Frequently asked questions

- **Why do I get the "Java heap space" error, and how to fix it ?**

By default, `rJava` allocate 512MB to the Java virtual machine (JVM). When generating large landscapes, this can be insufficient. The solution is, before loading `rflsgen` (and any other package using `rJava`), to increase the memory allocated to the JVM using `options(java.parameters = "-Xmx4g")`. Note that this allocates 4GB to the JVM. If you want to allocate another amount of memory just replace `4g` with the desired value.

- **Why `flsgen_structure()` did not find a structure satisfying my targets ?**

There are two possibilities:

1. If `rflsgen` indicates that "User targets cannot be satisfied", it means that the solver has finished exploring the search tree and that there is no solution satisfying your targets. As the underlying Constraint Programming solver (Choco) relies on exact algorithms, this message is the guarantee that your targets cannot be satisfied. To date, this is not possible to explain why the targets are contradictory in a human-readable way. So what you can do is try to see if there are obvious contradictions in your targets (e.g. two classes that both need to occupy 60% of the landscape), or try with other targets.
2. If `rflsgen` indicates that "User targets could not be satisfied under the specified time limit", it means that the time limit was reached before the solver could find a solution, or prove that there is no solution. By default the time limit in `flsgen_structure()` is 60 seconds, you can increase this limit with the `time_limit` argument, or disable it with the value 0. You can also change the `search_strategy`, which indicate to Choco-solver how to construct its search tree. Although this search strategy does not influence whether the targets will be satisfied or not, some strategies can be more efficient depending on the problem configuration. Available search strategies in Choco are: `""DEFAULT", "RANDOM", "DOM_OVER_W_DEG", "DOM_OVER_W_DEG_REF", "ACTIVITY_BASED", "CONFLICT_HISTORY", "MIN_DOM_LB", "MIN_DOM_UB"` (please refer to Choco documentation if you want more details).

- **Why `flsgen_generate()` failed to generate a landscape ?**

Although `flsgen_structure()` relies on exact search algorithms to identify suitable landscape structures, the spatial generation algorithm used in `flsgen_generate()` is stochastic and can fail. The reason for using a stochastic algorithm is that finding a spatial embedding of a landscape structure is equivalent to the polyomino packing problem, which is known to be NP-Complete (i.e. computational complexity increases

exponentially with problem size). It would not be possible to generate reasonably large landscapes (e.g. larger than 100x100) in a feasible time. Using a stochastic algorithm offers fast runtime, but at the price that it can fail without knowing if a solution exists. In practice, the success of `flsgen_generate()` is highly constrained by the proportion of the non-focal landscape class (NON_FOCAL_PLAND), and to a lesser extent by the `min_distance` parameter. In conclusion, you have two possibilities: (1) if NON_FOCAL_PLAND is less than 15%, you should probably generate a landscape structure with a higher NON_FOCAL_PLAND, (2) else, you can increase the number of trials with the `max_try` and `max_try_patch` parameters or decrease the `min_distance` parameter.

- **Can I specify the spatial extent and resolution of produced landscapes ?**

This can be specified with the `x`, `y`, `epsg`, `resolution_x` and `resolution_y` parameters in the `flsgen_generate()` function.

- **How can I change the spatial configuration of produced landscapes ?**

There are three parameters influencing the spatial configuration of produced landscapes in `flsgen_generate`:

- The `min_distance` parameter, defines the minimum distance between any two patches of the same class.
- The `terrain` parameter, is a continuous raster guiding the generation algorithm.
- The `terrain_dependency` parameter, defines to which extent the generation algorithm is influenced by the terrain.

Terrain rasters can either be generated "on-the-fly" by flsgen, which relies on the diamond-square (or midpoint displacement) algorithm or given as input. This last option makes it possible to use continuous neutral landscapes generated with other software packages such as `NLMR`, or even digital elevation models from real landscapes.

- **Can I speficy the minimum distance between patches of a same class ?**

Yes, with the `min_distance` parameter of `flsgen_generate()`. You can also play with the `min_max_distance`, which makes the minimum distance between the patch variable and comprised between `min_distance` and `min_max_distance`.

- **How does the computing time of rflsgen varies according to input and parameters ?**

As there are a lot of factors that can influence the computing time of `rflsgen`, there is no simple answer to this question.

Regarding `flsgen_structure()`, the number of classes and the landscape dimensions influences the size of the underlying constraint satisfaction problem. But a large problem can be solved very quickly if the solver can take advantage of the targets to filter the search tree. In general, fragmentation indices such as MESH, NPRO, or DIVI, are more difficult to satisfy if the targets only allow one value. However, in most cases `flsgen_structure()` only needs a few seconds (or less) to find a solution.

`flsgen_generate()` has a more predictable computing time, which is mainly driven by the landscape dimension and the total proportion of focal classes. In practice, it is fast even for large landscapes (several millions of cells), and the RAM should be a problem before the computing time.

- **How does `flsgen_structure()` selects the structures to return if there are several possibilities ?**

Most often, there can be hundreds, thousands, or even millions of possible structures satisfying user targets. `flsgen_structure()` relies on Choco-solver to identify suitable landscape structures. The default behaviour of Choco-solver is to return the first solution found. If we ask the solver to find another solution, it will return the second found, etc. The `nb_solutions` parameter in `flsgen_structure()` tells Choco-solver how many solutions it must find before stopping the search. There is also a `search_strategy` parameter in `flsgen_structure()` which indicate to Choco-solver how to construct its search tree. Although this search strategy does not influence whether the targets will be satisfied or not, it can help diversify the generated structure by exploring the search tree in different ways. Available search strategies in

Choco are: `""DEFAULT", "RANDOM", "DOM_OVER_W_DEG", "DOM_OVER_W_DEG_REF", "ACTIVITY_BASED", "CONFLICT_HISTORY", "MIN_DOM_LB", "MIN_DOM_UB"` (please refer to Choco documentation if you want more details).

- **What is Choco Solver ?**

Choco-solver is an open-source Java Constraint Programming solver. It is a reliable solver implemented with state-of-the-art algorithms which have been used in academic and industrial projects for years.

- **What is Constraint Programming ?**

Constraint programming (CP) is a declarative paradigm for modelling and solving constraint satisfaction and constrained optimization problems. In this context declarative means that the modelling of a problem is decoupled from its solving process, which allows the primary focus to be on *what* must be solved rather than describing *how* to solve it. CP is a subfield of artificial intelligence that relies on automated reasoning, constraint propagation and search heuristics. As an exact approach, CP can provide constraint satisfaction and optimality guarantees, as well as enumerate every solution of a problem. In CP, the modeller represents a problem by declaring *variables* whose possible values belong to a specified finite *domain*, by stating *constraints* (mainly logical relations between variables), and eventually by defining an objective function to minimize or maximize. A solution to the problem is an instantiation of every variable such that every constraint is satisfied. As opposed to mixed-integer linear programming, constraints can be non-linear and variables of several types (e.g. integer, real, set, graph). A CP solver then handles the solving process relying on an automated reasoning method alternating a constraint propagation algorithm (deduction process on values within domains that does not lead to any solution) and a backtracking search algorithm. In a nutshell, more than satisfiability, each constraint embeds a filtering algorithm able to detect inconsistent values in variables domains. At each step of the backtracking search algorithm, the solver calls the constraint propagation algorithm that repeatedly applies these algorithms until a fixed point is reached. When it is proven that a part of the search tree contains no solution, the solver rolls back to a previous state and explores another part of the search tree: this is backtracking. Note that most CP solvers are also able to handle Pareto multi-objective optimization.

If you are interested in learning more about Constraint Programming, you can read the Handbook of Constraint Programming: *Rossi, E. F., van Beek, P., & Walsh, T. (2006). Handbook of Constraint Programming.*