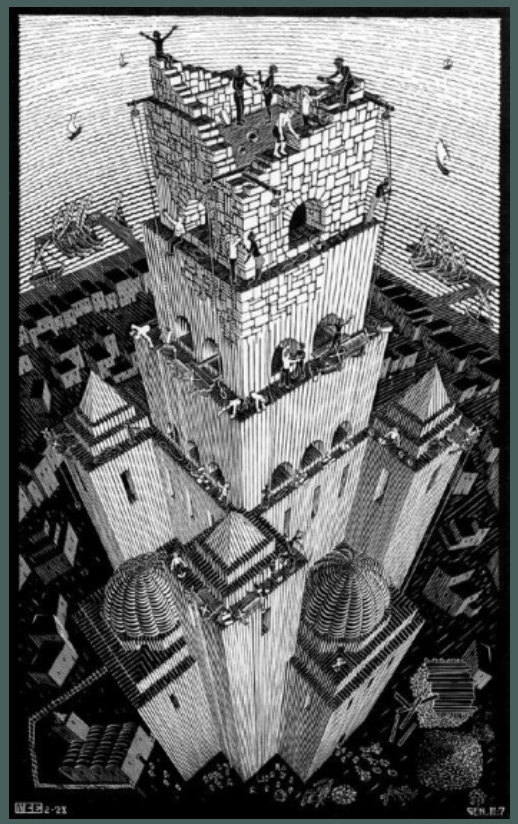
1.4 - Model inputs (exercise)

Miquel De Cáceres, Victor Granda, Aitor Ameztegui

Ecosystem Modelling Facility

2022-06-13



Exercise setting

Objectives

1. Build `forest` objects from a tree data frame of forest inventory data
2. Retrieve soil physical properties from SoilGrids
3. Interpolate daily weather on the plot location

Exercise setting

Objectives

1. Build `forest` objects from a tree data frame of forest inventory data
2. Retrieve soil physical properties from SoilGrids
3. Interpolate daily weather on the plot location

Data

Package **medfateutils** includes a data frame (`poblet_trees`), corresponding to forest inventory data in a dense holm oak forest.

- *Location*: Poblet (Catalonia, Spain); long/lat: 1.0219º, 41.3443º
- *Topography*: elevation = 850 m, slope = 15.1º, aspect = 15º
- *Plot*: Circular plot of 15-m radius
- *Tree data*: Stem diameter measurements on two plots: *control* and *managed*.

Exercise setting

Objectives

1. Build `forest` objects from a tree data frame of forest inventory data
2. Retrieve soil physical properties from SoilGrids
3. Interpolate daily weather on the plot location

Data

Package **medfateutils** includes a data frame (`poblet_trees`), corresponding to forest inventory data in a dense holm oak forest.

- *Location*: Poblet (Catalonia, Spain); long/lat: 1.0219º, 41.3443º
- *Topography*: elevation = 850 m, slope = 15.1º, aspect = 15º
- *Plot*: Circular plot of 15-m radius
- *Tree data*: Stem diameter measurements on two plots: *control* and *managed*.

As a result of the abandonment of former coppicing in the area, there is a high density of stems per individual in the control plot.

Exercise setting

Objectives

1. Build `forest` objects from a tree data frame of forest inventory data
2. Retrieve soil physical properties from SoilGrids
3. Interpolate daily weather on the plot location

Data

Package **medfateutils** includes a data frame (`poblet_trees`), corresponding to forest inventory data in a dense holm oak forest.

- *Location*: Poblet (Catalonia, Spain); long/lat: 1.0219º, 41.3443º
- *Topography*: elevation = 850 m, slope = 15.1º, aspect = 15º
- *Plot*: Circular plot of 15-m radius
- *Tree data*: Stem diameter measurements on two plots: *control* and *managed*.

As a result of the abandonment of former coppicing in the area, there is a high density of stems per individual in the control plot.

The management involved a reduction of the number of stems per individual (*sucker cutback* or *selecció de tanys*).

Exercise solution

Step 1. Loading packages

You may need to install **medfateutils** from GitHub:

```
devtools::install_github("emf-creaf/medfateutils")
```

Exercise solution

Step 1. Loading packages

You may need to install **medfateutils** from GitHub:

```
devtools::install_github("emf-creaf/medfateutils")
```

We begin by loading packages **medfate**, **medfateutils** and **meteoland**

```
library(medfate)
library(medfateutils)
library(meteoland)
```

Exercise solution

Step 1. Loading packages

You may need to install **medfateutils** from GitHub:

```
devtools::install_github("emf-creaf/medfateutils")
```

We begin by loading packages **medfate**, **medfateutils** and **meteoland**

```
library(medfate)
library(medfateutils)
library(meteoland)
```

Step 2. Load and inspect Poblet data

Normally, tree data would be in a **.csv** or **.xlsx** file. Here, we simply load the tree data from Poblet included in the package:

```
data("poblet_trees")
```

Exercise solution

Step 2. Load and inspect Poblet data

We can inspect its content, for example using:

```
summary(poblet_trees)
```

```
##   Plot.Code           Indv.Ref       Species           Diameter.cm    
##  Length:717         Min.   :  1.0   Length:717         Min.   : 7.50  
##  Class :character   1st Qu.: 45.0   Class :character   1st Qu.: 9.10  
##  Mode  :character   Median : 97.0   Mode  :character   Median :11.10  
##                     Mean   :103.4                      Mean   :11.62  
##                     3rd Qu.:156.0                      3rd Qu.:13.40  
##                     Max.   :261.0                      Max.   :26.00
```

Exercise solution

Step 2. Load and inspect Poblet data

We can inspect its content, for example using:

```
summary(poblet_trees)
```

```
##   Plot.Code            Indv.Ref       Species           Diameter.cm
##  Length:717         Min.   :  1.0   Length:717         Min.   : 7.50
##  Class :character   1st Qu.: 45.0   Class :character   1st Qu.: 9.10
##  Mode  :character   Median : 97.0   Mode  :character   Median :11.10
##                     Mean   :103.4                      Mean   :11.62
##                     3rd Qu.:156.0                      3rd Qu.:13.40
##                     Max.   :261.0                      Max.   :26.00
```

The data frame includes tree data corresponding to three forest inventories:

```
table(poblet_trees$Plot.Code)
```

```
##
##     POBL_CTL POBL_THI_AFT POBL_THI_BEF
##          267          189          261
```

Exercise solution

Step 3. Mapping trees from the control stand

We initialize an empty forest object using function `emptyforest()` from package **medfate**:

```
pobl_ctl <- emptyforest("POBL_CTL")
```

Exercise solution

Step 3. Mapping trees from the control stand

We initialize an empty forest object using function `emptyforest()` from package **medfate**:

```
pobl_ctl <- emptyforest("POBL_CTL")
```

To fill data for element `treeData` in the `forest` object, we need to define a mapping from column names in `poblet_trees` to variables in `treeData`. The mapping can be defined using a **named string vector**, i.e. a vector where element names are variable names in `treeData` and vector elements are strings of the variable names in `poblet_trees`:

```
mapping = c("Species.name" = "Species", "DBH" = "Diameter.cm")
```

Note: Using `"Species.name" = "Species"` we indicate that the function should interpret values in column `Species` as species names, not species codes.

Exercise solution

Step 3. Mapping trees from the control stand

We initialize an empty forest object using function `emptyforest()` from package **medfate**:

```
pobl_ctl <- emptyforest("POBL_CTL")
```

To fill data for element `treeData` in the `forest` object, we need to define a mapping from column names in `poblet_trees` to variables in `treeData`. The mapping can be defined using a **named string vector**, i.e. a vector where element names are variable names in `treeData` and vector elements are strings of the variable names in `poblet_trees`:

```
mapping = c("Species.name" = "Species", "DBH" = "Diameter.cm")
```

Note: Using `"Species.name" = "Species"` we indicate that the function should interpret values in column `Species` as species names, not species codes.

We can now replace the empty `treeData` in `pobl_ctl` using functions `subset()` and `forest_mapTreeTable()` from **medfateutils**:

```
pobl_ctl$treeData <- forest_mapTreeTable(subset(poblet_trees, Plot.Code=="POBL_CTL"),
                                         mapping_x = mapping, SpParams = SpParamsMED)
```

Exercise solution

Step 4. Check the mapping result

We can inspect the result using:

```
summary(pobl_ctl$treeData)
```

Exercise solution

Step 4. Check the mapping result

We can inspect the result using:

```
summary(pobl_ctl$treeData)
```

One way to evaluate if the tree data is correctly specified is to display a summary of the `forest` object using the `summary()` function defined in **medfate** for this object class:

```
summary(pobl_ctl, SpParamsMED)
```

```
## Tree density (ind/ha): 267
## Tree BA (m2/ha): 3.0179815
## Cover (%) trees (open ground): 42.1205627  shrubs: 0
## Shrub crown phytovolume (m3/m2): 0
## LAI (m2/m2) total: 0.530447  trees: 0.530447  shrubs: 0
## Live fine fuel (kg/m2) total: 0.1372838  trees: 0.1372838  shrubs: 0
## PAR ground (%): NA  SWR ground (%): NA
```

Exercise solution

Step 4. Check the mapping result

We can inspect the result using:

```
summary(pobl_ctl$treeData)
```

One way to evaluate if the tree data is correctly specified is to display a summary of the `forest` object using the `summary()` function defined in **medfate** for this object class:

```
summary(pobl_ctl, SpParamsMED)
```

```
## Tree density (ind/ha): 267
## Tree BA (m2/ha): 3.0179815
## Cover (%) trees (open ground): 42.1205627  shrubs: 0
## Shrub crown phytovolume (m3/m2): 0
## LAI (m2/m2) total: 0.530447  trees: 0.530447  shrubs: 0
## Live fine fuel (kg/m2) total: 0.1372838  trees: 0.1372838  shrubs: 0
## PAR ground (%): NA  SWR ground (%): NA
```

Are the values of tree density, stand basal area and stand LAI acceptable for a dense oak forest?

Exercise solution

Step 5. Specifying plot size

We were told that forest stand sampling was done using a *circular plot* whose radius was 15 m. We can calculate the sampled area using:

```
sampled_area = pi*15^2
```

Exercise solution

Step 5. Specifying plot size

We were told that forest stand sampling was done using a *circular plot* whose radius was 15 m. We can calculate the sampled area using:

```
sampled_area = pi*15^2
```

and use this information to map the tree data again, where we specify the parameter `plot_size_x`:

```
pobl_ctl$treeData <- forest_mapTreeTable(subset(poblet_trees, Plot.Code=="POBL_CTL"),
                                         mapping = mapping, SpParams = SpParamsMED,
                                         plot_size_x = sampled_area)
```

Exercise solution

Step 5. Specifying plot size

We were told that forest stand sampling was done using a *circular plot* whose radius was 15 m. We can calculate the sampled area using:

```
sampled_area = pi*15^2
```

and use this information to map the tree data again, where we specify the parameter `plot_size_x`:

```
pobl_ctl$treeData <- forest_mapTreeTable(subset(poblet_trees, Plot.Code=="POBL_CTL"),
                                         mapping = mapping, SpParams = SpParamsMED,
                                         plot_size_x = sampled_area)
```

We run again the summary and check whether stand's basal area and LAI make more sense:

```
summary(pobl_ctl, SpParamsMED)
```

```
## Tree density (ind/ha): 3777.27731604765
## Tree BA (m2/ha): 42.6957047
## Cover (%) trees (open ground): 100  shrubs: 0
## Shrub crown phytovolume (m3/m2): 0
## LAI (m2/m2) total: 6.7141704  trees: 6.7141704  shrubs: 0
## Live fine fuel (kg/m2) total: 1.7424533  trees: 1.7424533  shrubs: 0
## PAR ground (%): NA  SWR ground (%): NA
```

Exercise solution

Step 6. Adding tree heights

Another issue that we see is the `summary()` concerns percentage of PAR and SWR that reaches the ground, which have missing values. This indicates that light extinction cannot be calculated, in our case because tree heights are missing.

Exercise solution

Step 6. Adding tree heights

Another issue that we see is the `summary()` concerns percentage of PAR and SWR that reaches the ground, which have missing values. This indicates that light extinction cannot be calculated, in our case because tree heights are missing.

We should somehow estimate tree heights (in cm), for example using an allometric relationship:

```
poblet_trees$Height.cm = 100 * 1.806*poblet_trees$Diameter.cm^0.518
summary(poblet_trees$Height.cm)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   512.9   566.9   628.3   638.0   692.7   976.5
```

Exercise solution

Step 6. Adding tree heights

Another issue that we see is the `summary()` concerns percentage of PAR and SWR that reaches the ground, which have missing values. This indicates that light extinction cannot be calculated, in our case because tree heights are missing.

We should somehow estimate tree heights (in cm), for example using an allometric relationship:

```
poblet_trees$Height.cm = 100 * 1.806*poblet_trees$Diameter.cm^0.518
summary(poblet_trees$Height.cm)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   512.9   566.9   628.3   638.0   692.7   976.5
```

Once tree heights are defined, we can include them in our mapping vector:

```
mapping = c("Species.name" = "Species", "DBH" = "Diameter.cm", "Height" = "Height.cm")
```

Exercise solution

Step 6. Adding tree heights

Another issue that we see is the `summary()` concerns percentage of PAR and SWR that reaches the ground, which have missing values. This indicates that light extinction cannot be calculated, in our case because tree heights are missing.

We should somehow estimate tree heights (in cm), for example using an allometric relationship:

```
poblet_trees$Height.cm = 100 * 1.806*poblet_trees$Diameter.cm^0.518
summary(poblet_trees$Height.cm)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   512.9   566.9   628.3   638.0   692.7   976.5
```

Once tree heights are defined, we can include them in our mapping vector:

```
mapping = c("Species.name" = "Species", "DBH" = "Diameter.cm", "Height" = "Height.cm")
```

and rerun the tree data mapping.

Exercise solution

Step 7. Mapping trees from the managed stand

Now we can address the *managed* stand, which has two codes corresponding to *before* and *after* the thinning intervention. Let us first deal with the pre-thinning state:

```
pobl_thi_bef <- emptyforest("POBL_THI_BEF")
pobl_thi_bef$treeData <- forest_mapTreeTable(subset(poblet_trees,Plot.Code=="POBL_THI_BEF"),
                                             mapping_x = mapping, SpParams = SpParamsMED,
                                             plot_size_x = sampled_area)
summary(pobl_thi_bef$treeData)
```

```
##     Species            N              Height           DBH             Z50             Z95
##  Min.   :  4.0   Min.   :14.15   Min.   :512.9   Min.   : 7.50   Mode:logical   Mode:logical
##  1st Qu.: 19.0   1st Qu.:14.15   1st Qu.:563.7   1st Qu.: 9.00   NA's:261       NA's:261
##  Median :168.0   Median :14.15   Median :628.3   Median :11.10
##  Mean   :114.5   Mean   :14.15   Mean   :635.5   Mean   :11.51
##  3rd Qu.:168.0   3rd Qu.:14.15   3rd Qu.:681.9   3rd Qu.:13.00
##  Max.   :168.0   Max.   :14.15   Max.   :944.9   Max.   :24.40
##  NA's   :2
```

Beware of the missing values in column `Species`

Exercise solution

Step 8. Fixing species nomenclature

The `Species` variable contains two missing values. This will normally happen when some species cannot be identified. We can verify if this happens for other parts of the Poblet tree data:

```
sum(!(poblet_trees$Species %in% SpParamsMED$Name))
```

```
## [1] 4
```

If we display species counts we can identify which species is not being parsed:

```
table(poblet_trees$Species)
```

```
##
## Acer monspessulanum       Arbutus unedo Phillyrea latifolia     Quercus humilis        Quercus ilex
##                   2                 265                   6                   4                 440
```

In this case, the name used for the downy oak (*Quercus humilis*) is a synonym and needs to be replaced by its accepted name (*Quercus pubescens*), e.g.:

```
poblet_trees$Species[poblet_trees$Species=="Quercus humilis"] <- "Quercus pubescens"
```

Exercise solution

Step 8. Fixing species nomenclature

Now we repeat our mapping and check the results:

```
pobl_thi_bef$treeData <- forest_mapTreeTable(subset(poblet_trees,Plot.Code=="POBL_THI_BEF"),
                                             mapping_x = mapping, SpParams = SpParamsMED,
                                             plot_size_x = sampled_area)
summary(pobl_thi_bef, SpParamsMED)
```

```
## Tree density (ind/ha): 3692.39467973197
## Tree BA (m2/ha): 40.9224267
## Cover (%) trees (open ground): 100  shrubs: 0
## Shrub crown phytovolume (m3/m2): 0
## LAI (m2/m2) total: 6.5530107  trees: 6.5530107  shrubs: 0
## Live fine fuel (kg/m2) total: 1.6994566  trees: 1.6994566  shrubs: 0
## PAR ground (%): 2.7210404  SWR ground (%): 6.9269881
```

Like the control plot, the `summary()` indicates a dense oak forest.

Exercise solution

Step 9. Mapping trees from the managed stand

We can finally map tree data for the forest plot *after* the thinning intervention:

```
pobl_thi_aft <- emptyforest("POBL_THI_AFT")
pobl_thi_aft$treeData <- forest_mapTreeTable(subset(poblet_trees,Plot.Code=="POBL_THI_AFT"),
                                             mapping_x = mapping, SpParams = SpParamsMED,
                                             plot_size_x = sampled_area)
summary(pobl_thi_aft, SpParamsMED)
```

```
## Tree density (ind/ha): 2673.80304394384
## Tree BA (m2/ha): 31.6162035
## Cover (%) trees (open ground): 100  shrubs: 0
## Shrub crown phytovolume (m3/m2): 0
## LAI (m2/m2) total: 5.0833939  trees: 5.0833939  shrubs: 0
## Live fine fuel (kg/m2) total: 1.3224411  trees: 1.3224411  shrubs: 0
## PAR ground (%): 6.1061933  SWR ground (%): 12.6058106
```

And check the reduction caused by the thinning.

Exercise solution

Step 10. Checking the number of cohorts

So far we have considered that each tree record should correspond to a woody cohort. We can check the number of tree cohorts in each `forest` structure using:

```
nrow(pobl_ctl$treeData)
```

```
## [1] 267
```

```
nrow(pobl_thi_bef$treeData)
```

```
## [1] 261
```

```
nrow(pobl_thi_aft$treeData)
```

```
## [1] 189
```

Exercise solution

Step 10. Checking the number of cohorts

So far we have considered that each tree record should correspond to a woody cohort. We can check the number of tree cohorts in each `forest` structure using:

```
nrow(pobl_ctl$treeData)
```

```
## [1] 267
```

```
nrow(pobl_thi_bef$treeData)
```

```
## [1] 261
```

```
nrow(pobl_thi_aft$treeData)
```

```
## [1] 189
```

This large amount of cohorts can slow done simulations considerably!

Exercise solution

Step 11. Reducing the number of cohorts

One way of reducing the number of cohorts is via function `forest_mergeTrees()` from package **medfate**:

```
pobl_ctl <- forest_mergeTrees(pobl_ctl)
pobl_thi_bef <- forest_mergeTrees(pobl_thi_bef)
pobl_thi_aft <- forest_mergeTrees(pobl_thi_aft)
```

By default, the function will pool tree cohorts of the same species and diameter class (defined every 5 cm).

Exercise solution

Step 11. Reducing the number of cohorts

One way of reducing the number of cohorts is via function `forest_mergeTrees()` from package **medfate**:

```
pobl_ctl <- forest_mergeTrees(pobl_ctl)
pobl_thi_bef <- forest_mergeTrees(pobl_thi_bef)
pobl_thi_aft <- forest_mergeTrees(pobl_thi_aft)
```

By default, the function will pool tree cohorts of the same species and diameter class (defined every 5 cm).

We can check the new number of tree cohorts using again:

```
nrow(pobl_ctl$treeData)
```

```
## [1] 9
```

```
nrow(pobl_thi_bef$treeData)
```

```
## [1] 11
```

```
nrow(pobl_thi_aft$treeData)
```

```
## [1] 8
```

Exercise solution

Step 11. Reducing the number of cohorts

We can check whether stand properties were altered using the `summary()` function:

```
summary(pobl_thi_aft, SpParamsMED)
```

```
## Tree density (ind/ha): 2673.80304394384
## Tree BA (m2/ha): 31.6162035
## Cover (%) trees (open ground): 100  shrubs: 0
## Shrub crown phytovolume (m3/m2): 0
## LAI (m2/m2) total: 5.1144373  trees: 5.1144373  shrubs: 0
## Live fine fuel (kg/m2) total: 1.3298915  trees: 1.3298915  shrubs: 0
## PAR ground (%): 6.0028221  SWR ground (%): 12.4473851
```

Function `forest_mergeTrees()` will preserve the stand density and basal area that the stand description had before merging cohorts. Other properties like leaf area index may be slightly modified.

Tip: It is advisable to reduce the number of woody cohorts before running simulation models in **medfate**.

Exercise solution

Steps 12-13. Retrieving SoilGrids data

Retrieval of soil properties from SoilGrids can be done using function `soilgridsParams()` from package **medfateutils**.

Exercise solution

Steps 12-13. Retrieving SoilGrids data

Retrieval of soil properties from SoilGrids can be done using function `soilgridsParams()` from package **medfateutils**.

Assuming we know the plot coordinates, we first create an object `SpatialPoints` (see package **sp**):

```
cc = cbind(1.0219, 41.3443)
coords_sp <- SpatialPoints(cc, CRS(SRS_string = "EPSG:4326"))
```

Exercise solution

Steps 12-13. Retrieving SoilGrids data

Retrieval of soil properties from SoilGrids can be done using function `soilgridsParams()` from package **medfateutils**.

Assuming we know the plot coordinates, we first create an object `SpatialPoints` (see package **sp**):

```
cc = cbind(1.0219, 41.3443)
coords_sp <- SpatialPoints(cc, CRS(SRS_string = "EPSG:4326"))
```

This object can be used to query SoilGrids using `soilgridsParams()`:

```
pobl_soil_props <- soilgridsParams(coords_sp, widths = c(300, 700, 1000))
```

Exercise solution

Steps 12-13. Retrieving SoilGrids data

Retrieval of soil properties from SoilGrids can be done using function `soilgridsParams()` from package **medfateutils**.

Assuming we know the plot coordinates, we first create an object `SpatialPoints` (see package **sp**):

```
cc = cbind(1.0219, 41.3443)
coords_sp <- SpatialPoints(cc, CRS(SRS_string = "EPSG:4326"))
```

This object can be used to query SoilGrids using `soilgridsParams()`:

```
pobl_soil_props <- soilgridsParams(coords_sp, widths = c(300, 700, 1000))
```

This function returns a data frame of soil properties:

```
pobl_soil_props
```

```
##   widths     clay     sand       om       bd  rfc
## 1    300 26.43333 31.06667 4.133333 1.166667 18.0
## 2    700 30.40000 29.75000 0.900000 1.440000 19.2
## 3   1000 31.60000 29.60000 0.610000 1.500000 20.9
```

Exercise solution

Steps 14-15. Building the soil object

This data frame is a physical description of the soil. Remember that the soil data structure for **medfate** simulations is built using function `soil()`:

```
pobl_soil <- soil(pobl_soil_props)
```

We can inspect the soil definition using `print()` (or writing `pobl_soil` in the console).

Exercise solution

Steps 14-15. Building the soil object

This data frame is a physical description of the soil. Remember that the soil data structure for **medfate** simulations is built using function `soil()`:

```
pobl_soil <- soil(pobl_soil_props)
```

We can inspect the soil definition using `print()` (or writing `pobl_soil` in the console).

SoilGrids usually underestimates the amount of rocks in the soil, because soil samples do not normally contain large stones or blocks. Realistic simulations should reduce the soil water holding capacity by increasing `rfc`. For example, here we will assume that the third layer contains 80% of rocks:

```
pobl_soil_props$rfc[3] = 80
```

Exercise solution

Steps 14-15. Building the soil object

This data frame is a physical description of the soil. Remember that the soil data structure for **medfate** simulations is built using function `soil()`:

```
pobl_soil <- soil(pobl_soil_props)
```

We can inspect the soil definition using `print()` (or writing `pobl_soil` in the console).

SoilGrids usually underestimates the amount of rocks in the soil, because soil samples do not normally contain large stones or blocks. Realistic simulations should reduce the soil water holding capacity by increasing `rfc`. For example, here we will assume that the third layer contains 80% of rocks:

```
pobl_soil_props$rfc[3] = 80
```

If we rebuild the soil object and inspect its properties…

```
pobl_soil <- soil(pobl_soil_props)
pobl_soil
```

…we will see the decrease in water-holding capacity.

Exercise solution

Steps 16-17. Interpolating weather

Obtaining daily weather data suitable for simulations is not straightforward either. Here we illustrate one way of obtaining such data with package **meteoland**.

Exercise solution

Steps 16-17. Interpolating weather

Obtaining daily weather data suitable for simulations is not straightforward either. Here we illustrate one way of obtaining such data with package **meteoland**.

We begin by building an object of S4 class `SpatialPointsTopography`, which extends `SpatialPoints`and contains both the coordinates of our site as well as topographic variables.

```
pobl_spt <- SpatialPointsTopography(coords_sp,
                                    elevation = 850, slope = 15.1, aspect = 15)
pobl_spt
```

```
## Object of class SpatialPointsTopography
##         coordinates elevation slope aspect
## 1 (1.0219, 41.3443)       850  15.1     15
```

Exercise solution

Steps 16-17. Interpolating weather

Obtaining daily weather data suitable for simulations is not straightforward either. Here we illustrate one way of obtaining such data with package **meteoland**.

We begin by building an object of S4 class `SpatialPointsTopography`, which extends `SpatialPointsand` contains both the coordinates of our site as well as topographic variables.

```
pobl_spt <- SpatialPointsTopography(coords_sp,
                                    elevation = 850, slope = 15.1, aspect = 15)
pobl_spt
```

```
## Object of class SpatialPointsTopography
##         coordinates elevation slope aspect
## 1 (1.0219, 41.3443)       850  15.1     15
```

The more difficult part of using package **meteoland** is to assemble weather data from surface weather stations into an object of class `MeteorologyInterpolationData`.

Exercise solution

Steps 16-17. Interpolating weather

Obtaining daily weather data suitable for simulations is not straightforward either. Here we illustrate one way of obtaining such data with package **meteoland**.

We begin by building an object of S4 class `SpatialPointsTopography`, which extends `SpatialPoints`and contains both the coordinates of our site as well as topographic variables.

```
pobl_spt <- SpatialPointsTopography(coords_sp,
                                    elevation = 850, slope = 15.1, aspect = 15)
pobl_spt
```

```
## Object of class SpatialPointsTopography
##         coordinates elevation slope aspect
## 1 (1.0219, 41.3443)       850  15.1     15
```

The more difficult part of using package **meteoland** is to assemble weather data from surface weather stations into an object of class `MeteorologyInterpolationData`.

Here we will assume that such an object is already available, by using the example object provided in the **meteoland** package.

```
data("exampleinterpolationdata")
```

Exercise solution

Steps 16-17. Interpolating weather

Once we have this interpolator, obtaining interpolated weather for a set of target points is rather straightforward using function `interpolationpoints()` from **meteoland**:

```
meteo <- interpolationpoints(exampleinterpolationdata, pobl_spt)
```

```
## Warning in interpolationpoints(exampleinterpolationdata, pobl_spt): CRS projection of 'points'
## adapted to that of 'object'.

## Processing point '1' (1/1) -

## Warning in interpolationpoints(exampleinterpolationdata, pobl_spt): Point '1' outside the boundary
## box of interpolation data object.

##  done.
```

Note that a **warning** was raised, because the boundary box of the interpolator does not include the location of the Poblet forests. In this case, the result of the interpolation is not reliable and should not be used!

Exercise solution

Steps 16-17. Interpolating weather

Once we have this interpolator, obtaining interpolated weather for a set of target points is rather straightforward using function `interpolationpoints()` from **meteoland**:

```
meteo <- interpolationpoints(exampleinterpolationdata, pobl_spt)
```

```
## Warning in interpolationpoints(exampleinterpolationdata, pobl_spt): CRS projection of 'points'
## adapted to that of 'object'.

## Processing point '1' (1/1) -

## Warning in interpolationpoints(exampleinterpolationdata, pobl_spt): Point '1' outside the boundary
## box of interpolation data object.

##  done.
```

Note that a **warning** was raised, because the boundary box of the interpolator does not include the location of the Poblet forests. In this case, the result of the interpolation is not reliable and should not be used!

The output of function `interpolationpoints()` is an object of S4 class `SpatialPointsMeteorology`. We can access the weather data frame by subsetting the appropriate element of slot `data`:

```
pobl_weather <- meteo@data[[1]]
head(pobl_weather, 2)
```

M.C. Escher - Babel tower, 1928