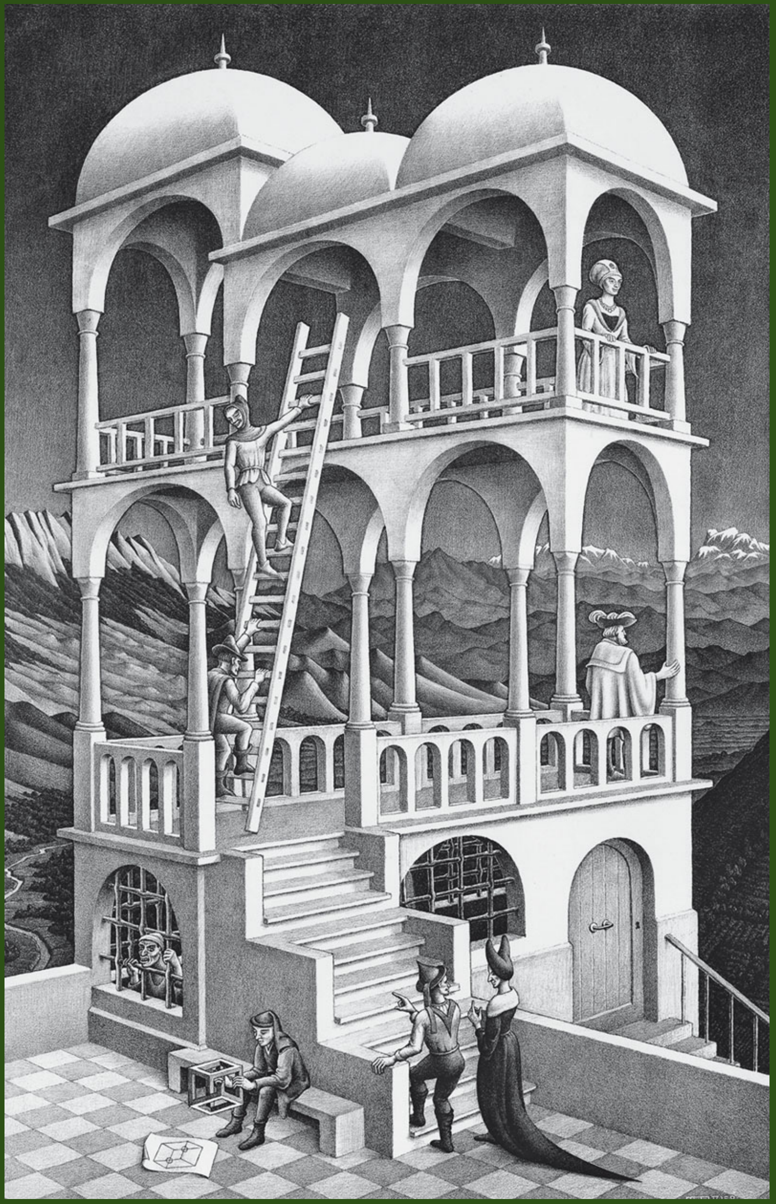
Landscape- and regional-scale simulations

Miquel De Cáceres, Rodrigo Balaguer
Ecosystem Modelling Facility, CREAF

Outline

1. Data structures in `medfateland`
2. Spatially-uncoupled simulations
3. Regional management scenarios
4. Watershed-level simulations
5. Creating spatial inputs I: forest inventory plots
6. Creating spatial inputs II: continuous landscapes

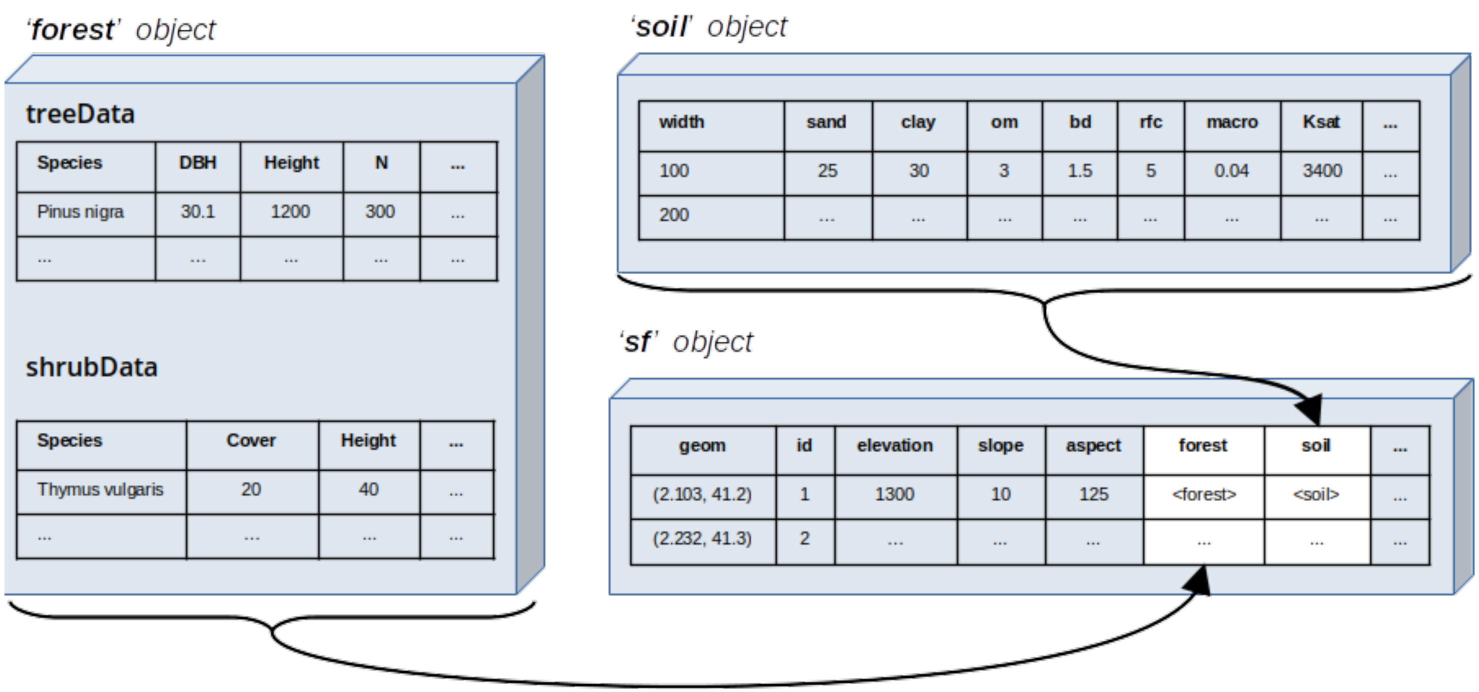


M.C. Escher - Belvedere, 1958

1. Data structures in `medfateland`

Spatial structures (1)

- Current versions of medfateland (ver. > 2.0.0) extensively use package **sf** (simple features) to represent spatial structures, where rows correspond to spatial units (normally point geometries) and columns include either *model inputs* (topography, forest, soil, weather forcing, etc.) or *model outputs*.
- Essentially, an `sf` object is a data frame with spatial (geometry) information and a coordinate reference system.
- Both `forest` and `soil` objects are nested in the corresponding columns of the `sf` object:



Spatial structures (2)

If we load the package we can inspect the structure of an example dataset with 100 forest inventory plots:

```
example_ifn
```

```
Simple feature collection with 100 features and 7 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.817095 ymin: 41.93301 xmax: 2.142956 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 100 × 8
                  geom id        elevation slope aspect land_cover_type soil
 *         <POINT [°]> <chr>         <dbl> <dbl>  <dbl> <chr>           <list>
 1 (2.130641 41.99872) 081015_A1       680  7.73  281.  wildland        <df>
 2 (2.142714 41.99881) 081016_A1       736 15.6   212.  wildland        <df>
 3 (1.828998 41.98704) 081018_A1       532 17.6   291.  wildland        <df>
 4 (1.841068 41.98716) 081019_A1       581  4.79  174.  wildland        <df>
 5 (1.853138 41.98728) 081020_A1       613  4.76   36.9 wildland        <df>
 6 (1.901418 41.98775) 081021_A1       617 10.6   253.  wildland        <df>
 7 (1.937629 41.98809) 081022_A1       622 20.6   360   wildland        <df>
 8  (1.949699 41.9882) 081023_A1       687 14.4   324.  wildland        <df>
 9  (1.96177 41.98831) 081024_A1       597 11.8    16.3 wildland        <df>
10  (1.97384 41.98842) 081025_A1       577 14.6   348.  wildland        <df>
# ℹ 90 more rows
# ℹ 1 more variable: forest <list>
```

Accessing a given position of the `sf` object we can inspect `forest` or `soil` objects:

```
example_ifn$soil[[3]]
```

```
  widths     clay  sand   om       bd      rfc
1    300 25.76667 37.90 2.73 1.406667 23.84454
2    700 27.30000 36.35 0.98 1.535000 31.63389
3   1000 27.70000 36.00 0.64 1.560000 53.90746
4   2000 27.70000 36.00 0.64 1.560000 97.50000
```

Spatial structures (3)

To perform simulations on a gridded landscape we require both an `sf` object and an object `SpatRaster` from package **terra**, which defines the raster topology. For example, the following `sf` describes 65 cells in a small watershed:

```
example_watershed
```

```
Simple feature collection with 66 features and 14 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 401430 ymin: 4671870 xmax: 402830 ymax: 4672570
Projected CRS: WGS 84 / UTM zone 31N
# A tibble: 66 × 15
           geometry    id elevation slope aspect land_cover_type
 *      <POINT [m]> <int>     <dbl> <dbl>  <dbl> <chr>
 1 (402630 4672570)     1      1162 11.3   79.2  wildland
 2 (402330 4672470)     2      1214 12.4   98.7  agriculture
 3 (402430 4672470)     3      1197 10.4  102.   wildland
 4 (402530 4672470)     4      1180  8.12  83.3  wildland
 5 (402630 4672470)     5      1164 13.9   96.8  wildland
 6 (402730 4672470)     6      1146 11.2    8.47 agriculture
 7 (402830 4672470)     7      1153  9.26 356.   agriculture
 8 (402230 4672370)     8      1237 14.5   75.1  wildland
 9 (402330 4672370)     9      1213 13.2   78.7  wildland
10 (402430 4672370)    10      1198  8.56  75.6  agriculture
# ℹ 56 more rows
# ℹ 9 more variables: forest <list>, soil <list>, state <list>,
#   depth_to_bedrock <dbl>, bedrock_conductivity <dbl>, bedrock_porosity <dbl>,
#   snowpack <dbl>, aquifer <dbl>, crop_factor <dbl>
```

The following code defines a 100-m raster topology with the same CRS as the watershed:

```
r <-terra::rast(xmin = 401380, ymin = 4671820, xmax = 402880, ymax = 4672620,
                nrow = 8, ncol = 15, crs = "epsg:32631")
r
```

```
class       : SpatRaster
dimensions  : 8, 15, 1  (nrow, ncol, nlyr)
resolution  : 100, 100  (x, y)
extent      : 401380, 402880, 4671820, 4672620  (xmin, xmax, ymin, ymax)
coord. ref. : WGS 84 / UTM zone 31N (EPSG:32631)
```

Weather forcing in medfateland

There are three ways of supplying weather forcing to simulation functions in **medfateland**, each with its own advantages/ disadvantages:

Supply method	Advantages	Disadvantages
A data frame as parameter `meteo`	Efficient both computationally and memory-wise	Assumes weather is spatially constant
A column `meteo` in `sf` objects	Allows a different weather forcing for each spatial unit	The resulting `sf` is often huge in memory requirements
An interpolator object of class `stars` (or a list of them) as issued from package **meteoland**	More efficient in terms of memory usage	Weather interpolation is performed during simulations, which entails some computational burden

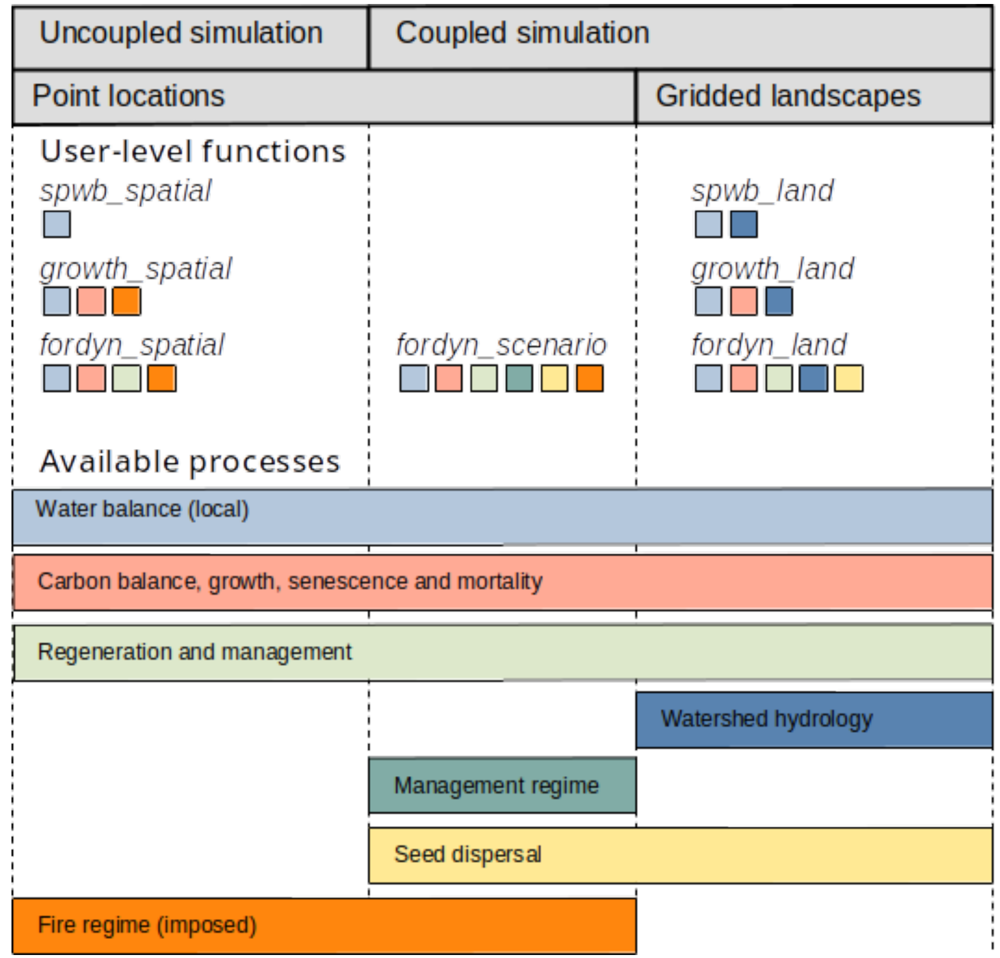
Tip

- If a list of interpolator objects is supplied, each of the interpolators should correspond to a different, consecutive, non-overlapping time period (e.g. 5-year periods).
- Taken together, the interpolators should cover the simulated target period.
- The simulation function will use the correct interpolator for each target date.

2. Spatially-uncoupled simulations

Spatially-uncoupled simulation functions

- Spatially-uncoupled simulations are those where simulations in different stands are completely independent.
- This situation is where *parallelization* is more advantageous.
- Following the nested models of **medfate**, **medfateland** offers functions `spwb_spatial()`, `growth_spatial()` and `fordyn_spatial()` for uncoupled simulations [1].



1. There exist `spwb_spatial_day()` and `growth_spatial_day()` for single-day simulations, paralleling `spwb_day()` and `growth_day()` in **medfate**.

Running spatially-uncoupled simulations

Since it builds on **medfate**, simulations using **medfateland** require *species parameters* and *control parameters* for local simulations:

```
data("SpParamsMED")
local_control <- defaultControl()
```

We can specify the target simulation period as a vector of `Date` or subset the target plots:

```
dates <- seq(as.Date("2001-01-01"), as.Date("2001-01-31"), by="day")
example_subset <- example_ifn[1:5, ]
```

If we are interested in water (or energy) balance, we can use function `spwb_spatial()` as follows:

```
res <- spwb_spatial(example_subset, SpParamsMED, examplemeteo,
                    dates = dates, local_control = local_control)
```

The output is an `sf` object as well, where column `result` contains the results of calling `spwb()` and column `state` contains the final status of `spwbInput` objects:

```
Simple feature collection with 5 features and 3 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.828998 ymin: 41.98704 xmax: 2.142714 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 5 × 4
             geometry id        state            result
          <POINT [°]> <chr>     <list>           <list>
1 (2.130641 41.99872) 081015_A1 <spwbInpt [19]> <spwb [10]>
2 (2.142714 41.99881) 081016_A1 <spwbInpt [19]> <spwb [10]>
3 (1.828998 41.98704) 081018_A1 <spwbInpt [19]> <spwb [10]>
4 (1.841068 41.98716) 081019_A1 <spwbInpt [19]> <spwb [10]>
5 (1.853138 41.98728) 081020_A1 <spwbInpt [19]> <spwb [10]>
```

Using summary functions (1)

Simulations with **medfate** can generate a lot of output. This can be reduced using `control` parameter, but simulation output with **medfateland** can require a lot of memory.

To save memory, it is possible to generate temporal summaries automatically after the simulation of each target forest stand, and avoid storing the full output of the simulation function (using `keep_results = FALSE`).

The key element here is the **summary function** (and possibly, its parameters), which needs to be defined and supplied.

In the following call to `spwb_spatial()` we provide the summary function for `spwb` objects available in **medfate**:

```
res_2 <- spwb_spatial(example_subset, SpParamsMED, examplemeteo,
                      dates = dates, local_control = local_control,
                      keep_results = FALSE,
                      summary_function = summary.spwb, summary_arguments = list(freq="months"))
res_2
```

```
Simple feature collection with 5 features and 4 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.828998 ymin: 41.98704 xmax: 2.142714 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 5 × 5
             geometry id        state           result summary
          <POINT [°]> <chr>     <list>          <list> <list>
1 (2.130641 41.99872) 081015_A1 <spwbInpt [19]> <NULL> <dbl [1 × 19]>
2 (2.142714 41.99881) 081016_A1 <spwbInpt [19]> <NULL> <dbl [1 × 19]>
3 (1.828998 41.98704) 081018_A1 <spwbInpt [19]> <NULL> <dbl [1 × 19]>
4 (1.841068 41.98716) 081019_A1 <spwbInpt [19]> <NULL> <dbl [1 × 19]>
5 (1.853138 41.98728) 081020_A1 <spwbInpt [19]> <NULL> <dbl [1 × 19]>
```

Using summary functions (2)

We can access the simulation summary for the first stand using:

```
res_2$summary[[1]]
```

```
                PET Precipitation     Rain     Snow  NetRain Snowmelt
2001-01-01 31.14173      74.74949 58.09884 16.65065 40.91681 13.09301
           Infiltration InfiltrationExcess SaturationExcess Runoff DeepDrainage
2001-01-01     54.00981                  0                0      0     32.61347
           CapillarityRise Evapotranspiration Interception SoilEvaporation
2001-01-01               0           30.34032     17.18203        5.405063
           HerbTranspiration PlantExtraction Transpiration
2001-01-01                 0        7.753223      7.753223
           HydraulicRedistribution
2001-01-01              0.01133329
```

Summaries can be generated *a posteriori* for a given simulation, using function `simulation_summary()`, e.g.:

```
simulation_summary(res, summary_function = summary.spwb, freq="months")
```

```
Simple feature collection with 5 features and 2 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.828998 ymin: 41.98704 xmax: 2.142714 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 5 × 3
             geometry id        summary
          <POINT [°]> <chr>     <list>
1 (2.130641 41.99872) 081015_A1 <dbl [1 × 19]>
2 (2.142714 41.99881) 081016_A1 <dbl [1 × 19]>
3 (1.828998 41.98704) 081018_A1 <dbl [1 × 19]>
4 (1.841068 41.98716) 081019_A1 <dbl [1 × 19]>
5 (1.853138 41.98728) 081020_A1 <dbl [1 × 19]>
```

Tip

Learning how to define summary functions is a good investment when using **medfateland**.

Continuing a previous simulation

The result of a simulation includes an element `state`, which stores the state of soil and stand variables at the end of the simulation. This information can be used to perform a new simulation from the point where the first one ended.

In order to do so, we need to update the state variables in spatial object with their values at the end of the simulation, using function `update_landscape()`:

```
example_mod <- update_landscape(example_subset, res)
example_mod
```

```
Simple feature collection with 5 features and 8 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.828998 ymin: 41.98704 xmax: 2.142714 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 5 × 9
                 geom id        elevation slope aspect land_cover_type soil
          <POINT [°]> <chr>         <dbl> <dbl>  <dbl> <chr>           <list>
1 (2.130641 41.99872) 081015_A1       680  7.73  281.  wildland        <soil>
2 (2.142714 41.99881) 081016_A1       736 15.6   212.  wildland        <soil>
3 (1.828998 41.98704) 081018_A1       532 17.6   291.  wildland        <soil>
4 (1.841068 41.98716) 081019_A1       581  4.79  174.  wildland        <soil>
5 (1.853138 41.98728) 081020_A1       613  4.76   36.9 wildland        <soil>
# ℹ 2 more variables: forest <list>, state <list>
```

Note that `example_mod` contains a new column `state` with initialized inputs.

Finally, we can call again the simulation function for a new consecutive time period:

```
dates <- seq(as.Date("2001-02-01"), as.Date("2001-02-28"), by="day")
res_3 <- spwb_spatial(example_mod, SpParamsMED, examplemeteo,
                      dates = dates, local_control = local_control)
```

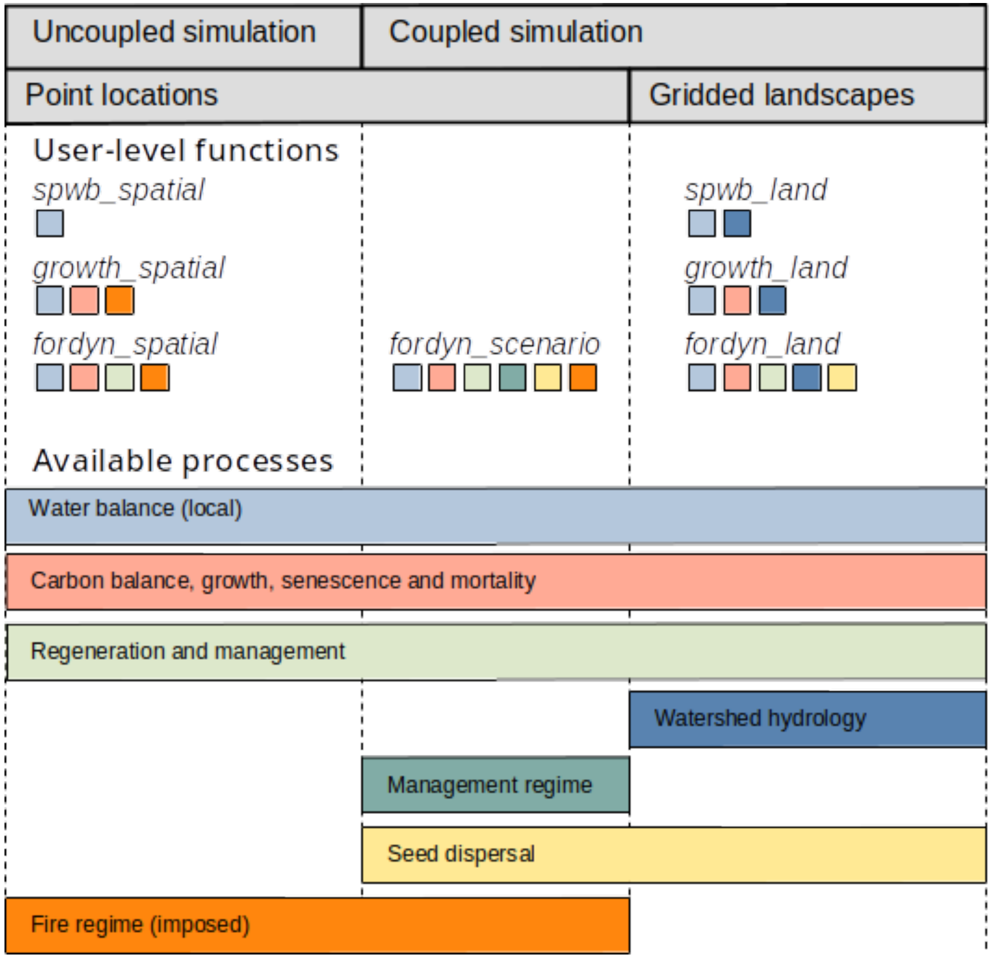
Important

Function `update_landscape()` will also modify column `soil`.

3. Regional management scenarios

Function `fordyn_scenario()`

- Function `fordyn_spatial()` allows running simulations of forest dynamics for a set of forest stands, possibly including forest management and stand-specific silviculture prescriptions.
- However, in `fordyn_spatial()` simulated stand dynamics are **uncoupled**.
- Function `fordyn_scenario()` allows simulating forest dynamics on a set of forest stands while evaluating a demand-based **management scenario**.
- Considering the management scenario leads to a relationship in the management actions on forest stands, hence **coupling simulations**.
- Running management scenarios is a complex task, we will cover all details in this tutorial.



Management units and prescriptions (1)

Management scenarios require classifying forest stands into **management units**. Each management unit can be interpreted as a set of stands that are managed following the same prescriptions.

Management units can be arbitrarily defined, but here we will define them on the basis of **dominant tree species**.

The following code allows determining the dominant tree species in each of the 5 forest stands:

```
example_subset$dominant_tree_species <- sapply(example_subset$forest,
                                                stand_dominantTreeSpecies, SpParamsMED)
example_subset$dominant_tree_species
```

```
[1] "Pinus sylvestris"  "Pinus sylvestris"  "Quercus pubescens"
[4] "Quercus ilex"      "Quercus faginea"
```

The package includes a table with **default prescription parameters** for a set of species, whose columns are management parameters:

```
names(defaultPrescriptionsBySpecies)
```

```
 [1] "Name"                "SpIndex"             "type"
 [4] "targetTreeSpecies"   "thinning"            "thinningMetric"
 [7] "thinningThreshold"   "thinningPerc"        "minThinningInterval"
[10] "yearsSinceThinning"  "finalMeanDBH"        "finalPerc"
[13] "finalPreviousStage"  "finalYearsBetweenCuts" "finalYearsToCut"
[16] "plantingSpecies"     "plantingDBH"         "plantingHeight"
[19] "plantingDensity"     "understoryMaximumCover"
```

whereas the rows correspond to species or species groups, whose names are:

```
head(defaultPrescriptionsBySpecies$Name)
```

```
[1] "Abies/Picea/Pseudotsuga spp." "Betula/Acer spp."
[3] "Castanea sativa"              "Eucalyptus spp."
[5] "Fagus sylvatica"              "Fraxinus spp."
```

Management units and prescriptions (2)

To specify the management unit for stands, we first define a column management_unit with missing values:

```
example_subset$management_unit <- NA
```

and then assign the corresponding row number of `defaultPrescriptionsBySpecies` for stands dominated by each species where management is to be conducted:

```
example_subset$management_unit[example_subset$dominant_tree_species=="Pinus sylvestris"] <- 14
example_subset$management_unit[example_subset$dominant_tree_species=="Quercus ilex"] <- 19
example_subset$management_unit[example_subset$dominant_tree_species=="Quercus pubescens"] <- 23
example_subset[,c("id", "dominant_tree_species", "management_unit")]
```

```
Simple feature collection with 5 features and 3 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.828998 ymin: 41.98704 xmax: 2.142714 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 5 × 4
  id        dominant_tree_species management_unit                geom
  <chr>     <chr>                           <dbl>         <POINT [°]>
1 081015_A1 Pinus sylvestris                   14 (2.130641 41.99872)
2 081016_A1 Pinus sylvestris                   14 (2.142714 41.99881)
3 081018_A1 Quercus pubescens                  23 (1.828998 41.98704)
4 081019_A1 Quercus ilex                       19 (1.841068 41.98716)
5 081020_A1 Quercus faginea                    NA (1.853138 41.98728)
```

In this example stands dominated by *Quercus faginea* are not harvested.

Management scenarios and represented area

Management scenarios

Management scenarios are defined using function `create_management_scenario()` [1].

Demand-based management scenarios require specifying the demand in annual volume [2].

```r
scen <- create_management_scenario(units = defaultPrescriptionsBySpecies,
                                   annual_demand_by_species = c("Quercus ilex/Quercus pubescens" = 1300,
                                                                "Pinus sylvestris" = 500))
```

Note that in this case the timber obtained from *Q. ilex* or *Q. pubescens* will be subtracted from the same annual demand.

We can check the kind of management scenario using:

```r
scen$scenario_type
```

```
[1] "input_demand"
```

Represented area

Finally, it is necessary to specify the area (in ha) that each forest stand represents, because all timber volumes are defined at the stand level in units of **m3/ha**, whereas the demand is in units of **m3/yr**.

In our example, we will assume a constant area of 100 ha for all stands:

```r
example_subset$represented_area_ha <- 100
```

1. Three different kinds of scenarios are allowed in `create_management_scenario()`, two of them being demand-based.
2. The fact that demand is specified in volume entails that simulations need to be able to estimate timber volume for any given tree. In practice, this requ… specifying a **volume_function**. See `fordyn_scenario` for details.

Launching simulations

We are now ready to launch the simulation of the management scenario using a call to function `fordyn_scenario()`.

```
fs <- fordyn_scenario(example_subset, SpParamsMED, meteo = examplemeteo,
                      management_scenario = scen,
                      parallelize = TRUE)
```

Tip

We will often set `parallelize = TRUE` to speed-up calculations (`fordyn_scenario()` makes internall calls to `fordyn_spatial()` for each simulated year).

Function `fordyn_scenario()` returns a list whose elements are:

```
names(fs)
```

```
[1] "result_sf"                "result_volumes"           "result_volumes_spp"
[4] "result_volumes_demand"    "next_demand"              "next_sf"
```

Stand-level results are available in element `result_sf`, whose columns should be easy to interpret if you have experience with `fordyn()`:

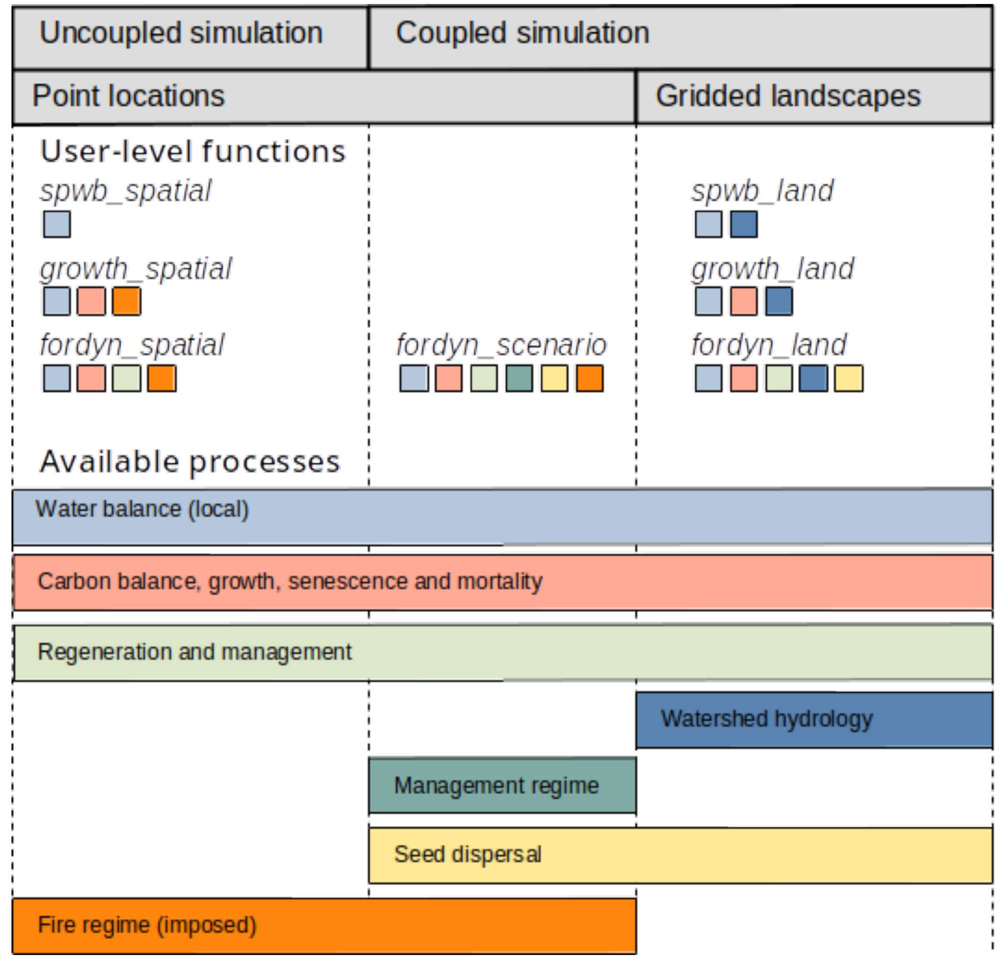
```
fs$result_sf
```

```
Simple feature collection with 5 features and 8 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.828998 ymin: 41.98704 xmax: 2.142714 ymax: 41.99881
Geodetic CRS:  WGS 84
# A tibble: 5 × 9
             geometry id        tree_table         shrub_table dead_tree_table
          <POINT [°]> <chr>     <list>             <list>      <list>
1 (2.130641 41.99872) 081015_A1 <tibble [48 × 11]> <tibble>    <tibble>
2 (2.142714 41.99881) 081016_A1 <tibble [30 × 11]> <tibble>    <tibble>
3 (1.828998 41.98704) 081018_A1 <tibble [2 × 11]>  <tibble>    <tibble [1 × 14]>
4 (1.841068 41.98716) 081019_A1 <tibble [2 × 11]>  <tibble>    <tibble [1 × 14]>
5 (1.853138 41.98728) 081020_A1 <tibble [4 × 11]>  <tibble>    <tibble [2 × 14]>
# ℹ 4 more variables: dead_shrub_table <list>, cut_tree_table <list>,
#   cut_shrub_table <list>, summary <list>
```

4. Watershed-level simulations

Watershed-level simulation functions

- Package **medfateland** allows conducting simulations of forest function and dynamics on a set of forest stands while including **lateral water transfer processes**.
- Similar to other models such as TETIS [1], three lateral flows are considered between adjacent cells:
    - Overland surface flows from upper elevation cells.
    - Lateral saturated soil flows (i.e. interflow) between adjacent cells.
    - Lateral groundwater flow (i.e. baseflow) between adjacent cells.
- Following the nested models of **medfate**, **medfateland** offers functions `spwb_land()`, `growth_land()` and `fordyn_land()` for watershed-level simulations [2].
- Here we will cover the basics of watershed simulations only.



1. Francés et al. (2007) Journal of Hydrology, 332, 226–240.
2. There exist `spwb_land_day()` and `growth_land_day()` for single-day simulations, paralleling `spwb_spatial_day()` and `growth_spatial_day()`.

Model inputs (1)

Spatial structures

To perform simulations on a gridded landscape we require both an `sf` object:

```
example_watershed
```

```
Simple feature collection with 66 features and 14 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 401430 ymin: 4671870 xmax: 402830 ymax: 4672570
Projected CRS: WGS 84 / UTM zone 31N
# A tibble: 66 × 15
           geometry    id elevation slope aspect land_cover_type
 *      <POINT [m]> <int>     <dbl> <dbl>  <dbl> <chr>
 1 (402630 4672570)     1      1162 11.3   79.2  wildland
 2 (402330 4672470)     2      1214 12.4   98.7  agriculture
 3 (402430 4672470)     3      1197 10.4  102.   wildland
 4 (402530 4672470)     4      1180  8.12  83.3  wildland
 5 (402630 4672470)     5      1164 13.9   96.8  wildland
 6 (402730 4672470)     6      1146 11.2    8.47 agriculture
 7 (402830 4672470)     7      1153  9.26 356.   agriculture
 8 (402230 4672370)     8      1237 14.5   75.1  wildland
 9 (402330 4672370)     9      1213 13.2   78.7  wildland
10 (402430 4672370)    10      1198  8.56  75.6  agriculture
# ℹ 56 more rows
# ℹ 9 more variables: forest <list>, soil <list>, state <list>,
#   depth_to_bedrock <dbl>, bedrock_conductivity <dbl>, bedrock_porosity <dbl>,
#   snowpack <dbl>, aquifer <dbl>, crop_factor <dbl>
```

and a `SpatRast` topology with the same coordinate reference system:

```
r <-terra::rast(xmin = 401380, ymin = 4671820, xmax = 402880, ymax = 4672620,
                nrow = 8, ncol = 15, crs = "epsg:32631")
r
```

```
class       : SpatRaster
dimensions  : 8, 15, 1  (nrow, ncol, nlyr)
resolution  : 100, 100  (x, y)
extent      : 401380, 402880, 4671820, 4672620  (xmin, xmax, ymin, ymax)
coord. ref. : WGS 84 / UTM zone 31N (EPSG:32631)
```

Model inputs (2)

Land cover type

Simulations over watersheds normally include different land cover types. These are described in column land_cover_type:

```
table(example_watershed$land_cover_type)
```

```
agriculture        rock    wildland 
         17           1          48 
```

Aquifer and snowpack

Columns `aquifer` and `snowpack` are used as state variables to store the water content in the aquifer and snowpack, respectively.

Crop factors

Since the landscape contains agricultural lands, we need to define crop factors, which will determine transpiration flow as a proportion of potential evapotranspiration:

```
example_watershed$crop_factor = NA
example_watershed$crop_factor[example_watershed$land_cover_type=="agriculture"] = 0.75
```

Watershed control options

Analogously to local-scale simulations with **medfate**, watershed simulations have overall control parameters. Notably, the user needs to decide which sub-model will be used for lateral water transfer processes (at present, only `"tetis"` is available):

```
ws_control <- default_watershed_control("tetis")
```

Launching simulations

As with other functions, we may specify a simulation period (subsetting the weather input):

```r
dates <- seq(as.Date("2001-01-01"), as.Date("2001-01-31"), by="day")
```

When calling the simulation function, we must provide the raster topology, the input `sf` object, among other inputs:

```r
res_ws <- spwb_land(r, example_watershed, SpParamsMED, examplemeteo,
                    dates = dates,
                    local_control = local_control,
                    watershed_control = ws_control,
                    progress = FALSE)
```

Important

Remember, watershed simulations require both control parameters for **local processes** and control parameter for **watershed processes**.

Simulation output (1)

As usual, the output of `spwb_land()` is a named list.

```
names(res_ws)
```

```
[1] "watershed_control"      "sf"                        "watershed_balance"
[4] "watershed_soil_balance" "outlet_export_m3s"
```

Where `sf` is analogous to those of functions `*_spatial()`, containing final state of cells as well as cell-level summaries:

```
res_ws$sf
```

```
Simple feature collection with 66 features and 6 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 401430 ymin: 4671870 xmax: 402830 ymax: 4672570
Projected CRS: WGS 84 / UTM zone 31N
# A tibble: 66 × 7
           geometry state           aquifer snowpack summary  result outlet
        <POINT [m]> <list>            <dbl>    <dbl> <list>   <list> <lgl>
 1 (402630 4672570) <spwbInpt [19]> 127.        3.56 <dbl[…]> <NULL> FALSE
 2 (402330 4672470) <aspwbInp [4]>    0.362     3.56 <dbl[…]> <NULL> FALSE
 3 (402430 4672470) <spwbInpt [19]>   2.29      3.56 <dbl[…]> <NULL> FALSE
 4 (402530 4672470) <spwbInpt [19]>   9.62      2.56 <dbl[…]> <NULL> FALSE
 5 (402630 4672470) <spwbInpt [19]> 149.        2.57 <dbl[…]> <NULL> FALSE
 6 (402730 4672470) <aspwbInp [4]>  874.        3.56 <dbl[…]> <NULL> TRUE
 7 (402830 4672470) <aspwbInp [4]>  412.        3.56 <dbl[…]> <NULL> FALSE
 8 (402230 4672370) <spwbInpt [19]>   0.344     2.84 <dbl[…]> <NULL> FALSE
 9 (402330 4672370) <spwbInpt [19]>   1.70      2.97 <dbl[…]> <NULL> FALSE
10 (402430 4672370) <aspwbInp [4]>    6.33      3.56 <dbl[…]> <NULL> FALSE
# ℹ 56 more rows
```

Simulation output (2)

In addition, element `watershed_balance` contains daily values of the water balance **at the watershed level**:

```r
head(res_ws$watershed_balance)
```

```
       dates Precipitation      Rain Snow Snowmelt Interception   NetRain
1 2001-01-01      4.869109  4.869109    0        0    0.7900101  4.079099
2 2001-01-02      2.498292  2.498292    0        0    0.6919287  1.806363
3 2001-01-03      0.000000  0.000000    0        0    0.0000000  0.000000
4 2001-01-04      5.796973  5.796973    0        0    0.7855456  5.011427
5 2001-01-05      1.884401  1.884401    0        0    0.5571451  1.327256
6 2001-01-06     13.359801 13.359801    0        0    0.8937189 12.466082
  Infiltration InfiltrationExcess SaturationExcess  CellRunon CellRunoff
1     4.079099         0.00000000         0.000000 0.00000000   0.000000
2     1.806363         0.00000000         0.000000 0.00000000   0.000000
3     0.000000         0.00000000         0.000000 0.00000000   0.000000
4     5.011427         0.00000000         1.150467 0.00000000   1.150467
5     1.327256         0.00000000         9.350710 0.00000000   9.350710
6    12.466082         0.05090607        16.077935 0.05090607  16.128841
  DeepDrainage CapillarityRise DeepAquiferLoss SoilEvaporation Transpiration
1     2.938050               0               0       0.3867130     0.2957627
2     1.639472               0               0       0.4679914     0.5244844
3     1.598464               0               0       0.3525398     0.4407831
4     2.353805               0               0       0.1848718     0.1967883
5     2.172521               0               0       0.3300812     0.5252368
6     2.311613               0               0       0.1924125     0.3710988
  HerbTranspiration InterflowBalance BaseflowBalance AquiferExfiltration
1                 0     0.000000e+00    0.000000e+00                   0
2                 0    -1.703512e-16    3.111989e-17                   0
3                 0     1.058497e-15    1.261617e-17                   0
```

Advanced topics

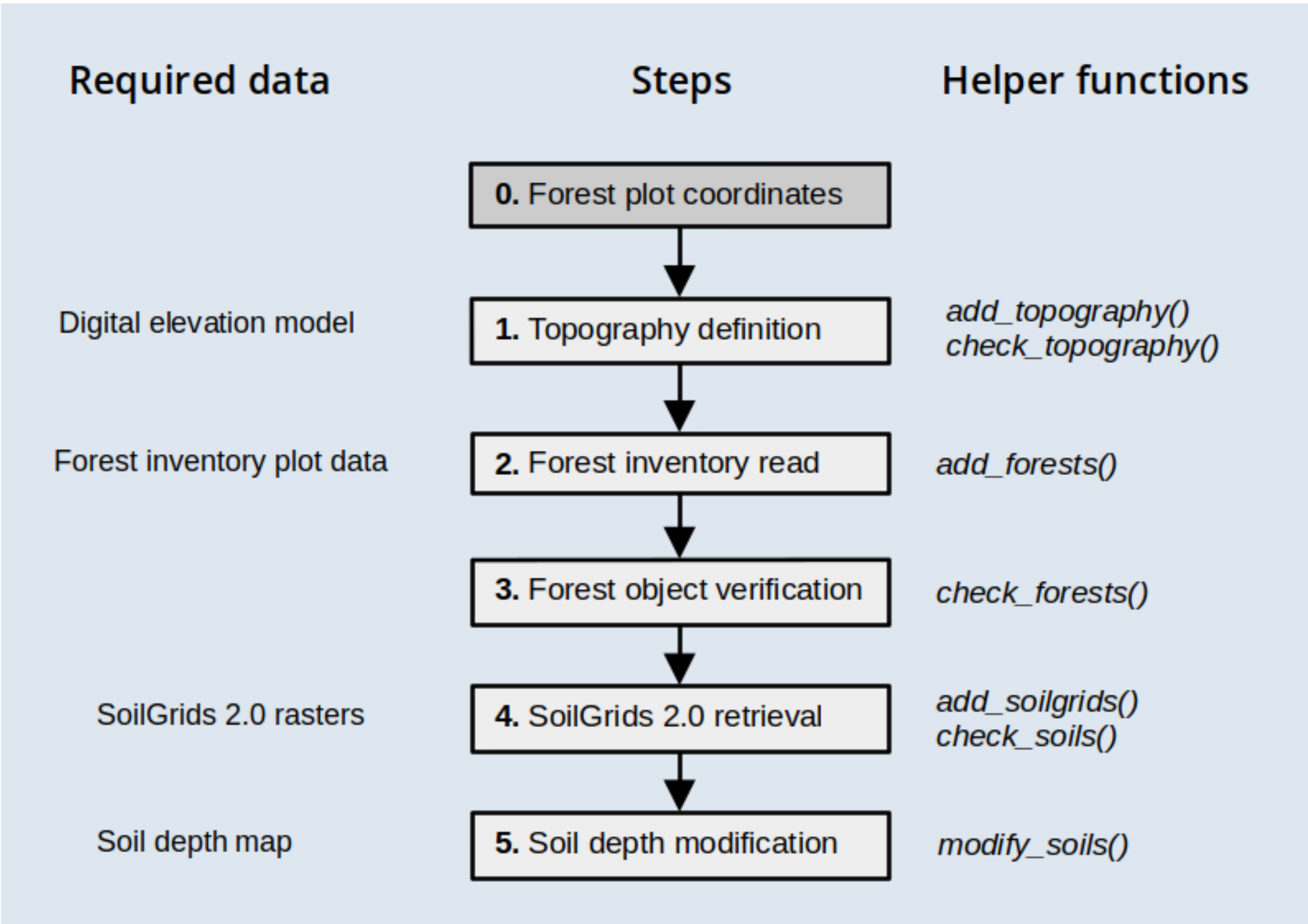
The following table summarises a set of advanced topics for watershed simulations.

Topic	Description
Burn-in	Watershed simulations always require **burn-in** periods where soil and aquifer levels reach equilibrium values. This is facilitated via function `update_landscape()`.
Calibration	Watershed simulations will normally require **calibration** of watershed-level control parameters.
Weather resolution	Weather interpolation can have a **coarser resolution** than the watershed grid (see `weather_aggregation_factor` in `?default_watershed_control`).
Parallelization	At present, **parallelization is not recommended** for watershed simulations.
Result cells	Whereas by default only water balance summaries are produced for individual cell, it is possible to specify full **medfate** results on target cells, via a column called `result_cell`.
Local control	Analogously to weather forcing, it is possible to specify **spatial variation in the control parameters** for local processes (e.g. Sperry or Sureau only in targetted cells), via a column called `local_control`.

5. Creating spatial inputs I: forest inventory plots

Input preparation workflow

The functions introduced in this section are meant to be executed sequentially to progressively add spatial information to an initial `sf` object of plot coordinates:



… but users are free to use them in the most convenient way.

Target locations: Poblet again!

We begin by defining an `sf` object with the target locations and forest stand identifiers (column id):

```r
cc <- rbind(c(1.0215, 41.3432),
            c(1.0219, 41.3443),
            c(1.0219, 41.3443))
d <- data.frame(lon = cc[,1], lat = cc[,2],
                id = c("POBL_CTL", "POBL_THI_BEF", "POBL_THI_AFT"))
x <- sf::st_as_sf(d, coords = c("lon", "lat"), crs = 4326)
x
```

```
Simple feature collection with 3 features and 1 field
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.0215 ymin: 41.3432 xmax: 1.0219 ymax: 41.3443
Geodetic CRS:  WGS 84
            id               geometry
1     POBL_CTL POINT (1.0215 41.3432)
2 POBL_THI_BEF POINT (1.0219 41.3443)
3 POBL_THI_AFT POINT (1.0219 41.3443)
```

where `POBL_CTL` is the control forest plot, `POBL_THI_BEF` is the managed plot before thinning and `POBL_THI_AFT` is the managed plot after thinning.

Topography

You should have access to a **Digital Elevation Model** (DEM) at a desired resolution.

Here we will use a DEM raster for Catalonia at 30 m resolution, which we load using package **terra**:

```
dataset_path <- "~/OneDrive/EMF_datasets/"
dem <- terra::rast(paste0(dataset_path,"Topography/Products/Catalunya/MET30m_ETRS89_UTM31_ICGC.tif"))
dem
```

```
class       : SpatRaster
dimensions  : 9282, 9391, 1  (nrow, ncol, nlyr)
resolution  : 30, 30  (x, y)
extent      : 258097.5, 539827.5, 4485488, 4763948  (xmin, xmax, ymin, ymax)
coord. ref. : ETRS89 / UTM zone 31N (EPSG:25831)
source      : MET30m_ETRS89_UTM31_ICGC.tif
name        : met15v20as0f0118Bmr1r050
min value   :                   -7.120
max value   :                 3133.625
```

Having these inputs, we can use function add_topography() to add topographic features to our starting `sf`:

```
y_1 <- add_topography(x, dem = dem, progress = FALSE)
```

```
|---------|---------|---------|---------|
=========================================


|---------|---------|---------|---------|
=========================================
```

We can check that there are no missing values in topographic features:

```
check_topography(y_1)
```

```
✔ No missing values in topography.
```

Forest inventory data (1)

The next step is to define forest objects for our simulations.

While at this point you would read your own data from a file or data base, here we simply load the Poblet data from **medfate**:

```
data(poblet_trees)
head(poblet_trees)
```

```
  Plot.Code Indv.Ref             Species Diameter.cm
1  POBL_CTL        1 Acer monspessulanum         7.6
2  POBL_CTL        2      Arbutus unedo          7.5
3  POBL_CTL        3      Arbutus unedo          7.5
4  POBL_CTL        4      Arbutus unedo          7.5
5  POBL_CTL        5      Arbutus unedo          7.5
6  POBL_CTL        6      Arbutus unedo          7.5
```

We learned in the first exercise to use function `forest_mapTreeData()` from package **medfate**. Here we can speed up the process for all plots defining the mapping and calling function `add_forests()`:

```
mapping <- c("id" = "Plot.Code", "Species.name" = "Species", "DBH" = "Diameter.cm")
y_2 <- add_forests(y_1, tree_table = poblet_trees, tree_mapping = mapping,
                   SpParams = SpParamsMED)
```

```
Warning in forest_mapTreeTable(x = tree_id, mapping_x = tree_mapping, SpParams
= SpParams): Taxon names that were not matched: Quercus humilis.
Warning in forest_mapTreeTable(x = tree_id, mapping_x = tree_mapping, SpParams
= SpParams): Taxon names that were not matched: Quercus humilis.
```

Two warnings were raised, informing us that *Quercus humilis* (downy oak) was not matched to any species name in `SpParamsMED` (that is the reason why we provided it as an input).

Forest inventory data (2)

We correct the scientific name for downy oak and repeat to avoid losing tree records:

```
poblet_trees$Species[poblet_trees$Species=="Quercus humilis"] <- "Quercus pubescens"
y_2 <- add_forests(y_1, tree_table = poblet_trees, tree_mapping = mapping,
                   SpParams = SpParamsMED)
```

We can use function `check_forests()` to verify that there is no missing data:

```
check_forests(y_2)
```

```
✔ No wildland locations with NULL values in column 'forest'.
✔ All objects in column 'forest' have the right class.
! Missing tree height values detected for 28 (100%) in 3 wildland locations (100%).
```

Like in the first exercise, we should provide plot size and tree height, which we do as follows:

```
poblet_trees$PlotSurface <- 706.86
poblet_trees$Height.cm <- 100 * 1.806*poblet_trees$Diameter.cm^0.518
mapping <- c(mapping, "plot.size" = "PlotSurface", "Height" = "Height.cm")
y_2 <- add_forests(y_1, tree_table = poblet_trees, tree_mapping = mapping, SpParams = SpParamsMED)
```

And check again…

```
check_forests(y_2)
```

```
✔ No wildland locations with NULL values in column 'forest'.
✔ All objects in column 'forest' have the right class.
✔ No missing/wrong values detected in key tree/shrub attributes of 'forest' objects.
```

Soil data (1)

Soil information is most usually lacking for the target locations.

Here we assume regional maps are not available, so that we resort to SoilGrids 2.0, a global product provided at 250 m resolution [1].

Function `add_soilgrids()` can perform queries using the REST API of SoilGrids, but this becomes problematic for multiple sites.

Hence, we recommend downloading SoilGrid rasters for the target region and storing them in a particular format, so that function `add_soilgrids()` can read them (check the details of the function documentation).

Assuming we have the raster data, parsing soil grids is straightforward:

```
soilgrids_path = paste0(dataset_path,"Soils/Sources/Global/SoilGrids/Spain/")
y_3 <- add_soilgrids(y_2, soilgrids_path = soilgrids_path, progress = FALSE)
```

And the result has an extra column `soil`:

```
y_3
```

```
Simple feature collection with 3 features and 6 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.0215 ymin: 41.3432 xmax: 1.0219 ymax: 41.3443
Geodetic CRS:  WGS 84
# A tibble: 3 × 7
  id                    geometry elevation slope aspect forest       soil
  <chr>              <POINT [°]>     <dbl> <dbl>  <dbl> <list>       <list>
1 POBL_CTL     (1.0215 41.3432)      853.   30.1   76.0 <forest [5]> <df [6 × 7]>
2 POBL_THI_BEF (1.0219 41.3443)      814.   29.3   40.3 <forest [5]> <df [6 × 7]>
3 POBL_THI_AFT (1.0219 41.3443)      814.   29.3   40.3 <forest [5]> <df [6 × 7]>
```

1. Hengl et al. (2017) PLoS ONE 12, e0169748; Poggio et al. (2021) Soil, 7, 217-240

Soil data (2)

We can use function `check_soils()` to detect whether there are missing values:

```
check_soils(y_3)
```

```
✔ No wildland/agriculture locations with NULL values in column 'soil'.
✔ No missing values detected in key soil attributes.
```

Warning

- SoilGrids tends to overestimate soil water holding capacity due to the underestimation of rock content
- Other products exist, and function `modify_soils()` can help using them. See vignette PreparingInputs
- Nevertheless, uncertainty in soil depth and rock content is always high!

Procedure using package forestables (1)

R package **forestables** allows reading and harmonizing national forest inventory data from the FIA (USA forest inventory), FFI (France forest inventory) and IFN (Spain forest inventory).

Using **forestables** simplifies the data preparation for large areas.

Here we will use a subset of IFN4 data that is provided as example output of **forestables**:

```r
library(forestables)
ifn4_example <- ifn_output_example |>
  dplyr::filter(version == "ifn4")
ifn4_example
```

```
# A tibble: 1,686 × 24
   id_unique_code    year plot  coordx coordy coord_sys   crs  elev aspect slope
   <chr>            <int> <chr>  <dbl>  <dbl> <chr>     <dbl> <dbl>  <dbl> <dbl>
 1 08_0001_NN_A1_A1  2015 0001  401922 4.68e6 ED50      23031    NA   202.    40
 2 08_0002_NN_A1_A1  2014 0002  399895 4.68e6 ED50      23031    NA   342     40
 3 08_0003_NN_A1_A1  2015 0003  400898 4.68e6 ED50      23031    NA    99     40
 4 08_0004_NN_A1_A1  2014 0004  401903 4.68e6 ED50      23031    NA   292.    40
 5 08_0005_NN_A1_A1  2014 0005  399917 4.68e6 ED50      23031    NA    99     40
 6 08_0006_NN_A1_A1  2014 0006  396931 4.68e6 ED50      23031    NA   351     40
 7 08_0009_xx_xx_A4  2016 0009  401899 4.68e6 ED50      23031    NA    90     40
 8 08_0014_NN_A1_A1  2015 0014  397927 4.68e6 ED50      23031    NA   234     40
 9 08_0016_xx_xx_A4  2014 0016  392906 4.68e6 ED50      23031    NA   346.    40
10 08_0020_NN_A1_A1  2015 0020  397842 4.68e6 ED50      23031    NA    36     40
# ℹ 1,676 more rows
# ℹ 14 more variables: country <chr>, version <chr>, class <chr>,
#   subclass <chr>, province_code <chr>, province_name_original <chr>,
#   ca_name_original <chr>, sheet_ntm <chr>, huso <dbl>, slope_mean <chr>,
#   type <int>, tree <list>, understory <list>, regen <list>
```

Procedure using package forestables (2)

Function `parse_forestable()` allows parsing a data frame (or sf) issued from package **forestables** and reshaping it for **medfateland**.

```
y_1 <- parse_forestable(ifn4_example[1:10,])
```

The function parses plot *ids*, *coordinates*, and *topography* (according to IFN data!). Importantly, it defines a new column called `forest` and parses tree and shrub data into it.

```
y_1
```

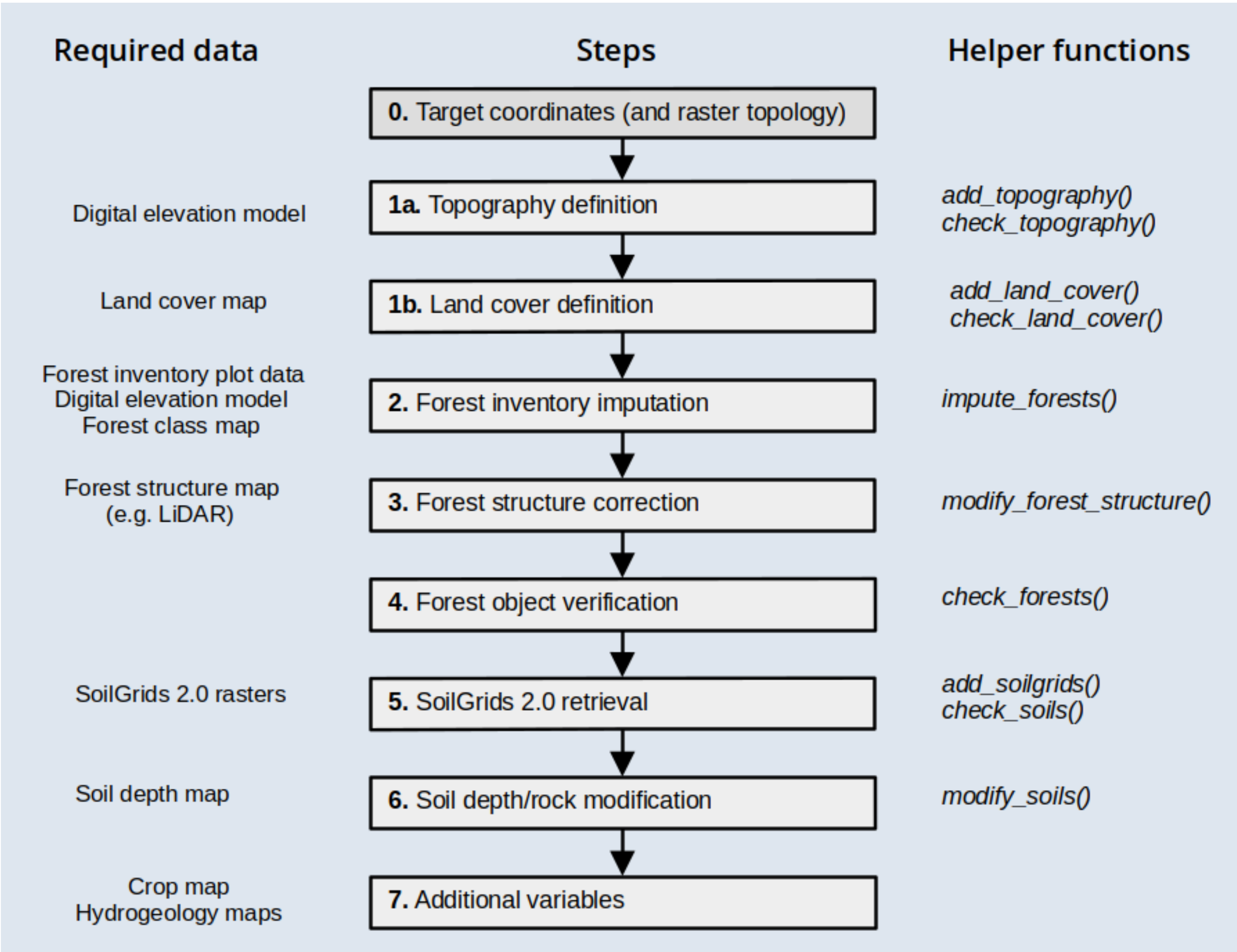
```
Simple feature collection with 10 features and 10 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 1.700602 ymin: 42.25217 xmax: 1.809254 ymax: 42.29746
Geodetic CRS:  WGS 84
# A tibble: 10 × 11
   id                         geometry  year plot  country version regen
   <chr>                   <POINT [°]> <int> <chr> <chr>   <chr>   <list>
 1 08_0001_NN_A1… (1.809047 42.29746)  2015 0001  ES      ifn4    <tibble>
 2 08_0002_NN_A1… (1.784613 42.28934)  2014 0002  ES      ifn4    <tibble>
 3 08_0003_NN_A1… (1.796776 42.28951)  2015 0003  ES      ifn4    <tibble>
 4 08_0004_NN_A1… (1.808972 42.28922)  2014 0004  ES      ifn4    <tibble>
 5 08_0005_NN_A1… (1.785068 42.27957)  2014 0005  ES      ifn4    <tibble>
 6 08_0006_NN_A1… (1.749024 42.27098)  2014 0006  ES      ifn4    <tibble>
 7 08_0009_xx_xx…  (1.809254 42.2717)  2016 0009  ES      ifn4    <tibble>
 8 08_0014_NN_A1… (1.761286 42.26156)  2015 0014  ES      ifn4    <tibble>
 9 08_0016_xx_xx… (1.700602 42.25217)  2014 0016  ES      ifn4    <tibble>
10 08_0020_NN_A1… (1.760414 42.25344)  2015 0020  ES      ifn4    <tibble>
# ℹ 4 more variables: elevation <dbl>, slope <dbl>, aspect <dbl>, forest <list>
```

The remaining steps are similar to the general procedure, with calls to `check_forests()`, `add_soilgrids()`, etc.

6. Creating spatial inputs II: continuous landscapes

Input preparation workflow for arbitrary locations

When target locations are not sampled forest inventory plots, as in continuous landscapes, the preparation workflow changes slightly:



The main difference lies in the need to conduct **imputation** of forest structure and composition.

Forest imputation and correction

Warning

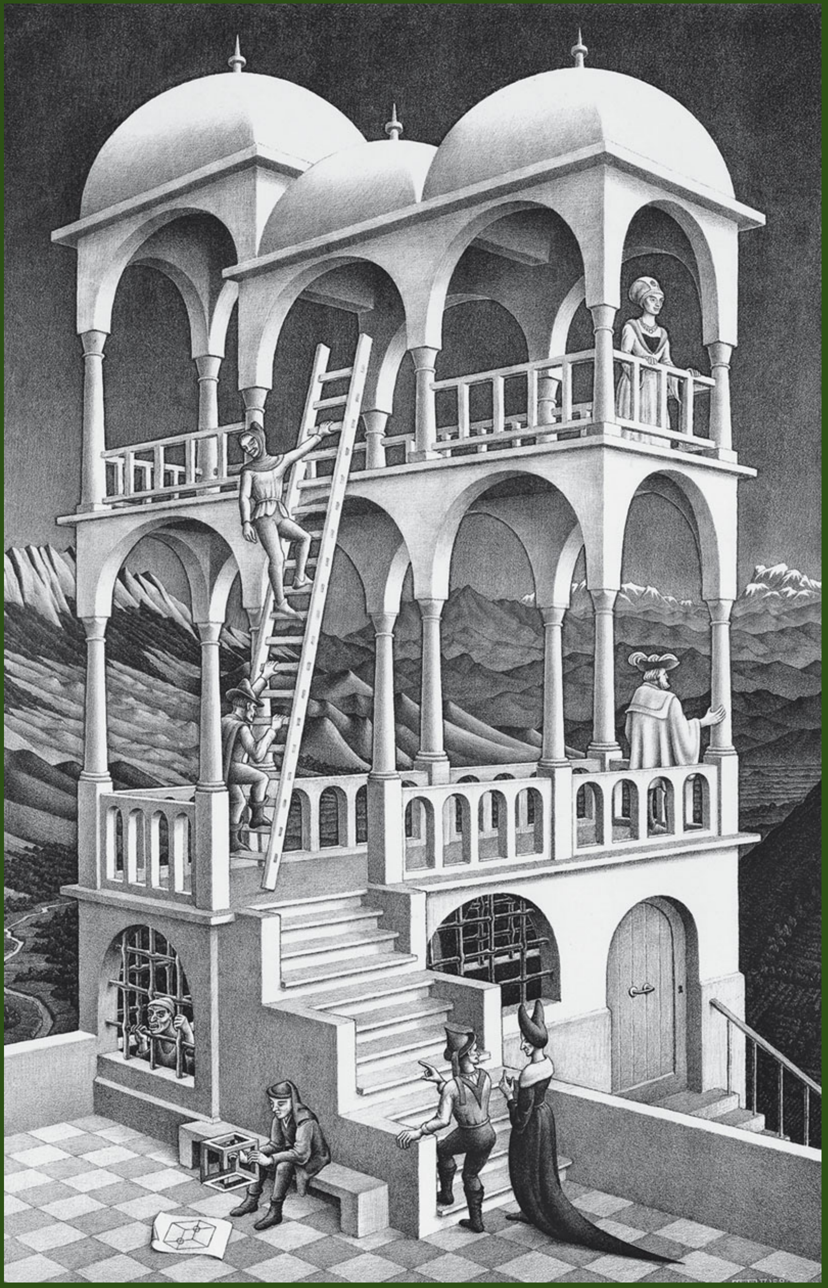
Forest imputation can be a difficult task!

Function `impute_forests()` performs a simple imputation of forest inventory plots on the basis of a **forest map** and **topographic position**.

Whereas forest composition is provided by the **forest map**, the forest structure resulting from `impute_forests()` should be corrected!

If one has access to structure mapes (e.g. from LiDAR data), this second step can be done using function `modify_forest_structure()`.

The whole procedure is illustrated in vignette PreparationInputs_II (see also Exercise 4a).



M.C. Escher - Belvedere, 1958