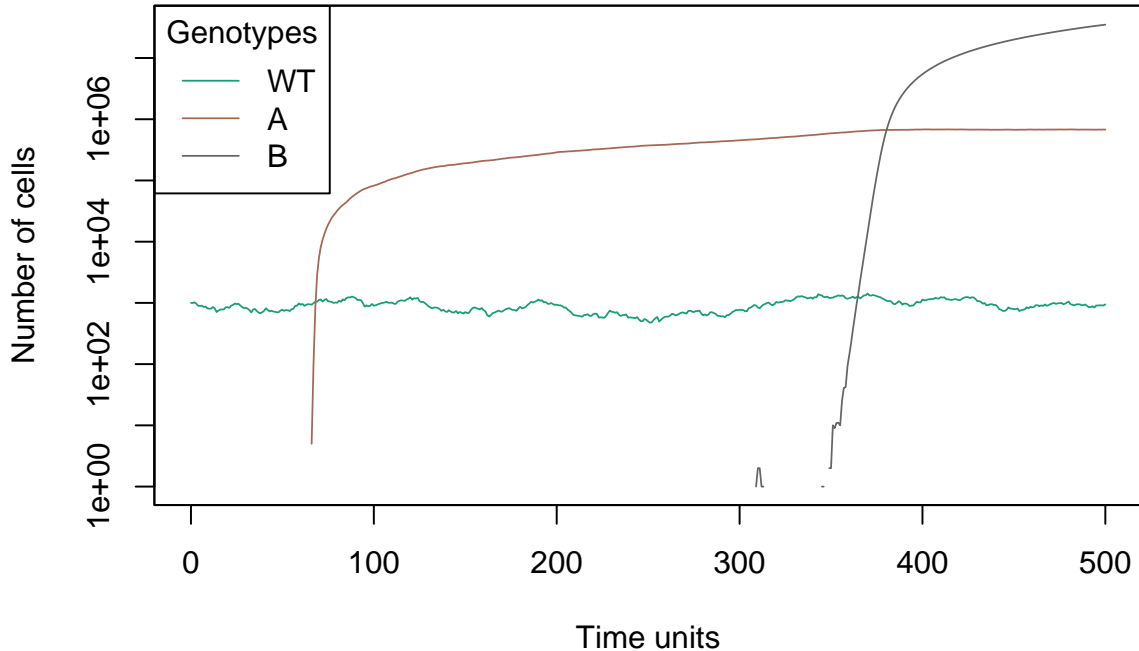

We can check if the fitness values have increased by evaluating the genotypes in the simulation time intervals.

```
evalAllGenotypes(afeT1, spPopSizes = c(10,10,10), currentTime = 49)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
```

374

```
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##   Genotype Fitness
## 2        A    0.5
## 3        B    0.5
```

```r
evalAllGenotypes(afeT1, spPopSizes = c(10,10,10), currentTime = 51)[c(2,3), ]
```

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##   Genotype Fitness
## 2        A   1.783
## 3        B   0.500
```

```r
evalAllGenotypes(afeT1, spPopSizes = c(10,10,10), currentTime = 201)[c(2,3), ]
```

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##           to last version.
##   Genotype Fitness
## 2        A   1.783
## 3        B   1.300
```

15.3.2 Intervention at a certain point to stop subpopulation growth

On the other hand, besides using T functionality to create time intervals, we can also use it as an intervention. In the following example we have used the functionality to increase the fitness value of genotype A, and suddenly decreases it as an intervention, where B population takes advantage to grow, since its fitness is greater now. When the intervention elapses, we can see how A population starts to grow again and outcompetes with wild-type population whose fitness does not change during the simulation.

```r
dfT2 <- data.frame(Genotype = c("WT", "A", "B"),
                   Fitness = c(
                     "1",
                     "if (T>0 and T<50) 0; else if (T>100 and T<150) 0.05; else 1
                     "0.8 + 0.45*(f_)"
                   ),
                   stringsAsFactors = FALSE)
```

```r
afeT2 <- allFitnessEffects(genotFitness = dfT2,
                           frequencyDependentFitness = TRUE,
                           frequencyType = "rel")
## Warning in allFitnessEffects(genotFitness = dfT2,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std
```

```r
## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below

data(simT2)

if (FALSE) {
set.seed(1)
simT2 <- oncoSimulIndiv(afeT2,
                        model = "McFL",
                        mu = 1e-5,
                        initSize = 10000,
                        finalTime = 225,
                        onlyCancer = FALSE,
                        seed = NULL,
                        keepEvery = 1)
}
## Had we not used keepEvery, we'd probably have used
## plot(simT2, show = "genotypes", thinData = TRUE)
plot(simT2, show = "genotypes")
```
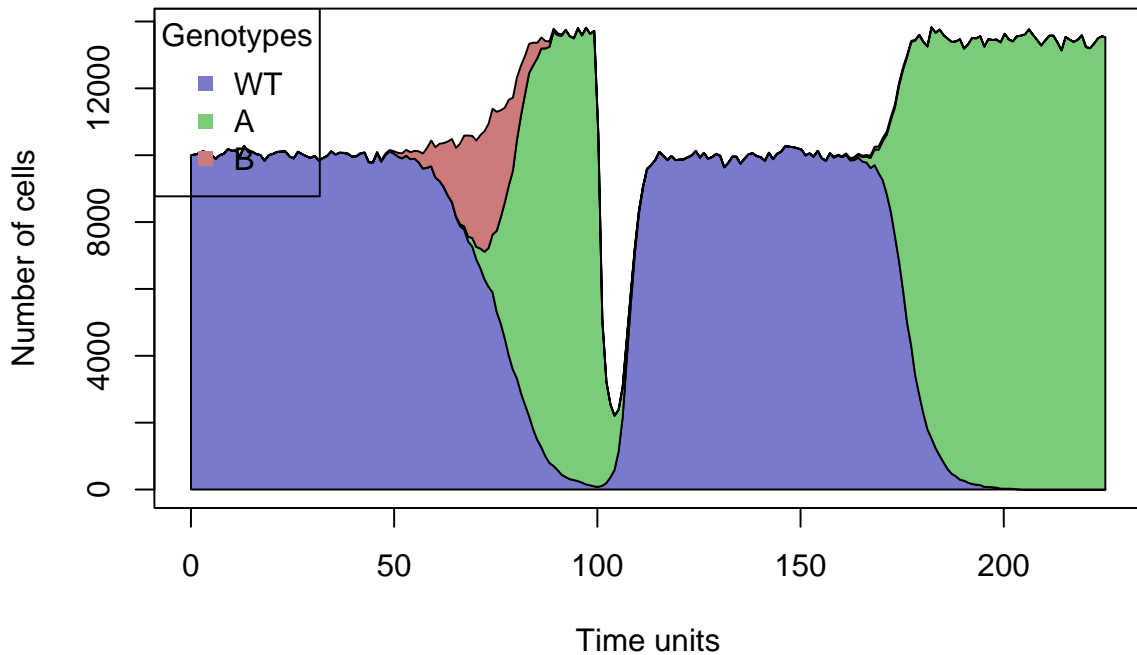
In this case, just like above, we can evaluate the fitness in each time interval and observe how fitness is differente in each one of them.

```
evalAllGenotypes(afeT2, spPopSizes = c(100,10,10), currentTime = 49)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
##   Genotype Fitness
## 2        A   0.000
## 3        B   1.175
```

```
evalAllGenotypes(afeT2, spPopSizes = c(100,10,10), currentTime = 51)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
##   Genotype Fitness
## 2        A   1.492
## 3        B   1.175
```

```
evalAllGenotypes(afeT2, spPopSizes = c(100,10,10), currentTime = 101)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
```

```
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 2        A   0.050
## 3        B   1.175
```

```
evalAllGenotypes(afeT2, spPopSizes = c(100,10,10), currentTime = 201)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##            to last version.
##   Genotype Fitness
## 2        A   1.492
## 3        B   1.175
```

15.3.3 Intervention to slow down collapsing populations

We can use time dependent frequency functionality to slow down a collapsing population by doing an intervention at a certain time interval. In this case, we observe that genotype B has previously a higher fitness than genotype A (as long as the number of cells is greater than 10), but at certain point we can reduce its fitness in order to make A grow and reach B. However, after the time interval, B recovers its initial fitness again and overtakes A, but eventually collapses due to there are no more A's around. This could be a way to slow down the collapse of a population when there is a size dependency between them.

```
dfT3 <- data.frame(Genotype = c("WT", "A", "B"),
                Fitness = c(
                  "1",
                  "1 + 0.2 * (n_2 > 10)",
                  "if (T>50 and T<80) 0.80; else 0.9 + 0.4 * (n_1 > 10)"),
                stringsAsFactors = FALSE)

afeT3 <- allFitnessEffects(genotFitness = dfT3,
                         frequencyDependentFitness = TRUE,
                         frequencyType = "abs")
## Warning in allFitnessEffects(genotFitness = dfT3,
## frequencyDependentFitness = TRUE, : v2 functionality detected.
## Adapting to v3 functionality.
## All single-gene genotypes as input to to_genotFitness_std


## For speed creating the vignette, we load
## precomputed simulation data. Otherwise, run code below
```

```
data(simT3)

if (FALSE) {
set.seed(2)
simT3 <- oncoSimulIndiv(afeT3,
                        model = "McFLD",
                        mu = 1e-4,
                        initSize = 5000,
                        finalTime = 500,
                        onlyCancer = FALSE,
                        seed = NULL,
                        errorHitWallTime = FALSE,
                        errorHitMaxTries = FALSE,
                        keepEvery = 1)
}

plot(simT3, show = "genotypes", type = "line")
```
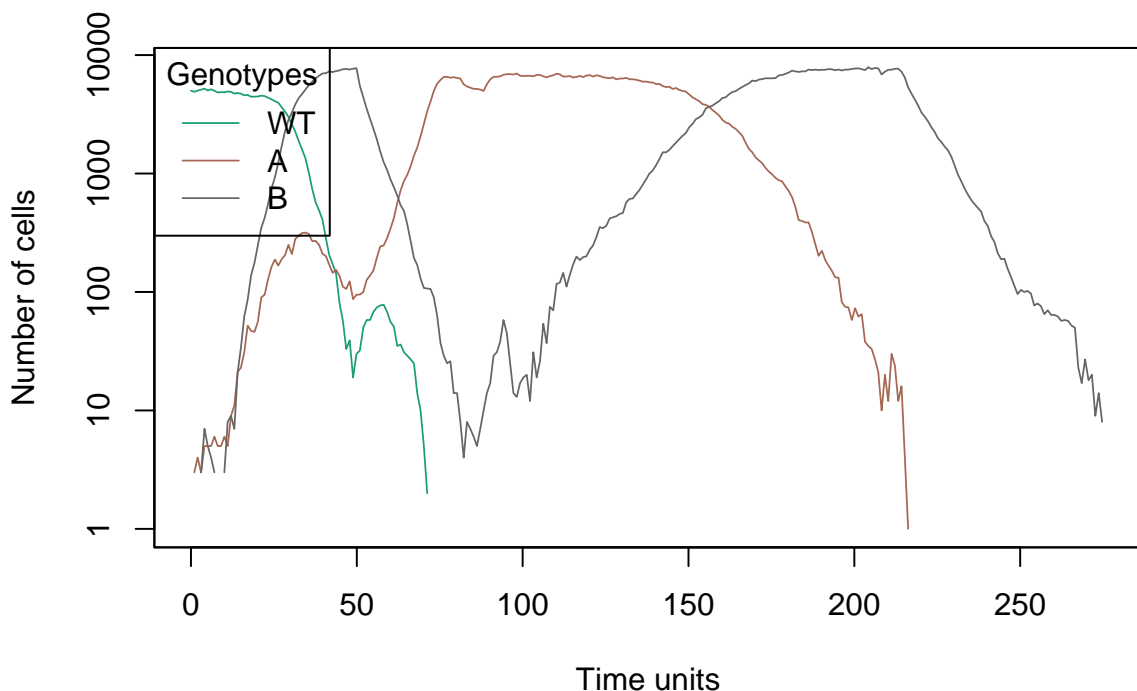

We evaluate the fitness before and after genotype B has a higher and lower fitness respectively, and finally ends up collapsing because the condition of "n_1 > 10" is no longer accomplished.

```
evalAllGenotypes(afeT3, spPopSizes = c(10,10,10), currentTime = 30)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##              to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
```

379

```
##             to last version.
##    Genotype Fitness
## 2         A      1.0
## 3         B      0.9
```

```
evalAllGenotypes(afeT3, spPopSizes = c(11,11,11), currentTime = 79)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
##    Genotype Fitness
## 2         A      1.2
## 3         B      0.8
```

```
evalAllGenotypes(afeT3, spPopSizes = c(11,11,11), currentTime = 81)[c(2,3), ]
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
## Warning in match_spPopSizes(spPopSizes, fmEffects): spPopSizes
## unnamed: cannot check genotype names.
## Using old version of fitnessEffects. Transforming fitnessEffects
##             to last version.
##    Genotype Fitness
## 2         A      1.2
## 3         B      1.3
```

16 Measures of evolutionary predictability and genotype diversity

Several measures of evolutionary predictability have been proposed in the literature (see, e.g., Szendro, Franke, et al. (2013) and references therein). We provide two, Lines of Descent (LOD) and Path of the Maximum (POM), following Szendro, Franke, et al. (2013); we also provide a simple measure of diversity of the actual genotypes sampled.

In Szendro, Franke, et al. (2013) "(. . .) paths defined as the time ordered sets of genotypes that at some time contain the largest subpopulation" are called "Path of the Maximum" (POM) (see their p. 572). In our case, POM are obtained by finding the clone with largest population size whenever we sample and, thus, the POMs will be affected by how often we sample (argument `sampleEvery`), since we are running a continuous time process.

Szendro, Franke, et al. (2013) also define Lines of Descent (LODs) which "(. . .) represent the lineages that arrive at the most populated genotype at the final time". In that same page (572) they provide the details on how the LODs are obtained. Starting with version 2.9.2 of OncoSimulR I only provide an implementation where a single LOD per simulation is returned, with the same meaning as in Szendro, Franke, et al. (2013).

To briefly show some output, we will use again the 5.5 example.

```
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"))

pancr16 <- oncoSimulPop(16, pancr,
                        model = "Exp",                                    onlyCancer = TRUE,
                        mc.cores = 2)

## Look a the first POM
str(POM(pancr16)[1:3])
## List of 3
##  $ : chr [1:2] "" "KRAS"
##  $ : chr [1:2] "" "KRAS"
##  $ : chr [1:2] "" "KRAS"

LOD(pancr16)[1:2]
## [[1]]
## [1] ""      "KRAS"
##
```

```
## [[2]]
## [1] ""      "KRAS"
```

```
## The diversity of LOD (lod_single) and POM might or might not
## be identical
diversityPOM(POM(pancr16))
## [1] 0.2338
```

```
diversityLOD(LOD(pancr16))
## [1] 0.2338
```

```
## Show the genotypes and their diversity (which might, or might
## not, differ from the diversity of LOD and POM)
sampledGenotypes(samplePop(pancr16))
##
##  Subjects by Genes matrix of 16 subjects and 7 genes.
##      Genotype Freq
## 1        KRAS   15
## 2 KRAS, SMAD4    1
##
##  Shannon's diversity (entropy) of sampled genotypes: 0.2338
```

Beware, however, that if you use multiple initial mutants (section 6.7) the LOD
function will probably not do what you want. It is not even clear that the LOD is
well defined in this case. We are working on this.

17 Generating random DAGs for restrictions

You might want to randomly generate DAGs like those often found in the literature on Oncogenetic trees et al. Function `simOGraph` might help here.

```
## No seed fixed, so reruns will give different DAGs.
(a1 <- simOGraph(10))
##       Root 1 2 3 4 5 6 7 8 9 10
## Root    0 1 1 1 0 0 0 0 0 0  0
## 1       0 0 0 0 0 0 0 0 1 0  0
## 2       0 0 0 0 1 1 0 0 0 0  0
## 3       0 0 0 0 0 0 1 0 1 0  1
## 4       0 0 0 0 0 0 0 0 0 1  0
## 5       0 0 0 0 0 0 0 0 0 0  1
## 6       0 0 0 0 0 0 0 1 0 0  0
## 7       0 0 0 0 0 0 0 0 0 0  0
## 8       0 0 0 0 0 0 0 0 0 0  0
## 9       0 0 0 0 0 0 0 0 0 0  0
## 10      0 0 0 0 0 0 0 0 0 0  0
```

```
library(graph) ## for simple plotting
plot(as(a1, "graphNEL"))
```
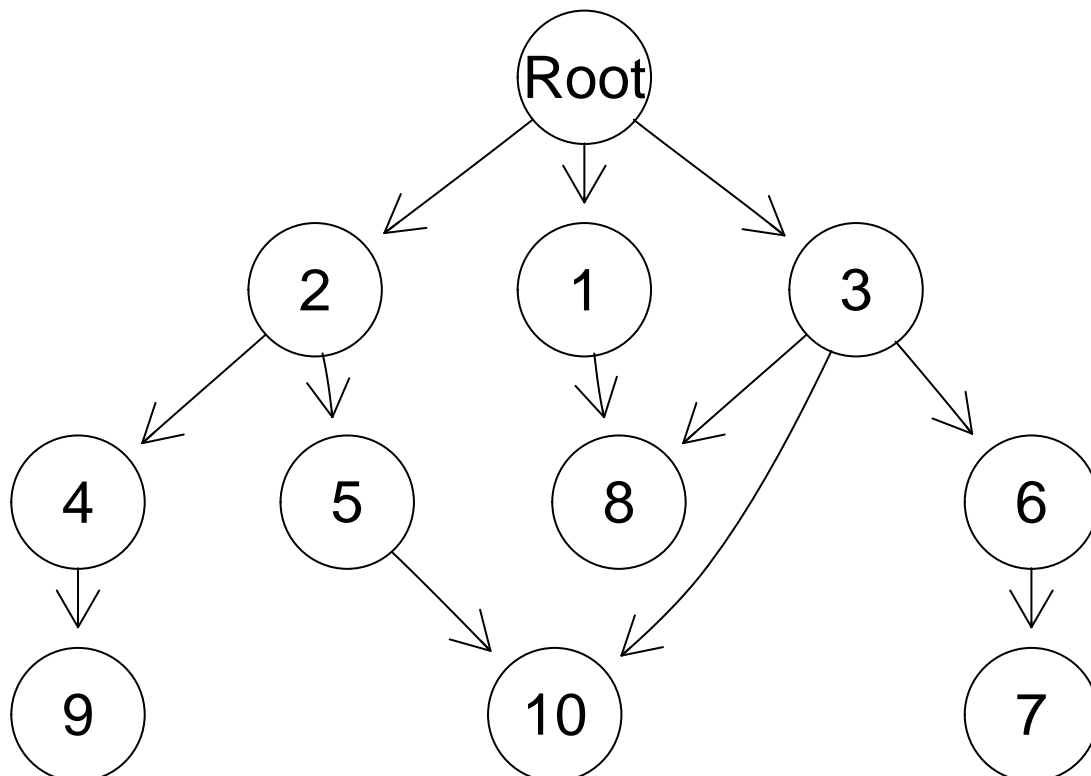

Once you obtain the adjacency matrices, it is for now up to you to convert them into appropriate posets or fitnessEffects objects.

Why this function? I searched for, and could not find any that did what I wanted,

in particular bounding the number of parents, being able to specify the approximate depth[15] of the graph, and optionally being able to have DAGs where no node is connected to another both directly (an edge between the two) and indirectly (there is a path between the two through other nodes). So I wrote my own code. The code is fairly simple to understand (all in file `generate-random-trees.R`). I would not be surprised if this way of generating random graphs has been proposed and named before; please let me know, best if with a reference.

Should we remove direct connections if there are indirect? Or, should we set `removeDirectIndirect = TRUE`? Setting `removeDirectIndirect = TRUE` is basically asking for the transitive reduction of the generated DAG. Except for Farahani & Lagergren (2013) and Ramazzotti et al. (2015), none of the DAGs I've seen in the context of CBNs, Oncogenetic trees, etc, include both direct and indirect connections between nodes. If these exist, reasoning about the model can be harder. For example, with CBN (AND or CMPN or monotone relationships) adding a direct connection makes no difference iff we assume that the relationships encoded in the DAG are fully respected (e.g., all $s_h = -\infty$). But it can make a difference if we allow for deviations from the monotonicity, specially if we only check for the satisfaction of the presence of the immediate ancestors. And things get even trickier if we combine XOR with AND. Thus, I strongly suggest you leave the default `removeDirectIndirect = TRUE`. If you change it, you should double check that the fitnesses of the possible genotypes are what you expect. In fact, I would suggest that, to be sure you get what you think you should get, you convert the fitness from the DAG to a fitness table, and pass that to the simulations, and this requires using non-exposed user functions; to give you an idea, this could work (but you've been warned: this is dangerous!)

```r
g2 <- simOGraph(4, out = "rT", removeDirectIndirect = FALSE)


fe_from_d <- allFitnessEffects(g2)
fitness_d <- evalAllGenotypes(fe_from_d)


fe_from_t <- allFitnessEffects(genotFitness =
                        OncoSimulR:::allGenotypes_to_matrix(fitness_d))


## Compare
fitness_d
(fitness_t <- evalAllGenotypes(fe_from_t))

identical(fitness_d, fitness_t)


## ... but to be safe use fe_from_t as the fitnessEffects object for simulations
```

_____

[15]Where depth is defined in the usual way to mean smallest number of nodes —or edges— to traverse to get from the bottom to the top of the DAG.

18   FAQ, odds and ends

18.1   What we mean by "clone"; and "I want clones disregarding passengers"

In this vignette we often use "clone" or "genotype" interchangeably. A clone denotes a set of cells that have identical genotypes. So if you are using a fitness specification with four genes (i.e., your genome has only four loci), there can be up to $16 = 2^4$ different genotypes or clones. Any two entities that differ in the genotype are different clones. And this applies regardless of whether or not you declare that some genes (loci) are drivers or not. So if you have four genes, it does not matter whether only the first or all four are regarded as drivers; you will always have at most 16 different clones or 16 different genotypes. Of course you can arrive at the same clone/genotype by different routes. Just think about loci A and B in our four-loci genome, and how you can end up with a cell with both A and B mutated.

Analogously, if you have 100 genes, 10 drivers and 90 passengers, you can have up to 2^{100} different clones or genotypes. Sure, one cell might have driver A mutated and passenger B mutated, and another cell might have driver A mutated and passenger C mutated. So if you only look at drivers you might be tempted to say that they are "the same clone for all practical purposes"; but they really are not the same clone as they differ in their genotype and this makes a lot of difference computationally.

If you want summaries of simulations that collapse over some genes (say, some "passengers", the 90 passengers we just mentioned) look at the help for `samplePop`, argument `geneNames`. This would allow you, for instance, to look at the diversity of clones/genotypes, considering as identical those genotypes that only differ in genes you deem relevant; something similar to defining a "drivers' clone" as the set formed from the union of all sets of cells that have identical genotype with respect to only the drivers (so that in the example of "A, B" and "A, C" just mentioned both cells would be considered "the same clone" as they only differ with respect to passengers). However, this "disregard some genes" only applies to summaries of simulations once we are done simulating data. OncoSimulR will always track clones, as defined above, regardless of whether many of those clones have the same genotype if you were to only focus on driver genes; see also section 18.2.

Labeling something as a "driver", therefore, does not affect what we mean by clone. Yes, labeling something as a driver can affect when you stop simulations if you use `detectionDrivers` as a stopping mechanism (see section 6.3). But, again, this has nothing to do with the definition of "clone".

If this is all obvious to you, ignore it. I am adding it here because I've seen strange misunderstandings that eventually could be traced to the apparently multiple meanings of clone. (And to make the story complete, Mather et al. (2012) use the expression "class" —e.g., Algorithm 4 in the paper, Algorithm 5 in the supplementary material).

18.2 Does OncoSimulR keep track of individuals or of clones? And how can it keep track of such large populations?

OncoSimulR keeps track of clones, where a clone is a set of cells that are genetically identical (note that this means completely identical over the whole set of genes/markers you are using; see section 18.1). We do not need to keep track of individual cells because, for all purposes, and since we do not consider spatial structure, two or more cells that are genetically identical are interchangeable. This means, for instance, that the computational cost of keeping a population of a single clone with 1 individual or with 10^9 individuals is exactly the same: we just keep track of the genotype and the number of cells. (Sure, it is much more likely we will see a mutation soon in a clone with 10^9 cells than in a clone with 1, but that is a different issue.)

Of course, the entities that die, reproduce, and mutate are individual cells. This is of course dealt with by tracking clones (as is clearly shown by Algorithms 4 and 5 in Mather et al. (2012)). Tracking individuals, as individuals, would provide no advantage, but would increase the computational burden by many orders of magnitude.

18.2.1 `sampleEvery`, `keepPhylog`, and pruning

At each sampling time (where `sampleEvery` determines the time units between sampling times) the abundance of all the clones with number of cells > 0 is recorded. This is the structure that at the end of the run is converted into the `pops.by.time` matrix.

Now, some clones might arise from mutation between successive population samples but these clones might be extinct by the time we take a population sample. These clones do not appear in the `pops.by.time` matrix because, as we just said, they have 0 cells at the time of sampling. Of course, some of these clones might appear again later and reach a size larger than 0 at some posterior sampling time; it is at this time when this/these clone(s) will appear in the `pops.by.time` matrix. This pruning of clones with 0 cells can allow considerable savings in computing time (OncoSimulR needs to track the genotype of clones, their population sizes, their birth, death, and mutation rates, their next mutation time and the last time they were updated and thus it is important that we only loop over structures with information that is really needed).

However, we still need to track clones as clones, not simply as classes such as "number of mutated genes". Therefore, very large genomes can represent a problem if they lead to the creation and tracking of many different clones (even if they have the same number of mutated genes), as we have seen, for instance, in section 2.3. In this case, programs that only keep track of numbers of mutated genes or of drivers, not individual clones, can of course achieve better speed.

What about the genealogy? If you ask OncoSimulR to keep track of the complete parent-child relationships (`keepPhylog = TRUE`), you might see in the genealogy clones that are not present in `pops.by.time` if these are clones that never had a

population size larger than 0 at any sampling time. To give an example, suppose that we will take population samples at times 0, 1, and 2. Clone A, with a population size larger than 0 at time 1, gives rise at time 1.5 to clone B; clone B then gives rise to clone C at time 1.8. Finally, suppose that at time 2 only clone C is alive. In other words, when we carry out the update of the population with Algorithm 5 from Mather et al. (2012), clones A and B have size 0. Now, at time 1 clones B and C did not yet exist, and clone B is never alive at times 1 or 2. Thus, clone B is not present in `pops.by.time`. But we cannot remove clone B from our genealogy if we want to reflect the complete genealogy of C. Thus, `pops.by.time` will show only clones A and C (not B) but the complete genealogy will show clones A, B, C (and will show that B appeared from A at time 1.5 and C appeared from B at time 1.8). Since function `plotClonePhylog` offers a lot of flexibility with respect to what clones to show depending on their population sizes at different times, you can prevent being shown B, but its existence is there should you need it (see also 2.3.5).

18.3   Dealing with errors in "oncoSimulPop"

When running OncoSimulR under Windows `mclapply` does not use multiple cores, and errors from `oncoSimulPop` are reported directly. For example:

```r
## This code will only be evaluated under Windows
if(.Platform$OS.type == "windows")
    try(pancrError <- oncoSimulPop(10, pancr,
                                   initSize = 1e-5,
                                   onlyCancer = TRUE,
                                   detectionSize = 1e7,
                                   keepEvery = 10,
                              mc.cores = 2))
```

Under POSIX operating systems (e.g., GNU/Linux or Mac OSX) `oncoSimulPop` can ran parallelized by calling `mclapply`. Now, suppose you did something like

```r
## Do not run under Windows
if(.Platform$OS.type != "windows")
    pancrError <- oncoSimulPop(10, pancr,
                               initSize = 1e-5,
                               onlyCancer = TRUE,
                               detectionSize = 1e7,
                               keepEvery = 10,
                               mc.cores = 2)
## Warning in mclapply(seq.int(Nindiv), function(x) oncoSimulIndiv(fp
## = fp, : all scheduled cores encountered errors in user code
```

The warning you are seeing tells you there was an error in the functions called by `mclapply`. If you check the help for `mclpapply` you'll see that it returns a try-error object, so we can inspect it. For instance, we could do:

```
pancrError[[1]]
```

But the output of this call might be easier to read:

```
pancrError[[1]][1]
```

And from here you could see the error that was returned by `oncoSimulIndiv`: `initSize < 1` (which is indeed true: we pass `initSize = 1e-5`).

18.4 Whole tumor sampling, genotypes, and allele counts: what gives? And what about order?

You are obtaining genotypes, regardless of order. When we use "whole tumor sampling", it is the frequency of the mutations in each gene that counts, not the order. So, for instance, "c, d" and "c, d" both contribute to the counts of "c" and "d". Similarly, when we use single cell sampling, we obtain a genotype defined in terms of mutations, but there might be multiple orders that give this genotype. For example, $d > c$ and $c > d$ both give you a genotype with "c" and "d" mutated, and thus in the output you can have two columns with both genes mutated.

18.5 Doesn't the BNB algorithm require small mutation rates for it to be advantageous?

As discussed in the original paper by Mather et al. (2012) (see also their supplementary material), the BNB algorithm can achieve considerable speed advantages relative to other algorithms especially when mutation events are rare relative to birth and death events; the larger the mutation rate, the smaller the gains compared to other algorithms. As mentioned in their supplementary material (see p.5) "Note that the 'cost' of each step in BNB is somewhat higher than in SSA [SSA is the original Gillespie's Stochastic Simulation Algorithm] since it requires generation of several random numbers as compared to only two uniform random numbers for SSA. However this cost increase is small compared with significant benefits of jumping over birth and death reactions for the case of rare mutations."

Since the earliest versions, OncoSimulR has provided information to assess these issues. The output of function `oncoSimulIndiv` includes a list called *"other"* that itself includes two lists named *"minDMratio"* and *"minBMratio"*, the smallest ratio, over all simulations, of death rate to mutation rate or birth rate to mutation rate, respectively. As explained above, the BNB algorithm thrives when those are large. Note, though, we say "it thrives": these ratios being large is not required for the BNB algorithm to be an exact simulation algorithm; these ratios being large make BNB comparatively much faster than other algorithms.

18.6   Can we use the BNB algorithm with state-dependent birth or death rates?

As discussed in the original paper by Mather et al. (2012) (see sections 2.6 and 3.2 of the paper and section E of the supplementary material), the BNB algorithm can be used as an approximate stochastic simulation algorithm "(. . .) with non-constant birth, death, and mutation rates by evolving the system with a BNB step restricted to a short duration t." (p. 9 in supplementary material). The justification is that "(. . .) the propensities for reactions can be considered approximately constant during some short interval." (p. 1234). This is the reason why, when we use McFarland's model, we set a very short `sampleEvery`. In addition, the output of the simulation functions contains the simple summary statistic `errorMF` that can be used to assess the quality of the approximation[16].

Note that, as the authors point out, approximations are common with stochastic simulation algorithms when there is density dependence, but the advantage of the BNB algorithm compared to, say, most tau-leap methods is that clones of different population sizes are treated uniformly. Mather et al. (2012) further present results from simulations comparing the BNB algorithm with the original direct SSA method and the tau-leaps (see their Fig. 5), which shows that the approximation is very accurate as soon as the interval between samples becomes reasonably short.

18.7   Sometimes I get exceptions when running with mutator genes

Yes, sure, the following will cause an exception; this is similar to the example used in 1.3.4 but there is one crucial difference:

```
sd <- 0.1 ## fitness effect of drivers
sm <- 0 ## fitness effect of mutator
nd <- 20 ## number of drivers
nm <- 5  ## number of mutators
mut <- 50 ## mutator effect  THIS IS THE DIFFERENCE

fitnessGenesVector <- c(rep(sd, nd), rep(sm, nm))
names(fitnessGenesVector) <- 1:(nd + nm)
```

---

[16]Death rates are affected by density dependence and, thus, it is on the death rates where the approximation that they are constant over a short interval plays a role. Thus, we examine how large the difference between successive death rates is. More precisely, let A and C denote two successive sampling periods, with $D_A = log(1 + N_A/K)$ and $D_C = log(1 + N_C/K)$ their death rates. `errorMF_size` stores the largest $abs(D_C - D_A)$ between any two sampling periods ever seen during a simulation. `errorMF` stores the largest $abs(D_C - D_A)/D_A$. Additionally, a simple procedure to use is to run the simulations with different values of `sampleEvery`, say the default value of 0.025 and values that are 10, 20, and 50 times larger or smaller, and assess their effects on the output of the simulations and the `errorMF` statistic itself. You can check that using a `sampleEvery` much smaller than 0.025 rarely makes any difference in `errorMF` or in the simulation output (though it increases computing time significantly). And, just for the fun of it, you can also check that using huge values for `sampleEvery` can lead to trouble and will be manifested too in the simulation output with large and unreasonable jumps in total population sizes and sudden extinctions.

```
mutatorGenesVector <- rep(mut, nm)
names(mutatorGenesVector) <- (nd + 1):(nd + nm)

ft <- allFitnessEffects(noIntGenes = fitnessGenesVector,
                        drvNames = 1:nd)
mt <- allMutatorEffects(noIntGenes = mutatorGenesVector)
```

Now, simulate using the fitness and mutator specification. We fix the number of drivers to cancer, and we stop when those numbers of drivers are reached. Since we only care about the time it takes to reach cancer, not the actual trajectories, we set `keepEvery = NA`:

```
ddr <- 4
set.seed(2)
RNGkind("L'Ecuyer-CMRG")
st <- oncoSimulPop(4, ft, muEF = mt,
                   detectionDrivers = ddr,
                   finalTime = NA,
                   detectionSize = NA,
                   detectionProb = NA,
                   onlyCancer = TRUE,
                   keepEvery = NA,
                   mc.cores = 2, ## adapt to your hardware
                   seed = NULL) ## for reproducibility
## set.seed(NULL) ## return things to their "usual state"
```

What happened? That you are using five mutator genes, each with an effect of multiplying by 50 the mutation rate. So the genotype with all those five genes mutated will have an increased mutation rate of $50^5 = 312500000$. If you set the mutation rate to the default of $1e-6$ you have a mutation rate of 312 which makes no sense (and leads to all sorts of numerical issues down the road and an early warning).

Oh, but you want to accumulate mutator effects and have some, or the early ones, have a large effects and the rest progressively smaller effects? You can do that using epistatic effects for mutator effects.

18.8   What are good values of `sampleEvery`?

First, we need to differentiate between the McFarland and the exponential models. If you use the McFarland model, you should read section 18.6 but, briefly, the small default is probably a good choice.

With the exponential model, however, simulations can often be much faster if `sampleEvery` is large. How large? As large as you can make it. `sampleEvery` should not be larger than your desired `keepEvery`, where `keepEvery` determines the resolution or granularity of your samples (i.e., how often you take a snapshot of the population). If you only care about the final state, then set `keepEvery = NA`.

The other factors that affects choosing a reasonable `sampleEvery` are mutation rate and population size. If population growth is very fast or mutation rate very large, you need to sample frequently to avoid the "Recoverable exception ti set to DBL_MIN. Rerunning." issue (see discussion in section 2.4).

18.9   mutationPropGrowth and is mutation associated to division?

With BNB mutation is actually "mutate after division": p. 1232 of Mather et al., 2012 explains: "(...) mutation is simply defined as the creation and subsequent departure of a single individual from the class". Thus, if we want individuals of clones/genotypes/populations that divide faster to also produce more mutants per unit time (per individual) we have to set `mutationPropGrowth = TRUE`.

When `mutationPropGrowth = FALSE`, two individuals, one from a fast growing genotype, and the other from a slow growing genotype, would be "emitting" (giving rise to) different numbers of identical (non-mutated) descendants per unit time, but they would be giving rise to the same number of mutated descendants per unit time.

There is an example in Mather et al, p. 1234, section 3.1.1 where "Mutation rate is proportional to growth rate (faster growing species also mutate faster)".

Of course, this only makes sense in models where birth rate changes.

18.10   Messages about 'Using old version of fitnessEffects' and 'v2 functionality detected. Adapting to v3 functionality.'

New functionality has been added that allows us to specify birth and death separately, including making each frequency dependent. See, for example, 12. We still allow the old specification where "fitness" actually meant birth rates, and death was fixed for each model (and the other way around for the Bozic model). You can continue using the old specification. And we continue to do so in most of this vignette (the plan is to eventually update all examples).

19 Session info and packages used

This is the information about the version of R and packages used:

```r
sessionInfo()
## R version 4.4.1 Patched (2024-06-25 r86847)
## Platform: x86_64-pc-linux-gnu
## Running under: Debian GNU/Linux trixie/sid
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas-openmp/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/openblas-openmp/libopenblasp-r0.3.27.so;  LA
##
## Random number generation:
##  RNG:     L'Ecuyer-CMRG
##  Normal:  Inversion
##  Sample:  Rejection
##
## locale:
##  [1] LC_CTYPE=en_GB.utf8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.utf8        LC_COLLATE=en_GB.utf8
##  [5] LC_MONETARY=en_GB.utf8    LC_MESSAGES=en_GB.utf8
##  [7] LC_PAPER=en_GB.utf8       LC_NAME=C
##  [9] LC_ADDRESS=C              LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.utf8 LC_IDENTIFICATION=C
##
## time zone: Europe/Madrid
## tzcode source: system (glibc)
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets
## [7] methods   base
##
## other attached packages:
## [1] igraph_2.0.3      graph_1.83.0       BiocGenerics_0.51.0
## [4] OncoSimulR_4.7.1  pander_0.6.5       bookdown_0.39
## [7] BiocStyle_2.33.1  rmarkdown_2.27
##
## loaded via a namespace (and not attached):
##  [1] utf8_1.2.4        generics_0.1.3     gtools_3.9.5
##  [4] lattice_0.22-6    stringi_1.8.4      digest_0.6.36
##  [7] magrittr_2.0.3    evaluate_0.24.0    grid_4.4.1
## [10] RColorBrewer_1.1-3 fastmap_1.2.0     Matrix_1.7-0
## [13] ggrepel_0.9.5     tinytex_0.51       BiocManager_1.30.23
## [16] fansi_1.0.6       scales_1.3.0       Rgraphviz_2.49.0
## [19] smatr_3.4-8       abind_1.4-5        cli_3.6.3
## [22] crayon_1.5.3      rlang_1.1.4        munsell_0.5.1
```

```
## [25] withr_3.0.0          yaml_2.3.8         tools_4.4.1
## [28] dplyr_1.1.4          colorspace_2.1-0   ggplot2_3.5.1
## [31] vctrs_0.6.5          R6_2.5.1           png_0.1-8
## [34] stats4_4.4.1         lifecycle_1.0.4    stringr_1.5.1
## [37] car_3.1-2            pkgconfig_2.0.3    pillar_1.9.0
## [40] gtable_0.3.5         data.table_1.15.4  glue_1.7.0
## [43] Rcpp_1.0.12          xfun_0.45          tibble_3.2.1
## [46] tidyselect_1.2.1     highr_0.11         knitr_1.47
## [49] farver_2.1.2         htmltools_0.5.8.1  carData_3.0-5
## [52] compiler_4.4.1
```

19.1 Time it takes to build the vignette and most time consuming chunks

Time to build the vignette:

```
## [1] "2.56609760920207 minutes"
```

```
## The 15 most time consuming chunks
sort(unlist(all_times), decreasing = TRUE)[1:15]
##       prbau003bb              fdf2d               rps5
##            2.579              2.278              2.118
##             fdf2c              sps3b exampleSimulDeath
##            2.083              2.051              1.826
##      example5scen1       switchChemo1           timefdf1
##            1.825              1.820              1.689
##      example5scen2        smyeloc1v23  AdaptiveTherapy3
##            1.654              1.594              1.593
##        wasthis111              fdf1c            relar3a
##            1.592              1.585              1.534
```

```
paste("Sum times of chunks = ", sum(unlist(all_times))/60, " minutes")
## [1] "Sum times of chunks =  1.39125513235728  minutes"
```

20 Funding

21 References

Archetti, M. (2013). Evolutionary game theory of growth factor production: Implications for tumour heterogeneity and resistance to therapies. *British Journal of Cancer*, *109*(4), 1056–1062. https://doi.org/10.1038/bjc.2013.336

Archetti, M., & Pienta, K. J. (2019). Cooperation among cancer cells: Applying game theory to cancer. *Nature Reviews Cancer*, *19*(2), 110–117. https://doi.org/10.1038/s41568-018-0083-7

Ashworth, A., Lord, C. J., & Reis-Filho, J. S. (2011). Genetic interactions in cancer progression and treatment. *Cell*, *145*(1), 30–38. https://doi.org/10.1016/j.cell.2011.03.020

Barton, S., K. Y., & Sendova, T. (2018). Modeling of breast cancer through evolutionary game theory. *Mathematical Sciences Publishers*, *11*(4). https://doi.org/10.2140/involve.2018.11.541

Basanta, D., & Deutsch, A. (2008). A Game Theoretical Perspective on the Somatic Evolution of cancer. In *Selected Topics in Cancer Modeling: Genesis, Evolution, Immune Competition, and Therapy* (pp. 1–16). Birkhäuser Boston. https://doi.org/10.1007/978-0-8176-4713-1_5

Basanta, D., Scott, J. G., Fishman, M. N., Ayala, G., Hayward, S. W., & Anderson, A. R. A. (2012). Investigating prostate cancer tumour–stroma interactions: Clinical and biological insights from an evolutionary game. *Br J Cancer*, *106*(1), 174–181. https://doi.org/10.1038/bjc.2011.517

Bauer, B., Siebert, R., & Traulsen, A. (2014). Cancer initiation with epistatic interactions between driver and passenger mutations. *Journal of Theoretical Biology*, *358*, 52–60. https://doi.org/10.1016/j.jtbi.2014.05.018

Beerenwinkel, N., Antal, T., Dingli, D., Traulsen, A., Kinzler, K. W., Velculescu, V. E., Vogelstein, B., & Nowak, M. A. (2007). Genetic progression and the waiting time to cancer. *PLoS Computational Biology*, *3*(11), e225. https://doi.org/10.1371/journal.pcbi.0030225

Beerenwinkel, N., Eriksson, N., & Sturmfels, B. (2007). Conjunctive Bayesian networks. *Bernoulli*, *13*(4), 893–909. https://doi.org/10.3150/07-BEJ6133

Bozic, I., Antal, T., Ohtsuki, H., Carter, H., Kim, D., Chen, S., Karchin, R., Kinzler, K. W., Vogelstein, B., & Nowak, M. A. (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. https://doi.org/10.1073/pnas.1010978107

Brouillet, S., Annoni, H., Ferretti, L., & Achaz, G. (2015). MAGELLAN: A tool to explore small fitness landscapes. *bioRxiv*, 031583. https://doi.org/10.1101/031583

Carvajal-Rodriguez, A. (2010). Simulation of genes and genomes forward in time. *Current Genomics*, *11*(1), 58–61. https://doi.org/10.2174/138920210790218007

Datta, R. S., Gutteridge, A., Swanton, C., Maley, C. C., & Graham, T. A. (2013). Modelling the evolution of genetic instability during tumour progression. *Evolutionary Applications*, *6*(1), 20–33. https://doi.org/10.1111/eva.12024

Desper, R., Jiang, F., Kallioniemi, O. P., Moch, H., Papadimitriou, C. H., & Schäffer, A. A. (1999). Inferring tree models for oncogenesis from comparative genome hybridization data. *J Comput Biol*, *6*(1), 37–51. http://view.ncbi.nlm.nih.gov/pubmed/10223663

Diaz-Uriarte, R. (2015). Identifying restrictions in the order of accumulation of mutations during tumor progression: effects of passengers, evolutionary models, and sampling. *BMC Bioinformatics*, *16*(41), 0–36. https://doi.org/doi:10.1186/s12859-015-0466-7

Doebeli, M., Ispolatov, Y., & Simon, B. (2017). Towards a mechanistic foundation of evolutionary theory. *eLife*, *6*, e23804. https://doi.org/10.7554/eLife.23804

Farahani, H., & Lagergren, J. (2013). Learning oncogenetic networks by reducing to mixed integer linear programming. *PLoS One*, *8*(6), e65773. https://doi.org/10.1371/journal.pone.0065773

Ferretti, L., Schmiegelt, B., Weinreich, D., Yamauchi, A., Kobayashi, Y., Tajima, F., & Achaz, G. (2016). Measuring epistasis in fitness landscapes: The correlation of fitness effects of mutations. *Journal of Theoretical Biology*, *396*, 132–143. https://doi.org/10.1016/j.jtbi.2016.01.037

Franke, J., Klözer, A., Visser, J. A. G. M. de, & Krug, J. (2011). Evolutionary Accessibility of Mutational Pathways. *PLoS Comput Biol*, *7*(8), e1002134. https://doi.org/10.1371/journal.pcbi.1002134

Gerrish, P. J., Colato, A., Perelson, A. S., & Sniegowski, P. D. (2007). Complete

genetic linkage can subvert natural selection. *Proceedings of the National Academy of Sciences of the United States of America*, *104*(15), 6266–6271. https://doi.org/10.1073/pnas.0607280104

Gerstung, M., Baudis, M., Moch, H., & Beerenwinkel, N. (2009). Quantifying cancer progression with conjunctive Bayesian networks. *Bioinformatics (Oxford, England)*, *25*(21), 2809–2815. https://doi.org/10.1093/bioinformatics/btp505

Gerstung, M., Eriksson, N., Lin, J., Vogelstein, B., & Beerenwinkel, N. (2011). The Temporal Order of Genetic and Pathway Alterations in Tumorigenesis. *PLoS ONE*, *6*(11), e27136. https://doi.org/10.1371/journal.pone.0027136

Gerstung, M., Nakhoul, H., & Beerenwinkel, N. (2011). Evolutionary Games with Affine Fitness Functions: Applications to Cancer. *Dynamic Games and Applications*, *1*(3), 370–385. https://doi.org/10.1007/s13235-011-0029-0

Gillespie, J. H. (1984). Molecular Evolution Over the Mutational Landscape. *Evolution*, *38*(5), 1116–1129. https://doi.org/10.2307/2408444

Gillespie, J. H. (1993). Substitution processes in molecular evolution. I. Uniform and clustered substitutions in a haploid model. *Genetics*, *134*(3), 971–981.

Greene, D., & Crona, K. (2014). The changing geometry of a fitness landscape along an adaptive walk. *PLoS Computational Biology*, *10*(5), e1003520. https://doi.org/10.1371/journal.pcbi.1003520

Hansen, E., & Read, A. F. (2020a). Cancer therapy: Attempt cure or manage drug resistance? *Evolutionary Applications*, *13*(7), 1660–1672. https://doi.org/10.1111/eva.12994

Hansen, E., & Read, A. F. (2020b). Modifying Adaptive Therapy to Enhance Competitive Suppression. *Cancers*, *12*(12), 3556. https://doi.org/10.3390/cancers12123556

Hartman, J. L., Garvik, B., & Hartwell, L. (2001). Principles for the buffering of genetic variation. *Science (New York, N.Y.)*, *291*(5506), 1001–1004. https://doi.org/10.1126/science.291.5506.1001

Hjelm, M., Höglund, M., & Lagergren, J. (2006). New probabilistic network models and algorithms for oncogenesis. *J Comput Biol*, *13*(4), 853–865. https://doi.org/10.1089/cmb.2006.13.853

Hoban, S., Bertorelle, G., & Gaggiotti, O. E. (2011). Computer simulations: Tools for population and evolutionary genetics. *Nature Reviews. Genetics*, *13*(2), 110–122. https://doi.org/10.1038/nrg3130

Hothorn, T., Hornik, K., Wiel, M. A. van de, & Zeileis, A. (2006). A lego system for conditional inference. *The American Statistician*, *60*(3), 257–263.

Hothorn, T., Hornik, K., Wiel, M. A. van de, & Zeileis, A. (2008). Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, *28*(8), 1–23. http://www.jstatsoft.org/v28/i08/

Hurlbut, E., Ortega, E., Erovenko, I., & Rowell, J. (2018). Game Theoretical Model of Cancer Dynamics with Four Cell Phenotypes. *Games*, *9*(3), 61. https://doi.org/10.3390/g9030061

Kaznatcheev, A., Vander Velde, R., Scott, J. G., & Basanta, D. (2017). Cancer treatment scheduling and dynamic heterogeneity in social dilemmas of tumour acidity and vasculature. *Br J Cancer*, *116*(6), 785–792. https://doi.org/10.1038/bjc.2017.5

Kerr, B., Riley, M. A., Feldman, M. W., & Bohannan, B. J. M. (2002). Local dispersal promotes biodiversity in a real-life game of rock-paper-scissors. *Nature*, *418*(6894), 171–174. https://doi.org/10.1038/nature00823

Korsunsky, I., Ramazzotti, D., Caravagna, G., & Mishra, B. (2014). *Inference of Cancer Progression Models with Biological Noise*. 1–29. http://arxiv.org/abs/1408.6032v1 http://biorxiv.org/content/ early/2014/08/25/00832

Krug, J. (2019). Accessibility percolation in random fitness landscapes. *arXiv:1903.11913 [Math, q-Bio]*. http://arxiv.org/abs/1903.11913

Masliah, I. F. T. P. B. Y. S. L. C. M. H. M. A. M. S. H. K. O. P. J. X. -J. Y. E. (2007). Dynamics of a-synuclein aggregation and inhibition of pore-like oligomer development by b-synuclein. *FEBS*, *274*(7), 16. https://doi.org/10.1111/j.1742-4658.2007.05733.x

Mather, W. H., Hasty, J., & Tsimring, L. S. (2012). Fast stochastic algorithm for simulating evolutionary population dynamics. *Bioinformatics (Oxford, England)*, *28*(9), 1230–1238. https://doi.org/10.1093/bioinformatics/bts130

Maynard Smith, J. M. (1982). *Evolution and the theory of games*. Cambridge Univ. Press.

McFarland, C. D. (2014). *The role of deleterious passengers in cancer* [PhD thesis, Harvard University]. http://nrs.harvard.edu/urn-3:HUL.InstRepos:13070047

McFarland, C. D., Korolev, K. S., Kryukov, G. V., Sunyaev, S. R., & Mirny, L. A. (2013). Impact of deleterious passenger mutations on cancer progression. *Proceedings of the National Academy of Sciences of the United States of America*, *110*(8), 2910–2915. https://doi.org/10.1073/pnas.1213968110

McFarland, C. D., Mirny, L., & Korolev, K. S. (2014). A tug-of-war between driver and passenger mutations in cancer and other adaptive processes. *Proc Natl Acad Sci U S A*, *111*(42), 15138–15143. https://doi.org/10.1101/003053

Misra, N., Szczurek, E., & Vingron, M. (2014). Inferring the paths of somatic evolution in cancer. *Bioinformatics (Oxford, England)*, *30*(17), 2456–2463. https://doi.org/10.1093/bioinformatics/btu319

Moran, P. A. P. (1962). *Statistical processes of evolutionary theory*. Oxford University Press.

Newton, P. K., & Ma, Y. (2019). Nonlinear adaptive control of competitive release and chemotherapeutic resistance. *Phys. Rev. E*, *99*, 022404. https://doi.org/10.1103/PhysRevE.99.022404

Nowak, M. A. (2006). *Evolutionary dynamics: Exploring the equations of life*. Belknap Press of Harvard University Press.

Ochs, I. E., & Desai, M. M. (2015). The competition between simple and complex evolutionary trajectories in asexual populations. *BMC Evolutionary Biology*, *15*(1), 1–9. https://doi.org/10.1186/s12862-015-0334-0

Orr, H. A. (2002). The population genetics of adaptation: The adaptation of dna sequences. *Evolution*, *56*(7), 1317–1330. https://doi.org/10.1554/0014-3820(2002)056%5B1317:TPGOAT%5D2.0.CO;2

Ortmann, C. A., Kent, D. G., Nangalia, J., Silber, Y., Wedge, D. C., Grinfeld, J., Baxter, E. J., Massie, C. E., Papaemmanuil, E., Menon, S., Godfrey, A. L., Dimitropoulou, D., Guglielmelli, P., Bellosillo, B., Besses, C., Döhner, K., Harrison, C. N., Vassiliou, G. S., Vannucchi, A., . . . Green, A. R. (2015). Effect

of Mutation Order on Myeloproliferative Neoplasms. *New England Journal of Medicine*, *372*, 601–612. https://doi.org/10.1056/NEJMoa1412098

Otto, S. P., & Day, T. (2007). *A biologist's guide to mathematical modeling in ecology and evolution.* Princeton University Press.

Ramazzotti, D., Caravagna, G., Olde Loohuis, L., Graudenzi, A., Korsunsky, I., Mauri, G., Antoniotti, M., & Mishra, B. (2015). CAPRI: Efficient inference of cancer progression models from cross-sectional data. *Bioinformatics (Oxford, England)*, *31*(18), 3016–3026. https://doi.org/10.1093/bioinformatics/btv296

Raphael, B. J., & Vandin, F. (2015). Simultaneous Inference of Cancer Pathways and Tumor Progression from Cross-Sectional Mutation Data. *Journal of Computational Biology*, *22*(00), 250–264. https://doi.org/10.1089/cmb.2014.0161

Reiter, J., Bozic, I., Chatterjee, K., & Nowak, M. (2013). TTP: tool for tumor progression. In N. Sharygina & H. Veith (Eds.), *Computer aided verification, lecture notes in computer science* (pp. 101–106). Springer-Verlag. http://link.springer.com/chapter/10.1007/ 978-3-642-39799-8_6 http://dx.doi.org/10.1007/ 978-3-642-39799-8_6 http://pub.ist.ac.at/ttp/

Sartakhti, J. S., Manshaei, M. H., Bateni, S., & Archetti, M. (2016). Evolutionary Dynamics of Tumor-Stroma Interactions in Multiple Myeloma. *PLOS ONE*, *11*(12), e0168856. https://doi.org/10.1371/journal.pone.0168856

Szabo, A., & Boucher, K. M. (2008). Oncogenetic trees. In W.-Y. Tan & L. Hanin (Eds.), *Handbook of cancer models with applications* (pp. 1–24). World Scientific. http://www.worldscibooks.com/lifesci/6677.html

Szendro, I. G., Franke, J., Visser, J. A. G. M. de, & Krug, J. (2013). Predictability of evolution depends nonmonotonically on population size. *Proceedings of the National Academy of Sciences*, *110*(2), 571–576. https://doi.org/10.1073/pnas.1213613110

Szendro, I. G., Schenk, M. F., Franke, J., Krug, J., & Visser, J. A. G. M. de. (2013). Quantitative analyses of empirical fitness landscapes. *Journal of Statistical Mechanics: Theory and Experiment*, *2013*(01), P01005. https://doi.org/10.1088/1742-5468/2013/01/P01005

Tomlinson, I. P. (1997). Game-theory models of interactions between tumour cells. *European Journal of Cancer*, *33*(9), 1495–1500. https://doi.org/10.1016/S0959-8049(97)00170-6

Tomlinson, I. P., Novelli, M. R., & Bodmer, W. F. (1996). The mutation rate and cancer. *Proceedings of the National Academy of Sciences of the United States of America*, *93*(25), 14800–14803.

Weissman, D. B., Desai, M. M., Fisher, D. S., & Feldman, M. W. (2009). The rate at which asexual populations cross fitness valleys. *Theoretical Population Biology*, *75*(4), 286–300. https://doi.org/10.1016/j.tpb.2009.02.006

Wu, A., & Ross, D. (2016). Evolutionary game between commensal and pathogenic microbes in intestinal microbiota. *Games*, *7*(3). https://doi.org/10.3390/g7030026

Yuan, X., Miller, D. J., Zhang, J., Herrington, D., & Wang, Y. (2012). An Overview of Population Genetic Data Simulation. *Journal of Computational Biology*, *19*(1), 42–54. https://doi.org/10.1089/cmb.2010.0188

Zanini, F., & Neher, R. a. (2012). FFPopSim: An efficient forward simulation

package for the evolution of large populations. *Bioinformatics*, *28*(24), 3332–3333. https://doi.org/10.1093/bioinformatics/bts633