

# SpaDES For Dummies

Guide to the SpaDES modelling toolkit in R

Ceres Barros

Eliot McIntire

Tati Micheletti

Alex Chubaty

2023-11-16



# Table of contents

Preface	1
1 Introducing SpaDES with a dummy ecological model	3
1.1 BEFORE SpaDES . . . . .	3
1.1.1 Setup . . . . .	4
1.1.2 Species abundance “simulations” . . . . .	4
1.1.3 Temperature “simulations” . . . . .	5
1.1.4 Data analysis . . . . .	6
1.2 AFTER SpaDES . . . . .	8
1.2.1 The control script . . . . .	8
1.2.2 General module structure: <i>speciesAbundance</i> module .	9
1.2.2.1 Defining the module . . . . .	10
1.2.2.2 Events and event functions . . . . .	13
1.2.2.3 Scheduling events . . . . .	15
1.2.2.4 <code>.inputObjects</code> function . . . . .	17
1.2.2.5 Additional module functions . . . . .	19
1.2.3 Creating and adding additional modules: the <i>temperature</i> module . . . . .	19
1.2.4 Modules that depend on other modules: the <i>speciesTempLM</i> module . . . . .	23
1.2.5 Simulation . . . . .	29
1.2.5.1 Simulation setup in a “global” script . . . . .	29
1.2.5.2 Checking the simulation setup . . . . .	30
1.2.5.3 Running SpaDES . . . . .	33
1.2.6 Additional notes . . . . .	34
2 A more realistic example of SpaDES	35

2.1	The example: projecting species distribution shifts under climate change . . . . .	36
2.2	Module creation and coding . . . . .	36
2.2.1	Data modules . . . . .	39
2.2.1.1	<i>speciesAbundanceData</i> module: . . . . .	39
2.2.1.2	<i>climateData</i> module: . . . . .	44
2.2.2	Prediction module . . . . .	53
2.2.3	Additional module scripts and functions . . . . .	62
2.3	Running the model . . . . .	64
2.3.1	Ensuring all packages are installed . . . . .	64
2.3.2	Simulation set-up . . . . .	66
2.3.3	Simulation runs . . . . .	68
2.3.3.1	Adding a new climate scenario . . . . .	73
2.3.3.2	Proposed exercises . . . . .	76
2.3.3.3	Making use of <code>simList</code> for reporting . . . . .	76
2.4	Caching . . . . .	79
2.4.1	Explicitly caching operations . . . . .	79
2.4.2	Implicit caching of events . . . . .	80
2.4.3	Controlling caching without changing module code . . . . .	80
2.5	Best practices . . . . .	81
2.5.1	Reproducible package installation . . . . .	81
2.5.2	Protect yourself and others from common mistakes/problems . . . . .	82
2.5.3	Readable code . . . . .	82
2.5.4	Module documentation – module <code>.Rmd</code> . . . . .	83
2.5.5	Coding for the future . . . . .	83
2.5.6	Transparent models . . . . .	84
2.6	Additional notes . . . . .	84
	<b>References</b>	<b>85</b>
	<b>Appendices</b>	<b>87</b>
	<b>A Complete R script to Chapter 1</b>	<b>87</b>
	<b>B Complete R script to Chapter 2</b>	<b>91</b>

# List of Figures

1.1	Inputs and outputs in SpaDES: Object A comes from outside of the module (e.g. from an internet URL, from data you have, or from <code>.inputObjects</code> ), while Module Z produces object C. Both objects serve as inputs for Module Y, which in return produce as outputs objects B and D, respectively from objects A and C. As Module Z uses a simple function <i>internally</i> to create object C, it doesn't have any inputs, such as our dummy example. . . . .	11
1.2	Diagram of module connections. . . . .	31
1.3	Module diagram showing module inter-dependencies with object names. . . . .	32
1.4	Simulation plots: Final plot of the simulation . . . . .	33
2.1	Study area within Canada. . . . .	68
2.2	Module network diagram. . . . .	69
2.3	Module diagram showing module inter-dependencies with object names. . . . .	70
2.4	Prediction plots. Input <i>Picea glauca</i> percent cover across the landscape. Note that values are converted to presence/absence. . . . .	71
2.5	Prediction plots. Bioclimatic variables under baseline (year 1) and future conditions. . . . .	72
2.6	Prediction plots: Raw predicted values of species probability of occurrence under (left to right) baseline climate conditions (first year of simulation), 2021-2040, 2041-2060, 2061-2080 and 2081-2100 climate conditions (second to fifth years of simulation) - using MaxEnt. . . . .	74

2.7 Prediction plots: Predictions of <i>Picea glauca</i> presence/absence under (left to right) baseline climate conditions (first year of simulation), 2021-2040, 2041-2060, 2061-2080 and 2081-2100 climate conditions (second to fifth years of simulation) - using MaxEnt. . . . .	75
2.8 Adding a new scenario: Predictions of <i>Picea glauca</i> probabilities of presences and presence/absence under (left to right) baseline climate conditions, 2041-2060, and 2081-2100 climate projections under two emission scenarios (SSP 136 and SSP 585, the default) – showing MaxEnt forecasts only. . . . .	78

# List of Tables



# Preface

DOI [10.5281/zenodo.8310405](https://doi.org/10.5281/zenodo.8310405)

This guide will take you through how to make and link your own modules using **SpaDES** in two examples. Both examples draw on basic uses of statistical models in ecology, notably the relationships between environmental variables and species abundance and presence.

[Chapter 1](#) is very minimal, and uses only dummy data. It is meant to introduce you to the different components of a **SpaDES** module.

[Chapter 2](#) uses real and freely available data, and provides a deeper look into several useful aspects of **SpaDES**, notably caching and spatial data processing.

Full R scripts to execute **SpaDES** workflow examples can be found in the Appendices section.

To install **SpaDES**, please have a look at [SpaDES installation](#), or follow steps at the start of [Chapter 2](#).



# Chapter 1

## Introducing SpaDES with a dummy ecological model

Authors: Ceres Barros, Tati Micheletti

Let's imagine we want to explore how the relationship between a species' abundance and temperature changes over time. Both the abundance data and the temperature data are being constantly updated by a simulation model, and we want to analyse the relationship between the two iteratively, without needing to manually run a script to account for the newly generated data inputs.

### 1.1 BEFORE SpaDES

If we use R to develop our species abundance and temperature simulation models in the ‘conventional way’, we’ll probably have i) (the worst case scenario) several scripts that run simulations and data treatment/analysis separately and have to be executed manually, or ii) a long script where everything happens - the simulations and data analysis -, iii) a main script that sources others that do the simulation and analyses. Option i) is more common when different software are used for different parts of the process (e.g., a simulation model in *C++* generates data that is then analysed in R). Option ii) is inconvenient because very long scripts make changes and updates to the script - debugging can also be more tiresome. Option iii), is similar to the **SpaDES** way of thinking. The difference is that **SpaDES** defines a standard way of writing different components of a model, or of a modelling

framework. This makes changing, updating and sharing code - or modules - easier, as well as swapping and adding modules in a modelling framework.

The example below is so minimal that it is unlikely to show the full benefits of using SpaDES - the same could be accomplished with a fairly short script. However, it introduces the different parts of a module and how to link modules.

[Chapter 2](#) goes a step further and uses real datasets to project species presences across a landscape in Canada. In [Chapter 2](#), we introduce SpaDES features that we most commonly use in our work (e.g., caching and spatial data processing) and provide some coding best practices that we use ourselves (e.g., code assertions).

### 1.1.1 Setup

This is what you'd normally do... Install all the packages in some way that you probably didn't record in your scripts and then start your script with loading the packages:

```
## please start from a clean R session
remotes::install_github("ropensci/NLMR") ## you will need this ;)

library(terra)
library(quickPlot)
library(ggplot2)
library(ggpubr)
```

And now create a raster template:

```
r <- rast(nrows = 100, ncols = 100, xmin = -50, xmax = 50, ymin = -50,
           ymax = 50)
```

### 1.1.2 Species abundance “simulations”

Our VERY simple “simulation” model (in form of a function) generates rasters that follow a Gaussian distribution

```
abundance_model <- function(ras, Time) {
  abund_outputs <- list()
  for (t in 1:Time) {
    abund_outputs[[t]] <- NLMR::nlm_mpd(ncol = ncol(ras),
                                             nrow = nrow(ras), resolution = unique(res(ras)),
```

```

        roughness = 0.5, rand_dev = 100, rescale = TRUE,
        verbose = FALSE) |>
      rast()
    }
  return(abund_outputs)
}

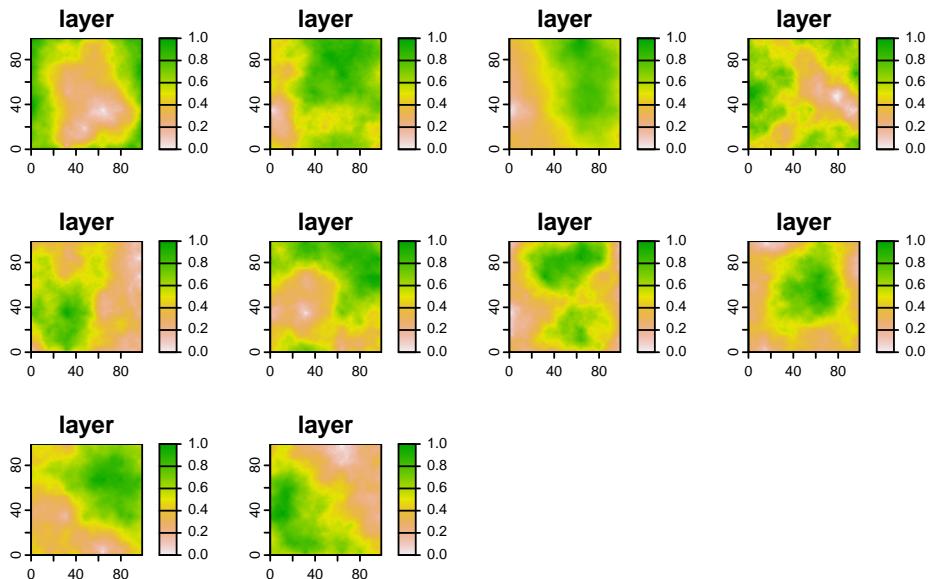
```

Set the length of the simulation (or simply the number of model iterations), run it and plot results (all ABUNDANCE plots together):

```

Time <- 10
abundance <- abundance_model(ras = r, Time = Time)
dev()
plot(rast(abundance))

```



### 1.1.3 Temperature “simulations”

The temperature simulation model will be similar to the vegetation one - remember this is a dummy example.

```

temperature_model <- function(ras, Time) {
  temp_outputs <- list()
  for (t in 1:Time) {

```

```

    temp_outputs[[t]] <- NLMR::nlm_mpd(ncol = ncol(ras),
      nrow = nrow(ras), resolution = unique(res(ras)),
      roughness = 0.5, rand_dev = 10, rescale = FALSE,
      verbose = FALSE) |>
      rast()
}
return(temp_outputs)
}

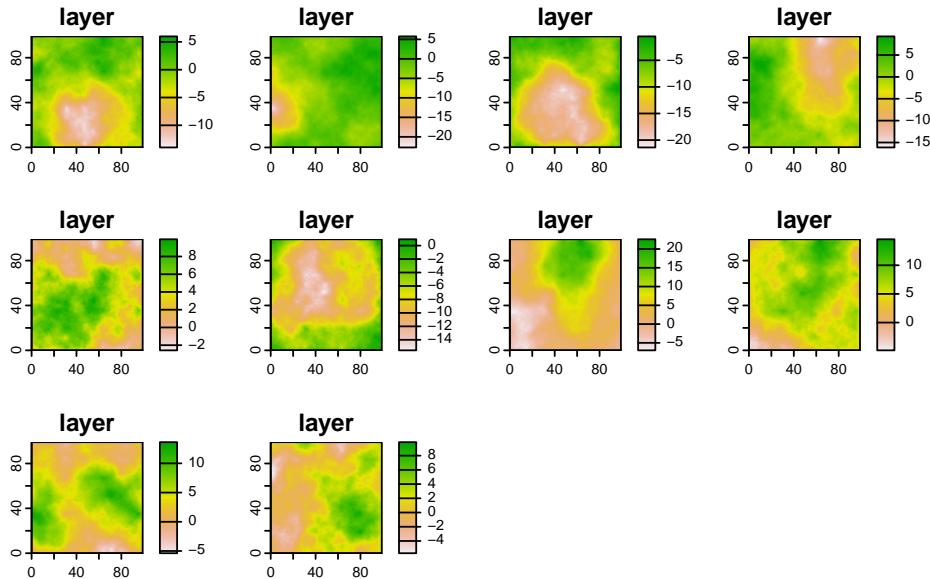
```

Run the model and plot results (all temperature plots together)

```

temperature <- temperature_model(ras = r, Time = Time)
plot(rast(temperature))

```



#### 1.1.4 Data analysis

Now we analyse if species abundance and temperature are correlated.

First, we create the data analysis function (a simple linear model):

```

stats_analysis <- function(Data) {
  if (all(c("abund", "temp") %in% colnames(Data))) {
    lm1 <- lm(abund ~ temp, data = Data)
  }
}

```

```

ggplot(Data) + geom_point(aes(x = temp, y = abund)) +
  geom_abline(intercept = lm1$coefficients["(Intercept)"],
  slope = lm1$coefficients["temp"], size = 2, col = "blue") +
  theme_bw() + labs(x = "Temp.", y = "Species abundance")

} else {
  stop("Data must contain 'abund' and 'temp' columns")
}
}

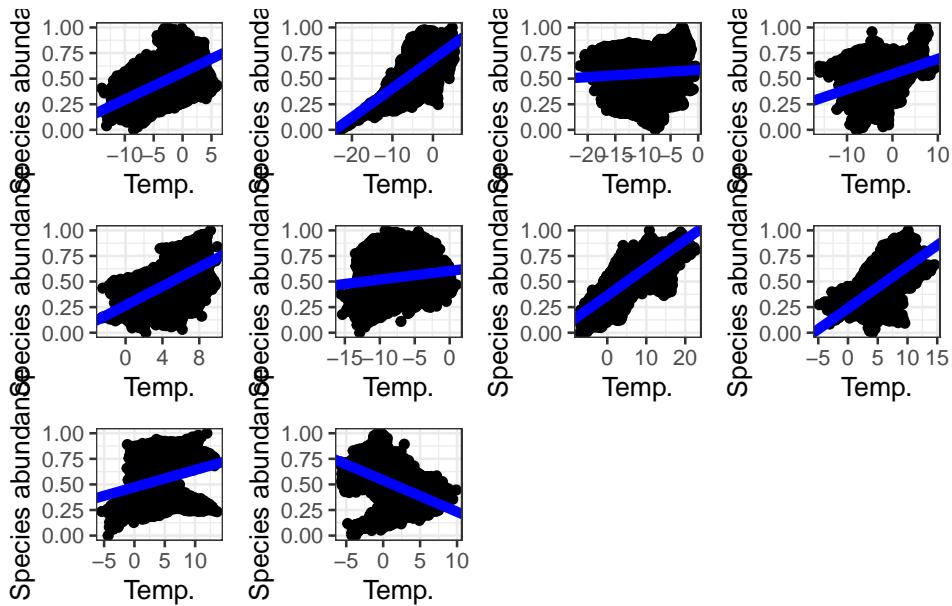
```

Then we create a loop to analyse each plot of our time-series:

```

lmPlots <- list()
for (t in 1:Time) {
  outputdata <- data.frame(abund = as.vector(abundance[[t]] []),
    temp = as.vector(temperature[[t]] []))
  lmPlots[[t]] <- stats_analysis(Data = outputdata)
}
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2
## 3.4.0.
## i Please use `linewidth` instead.
ggarrange(plotlist = lmPlots)

```



## 1.2 AFTER SpaDES

### 1.2.1 The control script

Let us now solve the same problem using the SpaDES approach.<sup>1</sup> We start by creating an `.R` script (it can have any name) that sets up and runs the SpaDES model. The control script for this example is located on the root of the *SpaDES4Dummies* GitHub repository under the name `Part1_DummyModel.R`. Note that Markdown (`.Rmd`) scripts can also be used instead of ‘`R`’ scripts.

We start by making sure all SpaDES packages and their dependencies are installed (and that the installation is *scripted*) using the `Require` package.

```
## start again from a clean R session
options(repos = c("https://predictiveecology.r-universe.dev/",
                  CRAN = "https://cloud.r-project.org"))

## decide where you're working
mainPath <- "~/SpaDES4Dummies_Part1"
pkgPath <- file.path(mainPath, "packages", version$platform,
                     paste0(version$major, ".", strsplit(version$minor, "[.]")[[1]][1]))
dir.create(pkgPath, recursive = TRUE)
.libPaths(pkgPath, include.site = FALSE)  ## install packages in project library

if (!"remotes" %in% installed.packages(lib.loc = pkgPath)) install.packages("remotes")

if (!"Require" %in% installed.packages(lib.loc = pkgPath) ||
    packageVersion("Require", lib.loc = pkgPath) < "0.3.1.9015") {
  remotes::install_github("PredictiveEcology/Require@2788b023ad191c29346ef8c64")
  upgrade = FALSE)
}

Require::Require(c("SpaDES", "DiagrammeR"), require = FALSE,
                upgrade = FALSE, dependencies = TRUE, standAlone = TRUE)  ## automatically download dependencies

library(SpaDES)

setPaths(cachePath = file.path(mainPath, "cache"), inputPath = file.path(mainPath,
  "inputs"), modulePath = file.path(mainPath, "modules"), outputPath = file.path(mainPath,
```

---

<sup>1</sup>The full R script to run this example Appendix A .

```

"outputs"))

getPaths() ## check that this is what you wanted

## Let's create a self-contained module that will simulate
## the species' abundance for any given period of time and
## frequency.
if (!dir.exists(file.path(getPaths()$modulePath, "speciesAbundance"))) {
  newModule(name = "speciesAbundance", path = getPaths()$modulePath)
}

```

We then create modules using `newModule`. `newModule` creates a module folder (`speciesAbundance`) inside `/modules` that contains both the module `.R` script template, as well as the documentation template (the `.Rmd` file). Although we will not be discussing the `.Rmd` file, please bear in mind that this is a **fundamental** part of creating a reproducible and transparent module - check out the [Guide to Reproducible Code in Ecology and Evolution](#) from the British Ecological Society). The documentation should contain a the description of the module, its input, parameters and outputs, and potentially a reproducible examples of how the module is executed.

`newModule` also created the folder `/data` where data necessary to the module can be put in, and the folder `/tests` that may contain testing scripts. We will not be using either of them in this example.

**!/\\ ATTENTION /!\\**

`newModule` should only be run once, otherwise it will replace all edits and contents of the module folder with the templates - this is why it is wrapped in an `if` statement above.

Now go ahead, open the `speciesAbundance.R` script and have a look at it.

---

### 1.2.2 General module structure: `speciesAbundance` module

The module template contains all the essential components of a module, with examples, and may seem overwhelming at first. We'll go through it step by step (although not necessarily following the order of the script). The module script can be divided into 4 parts:

**Defining the module:** this is where the module is **defined**, i.e., the module’s metadata (e.g. module author(s), time units, basic parameters, general inputs and outputs, etc.);

**Events and event functions:** these are the “actions” (or events) executed in the module (i.e. species reproduction, plotting, saving parameters) - simply put, **WHAT** the module does;

**Scheduling events:** this is how **SpaDES** schedules when each event is going to happen - in which order (e.g. during the simulation, when will **SpaDES** plot a graph) - simply put, **WHEN** the module does it;

**Additional module functions:** any additional functions needed (e.g. this is used to keep the coding of your module as clear and straightforward as possible);

The first thing to note is that **the user does not need to manually run** any of the code inside a module’s .R script. The function **simInit()** will do so when it sets up the simulation. We will see this see this later in detail.

### 1.2.2.1 Defining the module

The first section of the script defines the module’s **metadata**. It allows defining the module’s author, keywords, any required packages and module(s) and their versions, but also parameters (and their default values) and input objects that the module requires, and the output objects it creates.

Although this dummy module example requires no true input data, we will define the template raster ‘r’ as an “input” in the **expectsInput** function, and provide a default object in **.inputObjects** (see below). As for the outputs, it produces a list of abundance rasters (produced during the **abundanceSim** event). So we define it as an output in **theCreatesOutput** function.

Note that we removed several parameters that come with the template created by the **newModule** function, as they are not needed for this example.

To distinguish what input and output objects are in the context of a module, a good rule of thumb is that inputs are all the **sim\$...** objects that appear for the first time (in the module events) on the **right-hand side** of a **<-**, whereas output parameters are the **sim\$...** objects that appear for the first time to the **left-hand side** of a **<-**. Another way of explaining it for objects is illustrated in Figure 1.1 :

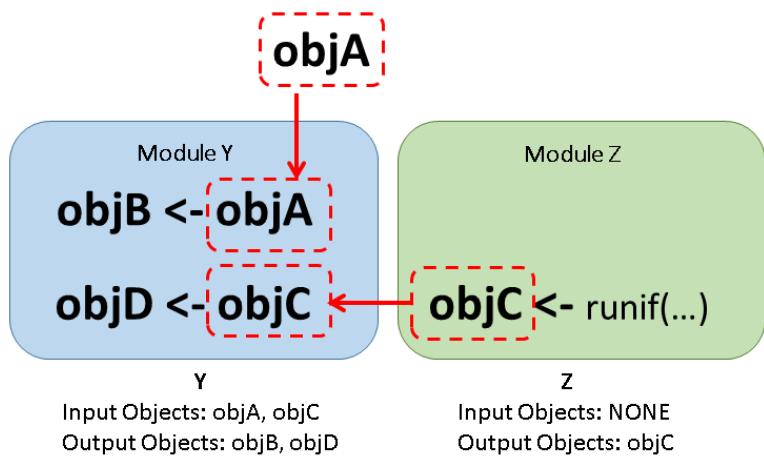


Figure 1.1: Inputs and outputs in SpaDES: Object A comes from outside of the module (e.g. from an internet URL, from data you have, or from `.inputObjects`), while Module Z produces object C. Both objects serve as inputs for Module Y, which in turn produce as outputs objects B and D, respectively from objects A and C. As Module Z uses a simple function *internally* to create object C, it doesn't have any inputs, such as our dummy example.

## 12 CHAPTER 1. INTRODUCING SPADES WITH A DUMMY ECOLOGICAL MODEL

The default input objects created by the `.inputObjects` function (see [`.inputObjects` function]) during the `simInit` call are exceptions to this rule.

Here is how we defined the `speciesAbundance` module:

```
defineModule(sim, list(
  name = "speciesAbundance",
  description = "",
  keywords = "",
  authors = person("Me", email = "me@example.com", role = c("aut", "cre")),
  childModules = character(0),
  version = list(speciesAbundanceData = "1.0.0"),
  timeframe = as.POSIXlt(c(NA, NA)),
  timeunit = "year",
  citation = list("citation.bib"),
  documentation = deparse(list("README.txt", "speciesAbundance.Rmd")),
  reqdPkgs = list("SpaDES.core (>=2.0.2)",
                  "terra", "quickPlot"),
  parameters = bindrows(
    #defineParameter("paramName", "paramClass", value, min, max, "parameter desc")
    defineParameter("simulationTimeStep", "numeric", 1, NA, NA,
                   "This describes the simulation time step interval"),
    defineParameter(".plotInitialTime", "numeric", 1, NA, NA,
                   "Describes the simulation time at which the first plot event"),
    defineParameter(".plotInterval", "numeric", 1, NA, NA,
                   "Describes the simulation time interval between plot events."),
  ),
  inputObjects = bindrows(
    # expectsInput("objectName", "objectClass", "input object description", source)
    expectsInput("r", objectClass = "SpatRaster", desc = "Template raster")
  ),
  outputObjects = bindrows(
    #createsOutput("objectName", "objectClass", "output object description", ...)
    createsOutput("abundRasters", "list", "List of layers of species abundance")
  )
))
```

Note that the package versions that you define will depend on the ones that are installed on your computer. So take care to change them accordingly. The `SpaDES` package version suggested by the template reflects the version on your computer.

The rest of the script defines the events and their sequences for this module - remember **SpaDES** = Spatial Discrete Event Simulator - and the events themselves.

**/!\ ATTENTION /!\**

*defineModule()* is not intended to be run directly by the user – it is run internally during a *simInit()* call (see *Simulation setup in a “global” script*). In other words, you don’t run any part of a module’s code directly in your session; you run *simInit()* with that module listed in the modules argument.

### 1.2.2.2 Events and event functions

Module events are defined and scheduled in the *doEvent.<module name>* function (in this example, *doEvent.speciesAbundance* function; see [Scheduling events]). Since we are only interested in simulating and plotting species abundances, we removed unnecessary events from the script and kept: the initialisation (*init*), an abundance simulation event (*SimulAbund*) and a plotting event (*abundPlot*). Each of these events can execute one or more functions.

*Event functions* (actual R functions) mustn’t be confused with *event names*, which are the names of the events appearing in the *doEvent.<module name>*.

**/!\ ATTENTION /!\**

*Event functions take only one argument, *sim* (the *SpaDES.core::simList* object that stores all objects, modules, functions, etc., of a simulation; see *?simList*) and event functions always (and only) return *sim* (using *return(invisible(sim))*).*

#### 1.2.2.2.1 Initialisation event function

The initialisation event function (here, *abundanceInit*) can be seen as the starting point of the module. Unlike the *init* event, which must always be present, the function itself does not need to exist (see [Scheduling events]) and can have whatever name we want.

Usually, this function will do pre-simulation steps that are only needed to be executed once. In our dummy example, it creates a template raster and a storage list for our species abundance outputs (which will also be rasters).

Notice that the only argument to `abundanceInit` is `sim`, a `simList` object that is also its only output.

```
abundanceInit <- function(sim) {
  ## create storage list of species abundance
  sim$abundRasters <- list()

  return(invisible(sim))
}
```

#### 1.2.2.2.2 Abundance simulation event function

The function `abundanceSim` is the core event function of this module, where species abundances are generated via the event. Notice how instead of a *for-loop*, `abundanceSim` runs the `abundance_model` function (which we define separately below) and stores its outputs in the `sim$abundRaster` object. Notice as well that we use `time(sim)` as the identifier of the list slots where outputs are stored (see `?SpaDES.core::time`).

As before, the sole argument and output to this event function is the `sim` object.

```
abundanceSim <- function(sim) {
  ## Generate species abundances - our 'simulation'
  sim$abundRasters[[as.character(time(sim))]] <- abundance_model(ras = sim$r)

  return(invisible(sim))
}
```

The `abundanceSim` function was called `Event1` in the template.

#### 1.2.2.2.3 Plotting event function

Finally, we created the `abundancePlot` event function to plot the species abundance rasters that are produced by the `abundanceSim` event function. Again, the sole argument and output of this function is `sim`.

```
abundancePlot <- function(sim) {
  ## plot abundances
  plotTitle <- paste("Species abundance\nat time", names(sim$abundRasters)[length(sim$abundRasters)])
  abundPlot <- sim$abundRasters[[length(sim$abundRasters)]]
  Plot(abundPlot, title = plotTitle, new = TRUE, addTo = "abundPlot")
```

```

    return(invisible(sim))
}

```

The `abundancePlot` function was called `plotFun` in the template.

### 1.2.2.3 Scheduling events

The order in which module events are executed is determined by the `doEvent.<module name>` function. This function also defines the events themselves and what happens in them. The `switch` function executes each event (here `init`, `SimulAbund`, and `abundPlot`) and the events schedule themselves. Two things are of particular importance:

1. The `init` event is **mandatory**. This is the only event whose *name* that cannot be changed and that cannot be removed (even if it does not execute any event functions). All other **events** are optional and can be renamed. `Spades` searches and executes all modules' `init` events automatically. Note that the names of event functions executed during `init` *can* have any name: here we changed the `Init` function name (suggested by the template) to `abundanceInit`.
2. **Events should only schedule themselves**. The only exception is the `init`, which schedules the first time all other events are executed (even if a particular event only occurs once at the end of the simulation).

It is usually easier to fill the `doEvent.<module name>` function *after* having defined the event functions (as we did above). For instance, we know that plotting should occur after the generation of species abundances, and so the `abundPlot` will be scheduled to occur after the `SimulAbund` event, by changing event priority (see `?priority`).

This is how we configured our `doEvent.speciesAbundance` function:

```

doEvent.speciesAbundance = function(sim, eventTime, eventType,
  debug = FALSE) {
  switch(eventType, init = {
    ## do stuff for this event
    sim <- abundanceInit(sim)

    ## schedule future event(s)
    sim <- scheduleEvent(sim, eventTime = start(sim), moduleName = "speciesAbundance"
      eventType = "SimulAbund")
  }
}

```

```

        sim <- scheduleEvent(sim, eventTime = P(sim)$plotInitialTime,
                              moduleName = "speciesAbundance", eventType = "abundPlot",
                              eventPriority = .normal() + 0.5)
    }, SimulAbund = {
        ## do stuff for this event
        sim <- abundanceSim(sim)

        ## schedule future event(s)
        sim <- scheduleEvent(sim, eventTime = time(sim) + P(sim)$simulationTimeStep,
                              moduleName = "speciesAbundance", eventType = "SimulAbund")
    }, abundPlot = {
        ## do stuff for this event
        sim <- abundancePlot(sim)

        ## schedule future event(s)
        sim <- scheduleEvent(sim, eventTime = time(sim) + P(sim)$plotInterval,
                              moduleName = "speciesAbundance", eventType = "abundPlot",
                              eventPriority = .normal() + 0.5)
    }, warning(paste("Undefined event type: '", current(sim)[1],
                     "eventType", with = FALSE], "' in module '", current(sim)[1],
                     "moduleName", with = FALSE], "'", sep = ""))
    return(invisible(sim))
}

```

We suggest having a look at `?base::switch` too fully understand its behaviour. In short, `base::switch` tells R to execute (or switch) different code depending on the value of EXPR (here `eventType`). Here, this means that the behaviour of the function `doEvent.speciesAbundance` will change depending on the present `eventType`. So we need to define what behaviour it should have for each event type defined in the module - namely, which functions will be executed and whether to schedule future events with `scheduleEvent`.

#### 1.2.2.3.1 init

The first event is, obviously, `init` - again **its name cannot be changed**.

In `init` we run the initialisation event function (`abundanceInit`) - optional - and schedule the first occurrence of all other events (here, the abundance simulation, `SimulAbund`, and plotting, `abundPlot`, events). Because the `init` is the only event that Spades always executes at the start of the sim-

ulation, if no events are scheduled during `init`, no events will be executed after the `init`. Notice two things:

1. The `SimulAbund` event is scheduled at `start(sim)` (i.e. at the first time step of the simulation), which means that it will run after the `init` event, but still in the same “year”.
2. `init` schedules the first plotting event to be executed at the time defined by the `.plotInitialTime` parameter, which is stored in the `sim` object (and obtained using `SpaDES.core::P(sim)`), but with a slightly lower event priority `eventPriority = .normal() + 0.5` (see `?priority`).

#### 1.2.2.3.2 SimulAbund

The `SimulAbund` event is defined next. This event used to be called `event1` in the template, and we changed its name to be more informative of what it does. It is the core event of this module, where species abundances are generated via the event function `abundanceSim`.

The even also **schedules itself** to occur at a frequency defined by the `simulationTimeStep` parameter

#### 1.2.2.3.3 abundPlot

Finally, we schedule the plotting event, `abundPlot` (which used to be called `plot` in the template). Similarly to the `SimulAbund` event, it executes an event function (`abundancePlot`) and reschedules itself. An important difference is that it uses the `.plotInterval` parameter, instead of `simulationTimeStep`, when rescheduling itself. This way, future events will occur depending on the time step and plot interval parameters defined in the global script (or their default values defined in the metadata section).

#### 1.2.2.4 .inputObjects function

The end of the template `.R` script defines a function called `.inputObjects`. This is where the developer should include code to provide the defaults for any input objects required by the module. This is the ideal place to produce the template raster, `r`, instead of doing so in `abundanceInit`. This will allow a future user (or module) to provide their own `r` object (e.g. for another study area). If on the other hand we don’t do this and create `r` during the `init` event, any `r` supplied by the user will be overridden by the execution of `init`.

As a rule of thumb, default inputs should be created in a way that allows their values to be overridden by the user (by supplying a named list of objects via `simInit(objects = ...)`) or by any other modules that produce these objects. For this, we rely on the `SpaDES.core::suppliedElsewhere` function, which detects if a given object has already been supplied by the user or if it will be supplied by another module.

Note that `suppliedElsewhere` does not know whether the module that supplies the object will be executed *before* the present module, as it is blind to module scheduling order. When modules are relatively simple and have an approximately linear flow of interdependencies, `SpaDES` is usually able to tell the order in which modules need to be executed. In more complex cases it is a good idea to pass a vector of module names to `simInit(loadOrder = ...)` defining the order of module execution.

Here's an example of how to do this (the commented instructions have been deleted):

```
.inputObjects <- function(sim) {
  if (!suppliedElsewhere("r")) {
    ## make template raster if not supplied elsewhere.
    sim$r <- rast(nrows = 100, ncols = 100, xmin = -50, xmax = 50,
                  ymin = -50, ymax = 50)
  }
  return(invisible(sim))
}
```

If we choose to supply the default `r` in `.inputObjects`, then we need to remove its creation from the `abundanceInit` function and add `r` to the metadata as an input. We have done this, so that `abundanceInit` only creates a storage list for the outputs:

```
abundanceInit <- function(sim) {
  ## create storage list of species abundance
  sim$abundRasters <- list()

  return(invisible(sim))
}
```

It is good practice to provide default input objects to all remaining modules, so that they can work stand-alone. We have done this below.

/!\ ATTENTION /!\

If *r* becomes an input with defaults it must be **added to the module metadata** inside an `expectsInput` call.

### 1.2.2.5 Additional module functions

Events can also rely on other functions that can either be sourced from other scripts, or defined at the end of the module script (e.g. usually before `.inputObjects`, although the order is irrelevant). This is the case for the species abundances generator function, which we coded in a separate script called `abundance_model.R`. Scripts with accessory functions like these go into module's R/ folder.

Functions should also be accompanied by metadata. Here we provide a description of the function, its parameters, returning value and what other package functions it relies on using the `roxygen2` documentation style (indicated by '#').

```
#' Accessory function to speciesAbundance module
#'
#' @param ras a raster layer used as template.
#' @return a fake abundance SpatRaster generated as a Gaussian map with scale = 100 and va
#' @importFrom NLMR nlm_mpd
#' @importFrom terra rast
abundance_model <- function(ras) {
  abund_ras <- NLMR::nlm_mpd(ncol = ncol(ras), nrow = nrow(ras),
    resolution = unique(res(ras)), roughness = 0.5, rand_dev = 100,
    rescale = TRUE, verbose = FALSE) |>
    rast()
  return(abund_ras)
}
```

### 1.2.3 Creating and adding additional modules: the *temperature* module

The order in which modules are first executed (i.e. their `init` events) can be automatically determined by inter-module dependencies (i.e. module inputs that are the outputs of other modules). If there are no inter-module dependencies this order is determined by the order in which the modules are listed in the `Part1_DummyModel.R` script, or via the `simInit(loadOrder = ...)` argument.

After the `init` event, the module execution order follows the order of events. This means that a module's events can be scheduled before and after another module's events within the same simulation time step. However, keep in mind that this can make the simulation flow hard to follow, debug and change when additional modules are added.

The second module we created generates yearly temperatures. Apart from different objects and functions names, this module also has the template raster `r` as required input object. Recall that `r` is created during the `.inputObjects` of the `speciesAbundance` module. When the two modules are linked, this object will not be created twice because `suppliedElsewhere("r")` will tell the `temperature` module that '`r`' will be supplied by another module. This may appear trivial in this example, but it can be extremely useful when inputs are heavy objects that require lengthy computations to be produced.

This is how we set up the `temperature.R` script looks like:

```
# Everything in this file gets sourced during simInit, and all functions and obj
# are put into the simList.
defineModule(sim, list(
  name = "temperature",
  description = "Temperature simulator",
  keywords = c("temperature", "gaussian", "spatial"),
  authors = structure(list(list(given = c("Ceres"), family = "Barros",
                                 role = c("aut", "cre"), email = "ceres.barros@ubc.ca")),
    childModules = character(0),
    version = list(temperature = "1.0.0"),
    timeframe = as.POSIXlt(c(NA, NA)),
    timeunit = "year",
    citation = list("citation.bib"),
    documentation = list("README.txt", "temperature.Rmd"),
    reqdPkgs = list("SpaDES.core (>=2.0.2)",
                    "terra", "ropensci/NLMR"),
    parameters = bindrows(
      #defineParameter("paramName", "paramClass", value, min, max, "parameter description"),
      defineParameter("simulationTimeStep", "numeric", 1, NA, NA,
                      "This describes the simulation time step interval"),
      defineParameter(".plotInitialTime", "numeric", 1, NA, NA,
                      "This describes the simulation time at which the first plot is generated"),
      defineParameter(".plotInterval", "numeric", 1, NA, NA,
                      "This describes the time interval between consecutive plots"))
  )
)

```

```

        "This describes the simulation time interval between plot events")
),
inputObjects = bindrows(
  #expectsInput("objectName", "objectClass", "input object description", sourceURL, ...)
  expectsInput("r", "SpatRaster", "Template raster")
),
outputObjects = bindrows(
  #createsOutput("objectName", "objectClass", "output object description", ...),
  createsOutput("tempRasters", "list", "List of raster layers of temperature at any gi
)
))

## event types
# - type `init` is required for initialiazation

doEvent.temperature = function(sim, eventTime, eventType, debug = FALSE) {
  switch(
    eventType,
    init = {
      ## do stuff for this event
      sim <- Init(sim)

      ## schedule future event(s)
      sim <- scheduleEvent(sim, eventTime = start(sim), moduleName = "temperature", event
      sim <- scheduleEvent(sim, eventTime = P(sim)$plotInitialTime, moduleName = "temper
          eventType = "tempPlot", eventPriority = .normal() + 0.5)
    },
    SimulTemp = {
      ## do stuff for this event
      sim <- update(sim)

      ## schedule future event(s)
      sim <- scheduleEvent(sim, eventTime = time(sim) + P(sim)$simulationTimeStep, module
          eventType = "SimulTemp")
    },
    tempPlot = {
      ## do stuff for this event
      sim <- plotting(sim)
    }
  )
}

```

```

## schedule future event(s)
sim <- scheduleEvent(sim, eventTime = time(sim) + P(sim)$plotInterval, mod
                      eventType = "tempPlot", eventPriority = .normal() + 0
),
warning(paste("Undefined event type: '", current(sim)[1, "eventType", with =
                         "' in module '", current(sim)[1, "moduleName", with = FALSE], "
)
return(invisible(sim))
}

## This is the 'init' event:
Init <- function(sim) {
  ## create storage list of species temperature
  sim$tempRasters <- list()

  return(invisible(sim))
}

## This is the temperature simulation event function
update <- function(sim) {
  ## Generate temperature - our "updated data"
  sim$tempRasters[[as.character(time(sim))]] <- temperature_model(ras = sim$r)

  return(invisible(sim))
}

## This is the plotting event funciton
plotting <- function(sim) {
  ## plot temperature
  plotTitle <- paste("Temperature\nat time",
                     names(sim$tempRasters)[length(sim$tempRasters)])
  tempPlot <- sim$tempRasters[[length(sim$tempRasters)]]
  Plot(tempPlot,
       title = plotTitle,
       new = TRUE, addTo = "tempPlot")

  return(invisible(sim))
}

```

```
.inputObjects <- function(sim) {
  if (!suppliedElsewhere("r")) {
    ## make template raster if not supplied elsewhere.
    sim$r <- rast(nrows = 100, ncols = 100, xmin = -50, xmax = 50, ymin = -50, ymax = 50)
  }
  return(invisible(sim))
}
```

Again, we added an accessory `temperature_model` function in a separate script `R/temperature_model.R`:

```
#' Accessory function to temperature module
#'
#' @param ras a raster layer used as template.
#' @return a fake temperature SpatRaster generated as a Gaussian map with scale = 100 and
#' @importFrom NLMR nlm_mpd
#' @importFrom terra rast
temperature_model <- function(ras) {
  temp_ras <- NLMR::nlm_mpd(ncol = ncol(ras), nrow = nrow(ras),
    resolution = unique(res(ras)), roughness = 0.5, rand_dev = 10,
    rescale = FALSE, verbose = FALSE) |>
    rast()
  return(temp_ras)
}
```

#### 1.2.4 Modules that depend on other modules: the *speciesTempLM* module

Our third and last module, `speciesTempLM`, will be used to run the statistical analysis at each year, after the abundances and temperatures are generated (`species` and **Temperature Linear Model**). Hence, it will depend on the outputs of the `speciesAbundance` and the `temperature` modules.

The interest of keeping the statistical analysis in a separate module lies on the fact that it allows us to easily swap and compare different statistical models to analyse our data if we want to.

It also allows for greater flexibility when it comes to **when** the statistical model is supposed to run. For example, we may want to fit it at every 5 years, instead of every year, using the previous 5 years of data. By having

the statistical analysis contained in its own module, we don't need to change other module scripts in order to make these changes.

Finally, we draw your attention to a few differences in this module's script before we see it:

- The **frequency** of the statistical analysis (and correspondent plots) will be determined by the parameter `statsTimestep`. This parameter also determines the number of data years to be used to fit the linear model. If `statsTimestep = 5`, the statistical analysis will use the precedent 5 years of data including the year in which the event is running (a total of 6 years of data);
- This module **requires inputs** that have no defaults in `.inputObjects`. They are specified in `inputObjects` part of `defineModule` - notice how we've respected the names, classes and description of the objects that come from the `speciesAbundance` and the `temperature` modules;
- We have **two additional functions** in a separate script (`R/linear_model_functions.R`): the function fitting the linear model and a plotting function.

Below is the full module script. Notice how the future events where scheduled to `P(sim)$statsTimestep + 0.1`, to force the statistical analyses to occur **after** the abundance and temperature rasters are ready.

```
# Everything in this file gets sourced during simInit, and all functions and obj
# are put into the simList.
defineModule(sim, list(
  name = "speciesTempLM",
  description = "Statistical analysis of species ~ temperature relationships usin
  keywords = c("linear model"),
  authors = structure(list(list(given = c("Ceres"), family = "Barros",
                                role = c("aut", "cre"), email = "ceres.barros@ubc.ca")),
    childModules = character(0),
    version = list(speciesTempLM = "1.0.0"),
    timeframe = as.POSIXlt(c(NA, NA)),
    timeunit = "year",
    citation = list("citation.bib"),
    documentation = list("README.txt", "speciesTempLM.Rmd"),
    reqdPkgs = list("SpaDES.core (>=2.0.2)",
                    "terra", "ggplot2", "data.table", "reshape2"),
    parameters = bindrows(
```

```

#defineParameter("paramName", "paramClass", value, min, max, "parameter description")
defineParameter("statsTimestep", "numeric", 1, NA, NA, "This describes the how often")
),
inputObjects = bindrows(
  #expectsInput("objectName", "objectClass", "input object description", sourceURL, ...),
  expectsInput("abundRasters", "list", "List of raster layers of species abundance at any given time"),
  expectsInput("tempRasters", "list", "List of raster layers of temperature at any given time"),
),
outputObjects = bindrows(
  #createsOutput("objectName", "objectClass", "output object description", ...),
  createsOutput("outputdata", "list", "List of dataframes containing species abundances"),
  createsOutput("outputLM", "list", "List of output yearly LMs (abundance ~ temperature)"),
  createsOutput("yrs", "numeric", "Vector of years used for statistical analysis")
)
))

## event types
# - type `init` is required for initialiazation

doEvent.speciesTempLM = function(sim, eventTime, eventType, debug = FALSE) {
  switch(
    eventType,
    init = {
      ## do stuff for this event
      sim <- statsInit(sim)

      ## schedule future event(s)
      sim <- scheduleEvent(sim, P(sim)$statsTimestep, "speciesTempLM",
                           "stats", eventPriority = .normal() + 2)
      sim <- scheduleEvent(sim, P(sim)$statsTimestep, "speciesTempLM",
                           "statsPlot", eventPriority = .normal() + 2.5)
    },
    stats = {
      ## do stuff for this event
      sim <- statsAnalysis(sim)

      ## schedule future event(s)
      sim <- scheduleEvent(sim, time(sim) + P(sim)$statsTimestep, "speciesTempLM",
                           "stats", eventPriority = .normal() + 2)
    }
  )
}

```

```

    },
  statsPlot = {
    ## do stuff for this event
    sim <- statsPlot(sim)

    ## schedule future event(s)
    sim <- scheduleEvent(sim, time(sim) + P(sim)$statsTimestep, "speciesTempLM"
                           "statsPlot", eventPriority = .normal() + 2.5)
  },
  warning(paste("Undefined event type: '", current(sim)[1, "eventType", with =
    "' in module '", current(sim)[1, "moduleName", with = FALSE], "
  )
  return(invisible(sim))
}

## template initialization
statsInit <- function(sim) {
  ## create outputs storage lists
  sim$outputLM <- list()

  return(invisible(sim))
}

## Statistical analysis event
statsAnalysis <- function(sim) {
  ## get all species abundances data available
  abundData <- as.data.table(rast(sim$abundRasters))
  abundData[, pixID := 1:nrow(abundData)]
  abundData <- melt.data.table(abundData, id.var = "pixID",
                                variable.name = "year", value.name = "abund")
  abundData[, year := as.numeric(sub("X", "", year))]

  ## get all temperature data available
  tempData <- as.data.table(rast(sim$tempRasters))
  tempData[, pixID := 1:nrow(tempData)]
  tempData <- melt.data.table(tempData, id.var = "pixID",
                                variable.name = "year", value.name = "temp")
  tempData[, year := as.numeric(sub("X", "", year))]
}

```

```

## merge per year
setkey(abundData, pixID, year)
setkey(tempData, pixID, year)
sim$outputdata <- abundData[tempData]

sim$outputLM[[as.character(time(sim))]] <- linearModel(Data = sim$outputdata)
return(invisible(sim))
}

## Plotting event
statsPlot <- function(sim) {
  model <- sim$outputLM[[as.character(time(sim))]]

  modelPlot <- ggplot(sim$outputdata) +
    geom_point(aes(x = temp, y = abund)) +
    geom_abline(intercept = model$coefficients["(Intercept)"],
                slope = model$coefficients["temp"], size = 2, col = "blue") +
    theme_bw() +
    labs(x = "Temp.", y = "Species abundance")

  plotTitle <- paste("abundance ~ temperature\n",
                     "years", range(sim$outputdata$year)[1],
                     "to", range(sim$outputdata$year)[2])
  Plot(modelPlot,
       title = plotTitle,
       new = TRUE, addTo = "modelPlot")

  return(invisible(sim))
}

.inputObjects <- function(sim) {
  # Any code written here will be run during the simInit for the purpose of creating
  # any objects required by this module and identified in the inputObjects element of def
  # This is useful if there is something required before simulation to produce the module
  # object dependencies, including such things as downloading default datasets, e.g.,
  # downloadData("LCC2005", modulePath(sim)).
  # Nothing should be created here that does not create a named object in inputObjects.
  # Any other initiation procedures should be put in "init" eventType of the doEvent func
  # Note: the module developer can check if an object is 'suppliedElsewhere' to
}

```

```

# selectively skip unnecessary steps because the user has provided those input#
# simInit call, or another module will supply or has supplied it. e.g.,
# if (!suppliedElsewhere('defaultColor', sim)) {
#   sim$map <- Cache(prepInputs, extractURL('map')) # download, extract, load #
# }

#cacheTags <- c(currentModule(sim), "function:.inputObjects") ## uncomment this
dPath <- asPathgetOption("reproducible.destinationPath", dataPath(sim)), 1)
message(currentModule(sim), ": using dataPath '", dPath, "'.")

# ! ----- EDIT BELOW ----- !
# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

```

And the script with the accessory functions:

```

## Accessory functions to speciesTempLM module

#' Accessory function to speciesTempLM module that calculates a
#' linear regression between species abundances and temperature
#'
#' @param Data a data.frame or data.table that contains an \code{abund}
#'   column and a \code{temp} column with abundance and temperature values
#'   in each location, respectively.
#' @return a linear model (\code{lm}) object fitted with the formula:
#'   \code{abund ~ temp}

linearModel <- function(Data) {
  lm1 <- lm(abund ~ temp, data = Data)
  return(lm1)
}

```

---

### 1.2.5 Simulation

#### 1.2.5.1 Simulation setup in a “global” script

We can now go back to our `Part1_DummyModel.R` script and set the simulation up.

The function `simInit` needs a few arguments listing simulation folder directories, parameters, simulation times, modules and, optionally, input objects supplied by the user. `simInit` will prepare a simulation object that can later be run by the `spades` function:

- The first list, `modules`, contains modules we want to activate.
- `times` is a named list containing the start and end times of the simulation and what time units we’re working with (with “start” and “end” being the list `names`). It thus defines the length of the simulation. It is important that the start and ending times are defined in decimals, because SpaDES allows decomposing time units into smaller fractions.
- `parameters` is a named list of named lists, containing parameters values passed to each module. Note that because the module metadata will (or should) contain default parameter values, here we pass only parameters which we want to change with respect to their defaults. For instance, `.plotInterval` is used and defined in the `speciesAbundance` and `temperature` modules, but not passed to the `simInit` function because we want to use the default value. As a developer providing a reproducible example, we may also chose to list important and useful parameters, even if the value is the same as the default. Here we chose to list `.plotInitialTime` (a parameter used and defined in the `speciesAbundance` and `temperature` modules), but provide the default value (we experimenting with it by changing its value in the `Part1_DummyModel.R`).
- `paths` contains the folder directory paths that we set earlier.

```
## list the modules to use
simModules <- list("speciesAbundance", "temperature", "speciesTempLM")

## Set simulation and module parameters
simTimes <- list(start = 1, end = 10, timeunit = "year")
simParams <- list(speciesAbundance = list(simulationTimeStep = 1,
    .plotInitialTime = 1), temperature = list(simulationTimeStep = 1,
```

```
.plotInitialTime = 1), speciesTempLM = list(statsTimestep = 5))

## make a list of directory paths
simPaths <- getPaths()

## Simulation setup
mySim <- simInit(times = simTimes, params = simParams, modules = simModules,
  paths = simPaths)
```

Finally, we highlight that `simInit` also executes all `.inputObjects` functions, and schedules the `init` events, but does not execute them:

```
events(mySim)
```

### 1.2.5.2 Checking the simulation setup

Before starting the simulations we should check if the modules were linked correctly.

#### Module diagram

`moduleDiagram` (Figure 1.2 ) is a useful function that shows module inter-dependencies as a network diagram. The direction of the arrows indicates an output to input flow. You can see that `speciesAbundance` and `temperature` inputs (specifically our ‘r‘ raster) are supplied by an external source (*INPUT*) - the user or `.inputObjects`. Whereas the inputs to the `speciesTempLM` module are outputs of the `speciesAbundance` and `temperature` modules.

```
moduleDiagram(mySim)
```

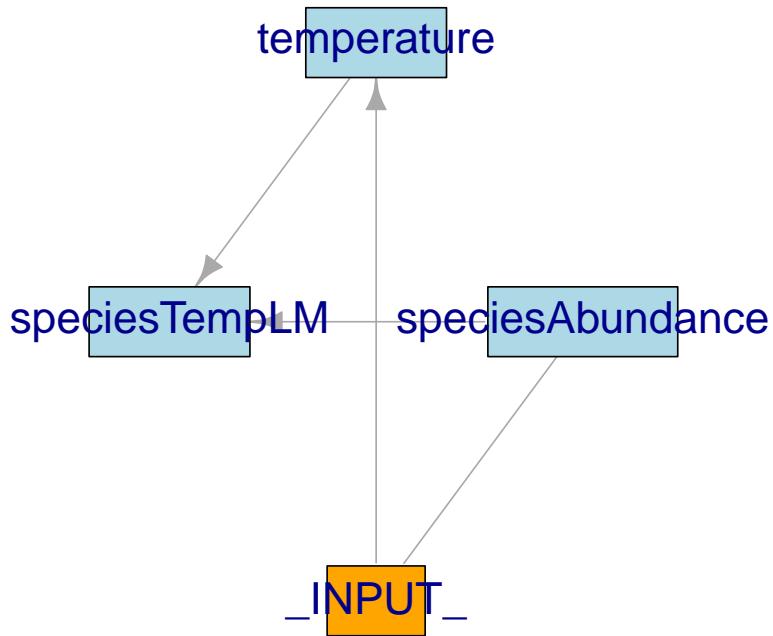


Figure 1.2: Diagram of module connections.

### Object diagram

`objectDiagram` (Figure 1.3) provides another way of checking module linkages. It explicitly shows module inter-dependencies by depicting the objects that establish links between modules.

```
objectDiagram(mySim)
```

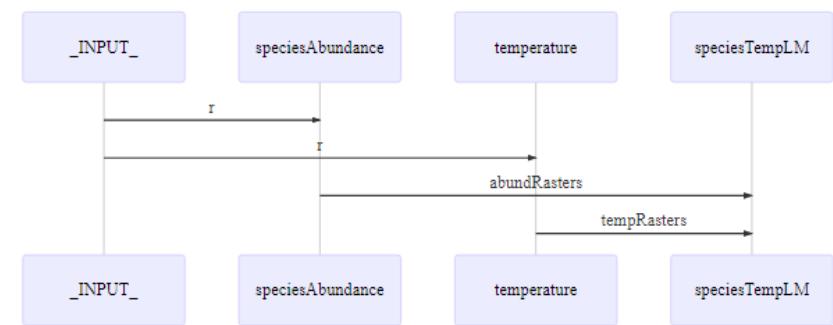


Figure 1.3: Module diagram showing module inter-dependencies with object names.

### 1.2.5.3 Running SpaDES

We run the simulation using the `spades` function, which takes the output of the `simInit`, executes the already scheduled `init` events, which schedule the remainder of the events. We passed `debug = TRUE` so that `spades` prints the events as they are being executed. In case something fails, this helps diagnosing where the issue occurred.

```
## run simulation
dev() # on Windows and Mac, this opens external device if using Rstudio, it is faster
clearPlot()
mySim2 <- spades(mySim, debug = TRUE)
```

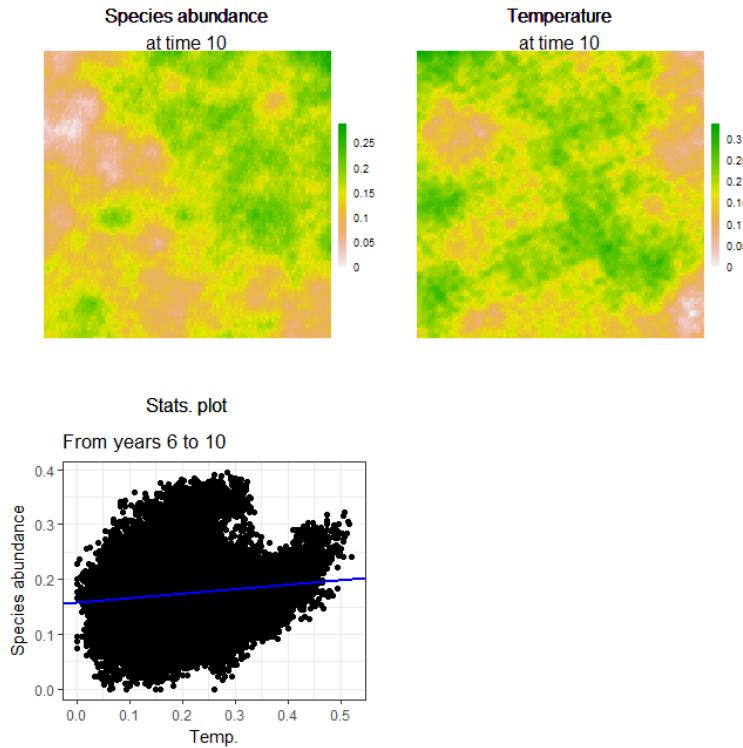


Figure 1.4: Simulation plots: Final plot of the simulation

We suggest experimenting with changing parameter values and trying to create and add other modules to further explore all the `SpaDES` flexibility. The more complex the project gets, the more advantageous it is to use

SpaDES to turn modules *on* or *off*, swapping modules to run, e.g., different statistical analyses, or to include different data.

---

### 1.2.6 Additional notes

SpaDES is an extremely powerful package, whose potential goes well beyond what has been discussed in this dummy example. If you want to explore it further, we recommend following [Chapter 2](#) for a more realistic (but still simple) SpaDES application.

Also, do go to the [SpaDES webpage](#) to find further information about the platform, as well as upcoming workshops and publications and to the [Predictive Ecology Github repository](#) to see all the SpaDES modules and SpaDES-related packages that we maintain at the [Predictive Ecology Lab](#).

---

**Happy SpaDESing!**

## Chapter 2

# A more realistic example of SpaDES

Authors: Ceres Barros, Alex M. Chubaty

In [Chapter 1](#) of this guide, we described how to create new **SpaDES** modules, their different components, how to link different modules and how to set up and run a simulation.

Here, we assume that you are familiar with these steps, but go further in showing important **SpaDES** features that facilitate many of the steps common to most ecological modelling exercises. For the sake of simplicity, we focus our example on projecting a species' distribution as a function of climate covariates. Yet, the true power of **SpaDES** is more evident when using complex dynamic simulation models parametrised using large datasets and ran across large spatial areas.

This example is broken into four main parts: [Module creation and coding](#); [Running the model](#); [Caching](#); and [Best practices](#). By no means does it cover caching or best practices in full, as each of these topics is very extensive, but it highlights some of their essentials in **SpaDES** and from our own experience.

The full R script used to run the example can be found in [Appendix B](#).

## 2.1 The example: projecting species distribution shifts under climate change

Species distribution models (SDMs) have been widely used in ecology to predict how species presences and absences across a landscape may change under changing environmental conditions. As a result, there are several R packages that have been built with this in mind (e.g. `dismo` Hijmans et al. 2021; `biomod2` Thuiller et al. 2021) and many ecologists do these analyses exclusively in R.

Often, these analyses are run only once for a given set of species, baseline and projected environmental conditions, and researchers will have a few R scripts (or maybe just one longer script) that load the data into R, do any necessary pre-processing steps, fit the models and run species distribution projections. The usefulness of `SpaDES` comes when we want an automated and standardized workflow that can be easily updated with new data and adapted with new algorithms. `SpaDES` provides a common standard and a modular approach to modelling that facilitates expanding, debugging and sharing code, but also various tools that bring many well-known best practices from computer- and data-science workflows (including reproducible, modular workflows, and caching), to the realm of ecological modelling, so that they can be used by non-computer-scientists with minimal learning. In an SDM project this means that updating data and algorithms, and automating iterative forecasting become easier and less prone to errors. When `SpaDES` modules are open and shared, this also expands a potential pool of users who can themselves help improve the code.

---

## 2.2 Module creation and coding

With the above in mind, in this example we created three modules that source and pre-process data ('data modules') and a module that fits an SDM and iteratively forecasts species distributions (we call it a 'simulation module', although the simulation only involves predicting from a statistical model). The idea is that we could, for instance, provide different data sources to one of the data modules and only update the parts of the simulation that are affected by this change (i.e. presumably the other data module steps will not be affected). Or, we could develop a second simulation module using a different SDM approach and swap the two modules to inspect

which provides better predictions.

Our data modules are *speciesAbundanceData* and *climateData*. The simulation module is *projectSpeciesDist*. We start by creating an .R script to set up and control the simulation. In this example this script is called *Part2\_SDMs.R*.

The script begins with a few lines of code that ensure a few packages are installed and loaded (see [Reproducible package installation](#)). It then defines the necessary folder directories for the simulation and creates the modules in the *modules/* folder.

```
options(repos = c("https://predictiveecology.r-universe.dev/",
                 CRAN = "https://cloud.r-project.org"))

if (getRversion() < "4.2.1") {
  warning(paste("dismo::maxent may create a fatal error",
                "when using R version < v4.2.1 and from RStudio.\n",
                "Please upgrade R, or run this script outside of RStudio.\n",
                "See https://github.com/rspatial/dismo/issues/13"))
}

## decide where you're working
mainPath <- file.path("~/SpaDES4Dummies_Part2")
pkgPath <- file.path(mainPath, "packages", version$platform,
                      paste0(version$major, ".", strsplit(version$minor, "[.]")[[1]][1]))
dir.create(pkgPath, recursive = TRUE)
.libPaths(pkgPath, include.site = FALSE) ## install packages in project library (proj-lib)

if (!"remotes" %in% installed.packages(lib.loc = pkgPath))
  install.packages("remotes")

if (!"Require" %in% installed.packages(lib.loc = pkgPath) ||
    packageVersion("Require", lib.loc = pkgPath) < "0.3.1.9015") {
  remotes::install_github("PredictiveEcology/Require@2788b023ad191c29346ef8c64df71b937be3",
                         upgrade = FALSE)
}

## Notes:
## 1) if you are working from RStudio and have an older version of base packages like `Rcyl`  

## (and others) installed, you may need to run the following lines (and code above) dire
```

```

## in order to update these base packages
## 2) Please ensure the appropriate Rtools version is installed (see)

## there seems to be a problem with `ragg` and a forced install solves it
if (!"ragg" %in% installed.packages(lib.loc = pkgPath)) {
  install.packages("ragg")
}

Require::Require(c("PredictiveEcology/SpaDES.project@transition (HEAD)",
                  "PredictiveEcology/SpaDES.core@master (HEAD)",
                  ## these will be needed later on:
                  "ggpubr",
                  "geodata",
                  "SpaDES.tools",
                  "PredictiveEcology/SpaDES.experiment@75d917b70b892802fed0bbdb2",
                  require = FALSE, ## don't load packages yet
                  upgrade = FALSE, standAlone = TRUE)

Require::Require("SpaDES.core", install = FALSE) ## load only
setPaths(cachePath = file.path(mainPath, "cache"),
         inputPath = file.path(mainPath, "inputs"),
         modulePath = file.path(mainPath, "modules"),
         outputPath = file.path(mainPath, "outputs"))

simPaths <- getPaths() ## check that this is what you wanted

## Let's create a self-contained module that will simulate the species' abundance
if (!dir.exists(file.path(simPaths$modulePath, "speciesAbundanceData"))) {
  newModule(name = "speciesAbundanceData", path = simPaths$modulePath)
}

if (!dir.exists(file.path(simPaths$modulePath, "climateData"))) {
  newModule(name = "climateData", path = simPaths$modulePath)
}

if (!dir.exists(file.path(simPaths$modulePath, "projectSpeciesDist"))) {
  newModule(name = "projectSpeciesDist", path = simPaths$modulePath)
}

```

Notice how we protect the `newModule` call with an `if` statement that first detects whether the module directory exists already. This is necessary to prevent overwriting existing modules should this script be run a second time in the same location (see [Protect yourself and others from common mistakes/problems](#)).

`setPaths` will create the project folder directories in case they do not exist (no overwriting occurs in case they do).

Finally, note that we do not load any R packages yet, as we will later use `Require` to make sure all module dependencies are installed before running the simulation (see [Reproducible package installation](#)). Because `Require` may attempt to install missing packages and because installing packages should be done in a clean R session, we will only load any packages after all the installation steps are complete.

### 2.2.1 Data modules

The next two sections show our two data modules .R scripts. We assume you are already familiar with the different parts of a module .R script; if not, see [Chapter 1](#). We do not discuss the module .Rmd files, which should document each module in detail (see [Module documentation – module .Rmd]).

#### 2.2.1.1 *speciesAbundanceData* module:

This module downloads freely available spatial layers of *Picea glauca* percent cover (% cover) across Canada and pre-processes them to match a user-supplied study area raster. We use the new `terra` package throughout this example, since the `raster` package will soon be discontinued.

The `prepInputs` function downloads the % cover layer from the Canadian National Forest Inventory data server using the URL supplied by `sppAbundURL` and processes it to match the study area raster (`studyAreaRas`) supplied by the user. The module then outputs *Picea glauca* % cover as a raster (`sppAbundanceRas`) and as a `data.table` (`sppAbundanceDT`). The `data.table` contains added information about the year of the simulation during which the data should be used (here, only the first year when SDM fitting happens).

We export species % cover in two formats (a raster and a table) for demonstrational purposes, but also because we could envision that this model (i.e. group of modules) could save the species distribution projections for

several points in time in a more compact format of a `data.table` – large raster layers can consume a considerable amount of disk space (see [Coding for the future](#)).

```
## Everything in this file and any files in the R directory are sourced during `source`.
## all functions and objects are put into the `simList`.
## To use objects, use `sim$xxx` (they are globally available to all modules).
## Functions can be used inside any function that was sourced in this module;
## they are namespaced to the module, just like functions in R packages.
## If exact location is required, functions will be: `sim$.mods$<moduleName>$Function`.
defineModule(sim, list(
  name = "speciesAbundanceData",
  description = paste("Data module to prepare tree species cover data for species abundance",
                      "Defaults to using Canadian National Forest Inventory data"),
  keywords = c("minimal SpaDES example", "species distribution model"),
  authors = structure(list(list(given = c("Ceres"), family = "Barros", role = c("author"),
                                 email = "ceres.barros@ubc.ca", comment = NULL)),
    childModules = character(0),
    version = list(speciesAbundanceData = "1.0.0"),
    timeframe = as.POSIXlt(c(NA, NA)),
    timeunit = "year",
    citation = list("citation.bib"),
    documentation = list("README.md", "speciesAbundanceData.Rmd"), ## same file
    reqdPkgs = list("SpaDES.core (>=2.0.2)",
                    "httr", "terra", "ggplot2", "rasterVis"),
    parameters = bindrows(
      #defineParameter("paramName", "paramClass", value, min, max, "parameter description"),
      defineParameter("sppAbundURL", "character",
                     paste0("https://ftp.maps.canada.ca/pub/nrcan_rncan/Forests_Fores",
                           "canada-forests-attributes_attributes-forests-canada/",
                           "2001-attributes_attributes-2001/",
                           "NFI_MODIS250m_2001_kNN_Species_Pice_Gla_v1.tif"), NA,
                     paste("URL where the first SpatRaster of species abundance resides"),
                     "This will be the abundance data used to fit the species abundance model",
                     "Defaults to *Picea glauca* percent cover across Canada",
                     "(from Canadian National Forest Inventory forest attributes dataset)"),
      defineParameter(".plots", "character", "screen", NA, NA,
                     "Used by Plots function, which can be optionally used here"),
      defineParameter(".plotInitialTime", "numeric", start(sim), NA, NA,
                     "Describes the simulation time at which the first plot event occurs"))
  )
)
```

```

defineParameter(".plotInterval", "numeric", NA, NA, NA,
               "Describes the simulation time interval between plot events."),
defineParameter(".saveInitialTime", "numeric", NA, NA, NA,
               "Describes the simulation time at which the first save event should occur"),
defineParameter(".saveInterval", "numeric", NA, NA, NA,
               "This describes the simulation time interval between save events."),
defineParameter(".studyAreaName", "character", NA, NA, NA,
               "Human-readable name for the study area used. If NA, a hash of studyAreaRas will be used"),
## .seed is optional: `list('init' = 123)` will `set.seed(123)` for the `init` event
defineParameter(".seed", "list", list(), NA, NA,
               "Named list of seeds to use for each event (names)."),
defineParameter(".useCache", "logical", FALSE, NA, NA,
               "Should caching of events or module be used?")
),
inputObjects = bindrows(
  #expectsInput("objectName", "objectClass", "input object description", sourceURL, ...),
  expectsInput("studyAreaRas", objectClass = "SpatRaster",
               desc = "A binary raster of the study area")
),
outputObjects = bindrows(
  #createsOutput("objectName", "objectClass", "output object description", ...),
  createsOutput("sppAbundanceDT", "data.table",
               desc = paste("Species abundance data from `sppAbundanceRas`, with columns",
                           "'x'", 'y', 'sppAbund' and 'year' (an integer matching the number of years in the input raster),
                           "names(`sppAbundanceRas`).")),
  createsOutput("sppAbundanceRas", "SpatRaster",
               desc = paste("A species abundance layer used to fit a species distribution model",
                           "at the start of the simulation. Layers named as:",
                           "paste('year', start(sim):end(sim), sep = '_')). Data obtained from sppAbundanceDT"))
)
))

## event types
# - type `init` is required for initialization

doEvent.speciesAbundanceData = function(sim, eventTime, eventType, debug = FALSE) {
  switch(
    eventType,

```



```

        overwrite = TRUE,
        cacheRepo = cachePath(sim))
})

options(opts)

if (is(sppAbundanceRas, "RasterLayer")) {
  sppAbundanceRas <- terra::rast(sppAbundanceRas)
}

names(sppAbundanceRas) <- paste("year", time(sim), sep = "_")
sppAbundanceDT <- as.data.table(as.data.frame(sppAbundanceRas, xy = TRUE, cells = TRUE))
sppAbundanceDT[, year := as.integer(sub("year_", "", names(sppAbundanceRas)))]
setnames(sppAbundanceDT, "year_1", "sppAbund")

## export to sim
sim$sppAbundanceRas <- sppAbundanceRas
sim$sppAbundanceDT <- sppAbundanceDT

return(invisible(sim))
}

## Plotting event function
abundancePlot <- function(sim) {
  ## plot species abundance
  Plots(sim$sppAbundanceRas, fn = plotSpatRaster, types = P(sim)$plots,
        usePlot = TRUE, filename = file.path(outputPath(sim), "figures", "speciesAbundance"),
        plotTitle = "Species abundance data", xlab = "Longitude", ylab = "Latitude")

  return(invisible(sim))
}

.inputObjects <- function(sim) {
  #cacheTags <- c(currentModule(sim), "function:.inputObjects") ## uncomment this if Cache
  dPath <- asPathgetOption("reproducible.destinationPath", dataPath(sim)), 1)
  message(currentModule(sim), ": using dataPath '", dPath, "'.")

# ! ----- EDIT BELOW ----- !

```

```

if (!suppliedElsewhere(sim$studyAreaRas)) {
  ## code check: did the user supply a study area?
  stop("Please supply a 'studyAreaRas' SpatRaster")
}

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

```

### 2.2.1.2 *climateData* module:

This module downloads and processes freely available spatial layers of four bioclimatic variables used to fit the SDM of *Picea glauca* in the study area.

The module uses a different way to download data. It relies on two input `data.tables` that contain the URLs for each climate covariate, one for baseline conditions, the other for projected climate conditions, both containing information about when each layer should be used during the simulation (the “year” column).

We have only supplied one set of data sources for default baseline climate conditions (`baselineClimateURLs`) and for climate projections (`projClimateURLs`), all of which are downloaded from WorldClim at 2.5 minutes resolution. The baseline climate data correspond to the 1970-2000 period Fick and Hijmans (2017), which aligns well with the species % cover data year (2001). The climate projections were obtained for 2021-2040, 2041-2060, 2061-2080 and 2081-2100, from CMIP6 downscaled future projections using the CanESM5 model (Swart et al. 2019) under the SSP 585 climate scenario.

We encourage providing different (or additional) URLs referring to projections for other climate periods, other climate models and other climate scenarios (see [WorldClim](#) for a list of climate projections).

If providing other URLs to obtain different climate data, pay special attention to the “year” column of `projClimateURLs` – the URLs need to correspond to the simulation year during which they will be used (not necessarily the actual climate year, unless the simulation years follow the same numbering).

Like in the `speciesAbundanceData` module, the `prepInputs` function pro-

cesses the climate layers to match the study area raster (`studyAreaRas`) and compiles all climate data in the `climateDT` object and as raster layer objects (`baselineClimateRas` and `projClimateRas`) – the module's outputs.

```
## Everything in this file and any files in the R directory are sourced during `simInit()`
## all functions and objects are put into the `simList`.
## To use objects, use `sim$xxx` (they are globally available to all modules).
## Functions can be used inside any function that was sourced in this module;
## they are namespaced to the module, just like functions in R packages.
## If exact location is required, functions will be: `sim$.mods$<moduleName>$FunctionName`

defineModule(sim, list(
  name = "climateData",
  description = paste("Data module to prepare climate data for species distribution model",
                      "Defaults to using bioclimatic variables from Worldclim."),
  keywords = c("minimal SpaDES example", "species distribution model"),
  authors = structure(list(list(given = c("Ceres"), family = "Barros", role = c("aut", "c"),
  childModules = character(0),
  version = list(climateData = "1.0.0"),
  timeframe = as.POSIXlt(c(NA, NA)),
  timeunit = "year",
  citation = list("citation.bib"),
  documentation = list("README.md", "climateData.Rmd"), ## same file
  reqdPkgs = list("SpaDES.core (>=2.0.2)",
                  "ggplot2", "rasterVis", "terra", "data.table"),
  parameters = bindrows(
    #defineParameter("paramName", "paramClass", value, min, max, "parameter description")
    defineParameter(".plots", "character", "screen", NA, NA,
                   "Used by Plots function, which can be optionally used here"),
    defineParameter(".plotInitialTime", "numeric", start(sim), NA, NA,
                   "Describes the simulation time at which the first plot event should occur"),
    defineParameter(".plotInterval", "numeric", NA, NA, NA,
                   "Describes the simulation time interval between plot events."),
    defineParameter(".saveInitialTime", "numeric", NA, NA, NA,
                   "Describes the simulation time at which the first save event should occur"),
    defineParameter(".saveInterval", "numeric", NA, NA, NA,
                   "This describes the simulation time interval between save events."),
    defineParameter(".studyAreaName", "character", NA, NA, NA,
                   "Human-readable name for the study area used. If NA, a hash of studyAreaRas"),
    ## .seed is optional: `list('init' = 123)` will `set.seed(123)` for the `init` event
    defineParameter(".seed", "list", list(), NA, NA,
```

```

    "Named list of seeds to use for each event (names)."),
defineParameter(".useCache", "logical", FALSE, NA, NA,
               "Should caching of events or module be used?")
),
inputObjects = bindrows(
  #expectsInput("objectName", "objectClass", "input object description", source)
  expectsInput("baselineClimateURLs", "data.table",
               desc = paste("A table with columns 'vars', 'URL', 'targetFile' and
                            "variable names, URLs and raster file names of each
                            "used in the species distribution models. Year is the
                            "simulation (not the reference climate period). Default
                            "'bio1', 'bio4', 'bio12' and 'bio15' bioclimatic variables
                            "climate period, at 2.5 minutes.")),
  expectsInput("projClimateURLs", "data.table",
               desc = paste("Same as `baselineClimateURLs` but referring to projections
                            "Variable names in 'vars' need to be the same as in `baselineClimateURLs`
                            "and P(sim)$projClimateURLs. Years should correspond to the
                            "Defaults to 2081-2100 projections using the CanESM5 model
                            "SSP 585 climate scenario, at 2.5 minutes, obtained from
                            "the R package 'downscaleR'.")),
  expectsInput("studyAreaRas", objectClass = "SpatRaster",
               desc = "A binary raster of the study area")
),
outputObjects = bindrows(
  #createsOutput("objectName", "objectClass", "output object description", ...)
  createsOutput("climateDT", "data.table",
               desc = paste("A data.table with as many columns as the climate
                            "used in the species distribution model and 'year'
                            "the simulation year to which the data corresponds"))
  createsOutput("baselineClimateRas", "SpatRaster",
               desc = paste("Baseline climate layers obtained from `baselineClimateURLs`"))
  createsOutput("projClimateRas", "SpatRaster",
               desc = paste("Baseline climate layers obtained from `projClimateURLs`"))
)
))

## event types
# - type `init` is required for initialization

doEvent.climateData = function(sim, eventTime, eventType, debug = FALSE) {

```

```

switch(
  eventType,
  init = {
    ## do stuff for this event
    sim <- climateInit(sim)

    ## schedule future event(s)
    sim <- scheduleEvent(sim, eventTime = P(sim)$plotInitialTime,
                          moduleName = "climateData", eventType = "climPlot",
                          eventPriority = .normal())
  },
  climPlot = {
    ## do stuff for this event
    sim <- climatePlot(sim)
  },
  warning(paste("Undefined event type: '", current(sim)[1, "eventType", with = FALSE],
                "' in module '", current(sim)[1, "moduleName", with = FALSE], "'", sep =
  )
)
return(invisible(sim))
}

## event functions
# - keep event functions short and clean, modularize by calling subroutines from section

## Initialisation Event function
climateInit <- function(sim) {
  ## GET BASELINE DATA
  ## make a vector of archive (zip) file names if the url points to one.
  archiveFiles <- sapply(sim$baselineClimateURLs$URL, function(URL) {
    if (grepl("\\\\.zip$", basename(URL))) {
      basename(URL)
    } else {
      NULL
    }
  }, USE.NAMES = FALSE)

  ## check that baseline climate data only has one year value
  if (length(unique(sim$baselineClimateURLs$year)) != 1) {
    stop(paste("'baselineClimateURLs' should all have the same 'year' value,",
```

```

    "corresponding to the first year of the simulation"))
}

## download data - prepInputs does all the heavy-lifting of dowloading and pre
baselineClimateRas <- Cache(Map,
  f = prepInputs,
  url = sim$baselineClimateURLs$URL,
  targetFile = sim$baselineClimateURLs$targetFile,
  archive = archiveFiles,
  MoreArgs = list(
    fun = "terra::rast",
    overwrite = TRUE,
    projectTo = sim$studyAreaRas,
    cropTo = sim$studyAreaRas,
    maskTo = sim$studyAreaRas,
    rasterToMatch = sim$studyAreaRas,
    cacheRepo = cachePath(sim)),
  cacheRepo = cachePath(sim))

names(baselineClimateRas) <- paste0(sim$baselineClimateURLs$vars, "_year", sim$baselineClimateURLs$year)

## make a stack
baselineClimateRas <- rast(baselineClimateRas)

## make a data.table
baselineClimateData <- as.data.table(as.data.frame(baselineClimateRas, xy = TRUE))
setnames(baselineClimateData, sub("_year.*", "", names(baselineClimateData)))
baselineClimateData[, year := unique(sim$baselineClimateURLs$year)]

## GET PROJECTED DATA
## make a vector of archive (zip) file names if the url points to one.
archiveFiles <- lapply(sim$projClimateURLs$URL, function(URL) {
  if (grepl("\\\\.zip$", basename(URL))) {
    basename(URL)
  } else {
    NULL
  }
})

## download data - prepInputs does all the heavy-lifting of dowloading and pre

```

```

## workaround Mar 30th 2022 cache issue with terra.
projClimateRas <- Cache(Map,
                        f = prepInputs,
                        url = sim$projClimateURLs$URL,
                        targetFile = sim$projClimateURLs$targetFile,
                        archive = archiveFiles,
                        MoreArgs = list(
                            overwrite = TRUE,
                            fun = "raster::stack",
                            projectTo = sim$studyAreaRas,
                            cropTo = sim$studyAreaRas,
                            maskTo = sim$studyAreaRas,
                            rasterToMatch = sim$studyAreaRas,
                            cacheRepo = cachePath(sim)),
                        cacheRepo = cachePath(sim))
if (any(sapply(projClimateRas, function(x) is(x, "RasterLayer") | is(x, "RasterStack")))
    projClimateRas <- lapply(projClimateRas, terra::rast)
}

## these rasters are different. The tif file contains all the variables in different layers
## so, for each variable, we need to keep only the layer of interest
projClimateRas <- mapply(function(stk, var) {
    lyr <- which(sub(".*_", "BIO", names(projClimateRas[[1]])) == var)
    return(stk[[lyr]])
}, stk = projClimateRas, var = sim$projClimateURLs$vars)
names(projClimateRas) <- paste0(sim$projClimateURLs$vars, "_year", sim$projClimateURLs$vars)

## make a stack
projClimateRas <- rast(projClimateRas)

## make a data.table
projClimateData <- as.data.table(as.data.frame(projClimateRas, xy = TRUE, cells = TRUE))

## melt so that year is in a column
projClimateDataMolten <- lapply(unique(sim$projClimateURLs$vars), function(var, projClimateData) {
    cols <- grep(paste0(var, "_year"), names(projClimateData), value = TRUE)
    idCols <- names(projClimateData)[!grepl("_year", names(projClimateData))]
    moltenDT <- melt(projClimateData, id.vars = idCols, measure.vars = cols,
}

```

```

            variable.name = "year", value.name = var)
moltenDT[, year := sub(paste0(var, "_year"), "", year)]
moltenDT[, year := as.integer(year)]
return(moltenDT)
}, projClimateData = projClimateData)

idCols <- c(names(projClimateData)[!grepl("_year", names(projClimateData))], "year")
## set keys for merge
projClimateDataMolten <- lapply(projClimateDataMolten, function(DT, cols) {
  setkeyv(DT, cols = cols)
  return(DT)
}, cols = idCols)

projClimateData <- Reduce(merge, projClimateDataMolten)

## bind the two data.tables
if (!identical(sort(names(baselineClimateData)), sort(names(projClimateData))))
  stop("Variable names in `projClimateURLs` differ from those in `baselineClimateData`")
}

## check
if (!compareGeom(baselineClimateRas, projClimateRas, res = TRUE, stopOnError = TRUE))
  stop("`baselineClimateRas` and `projClimateRas` do not have the same raster properties")
}

## export to sim
sim$baselineClimateRas <- baselineClimateRas
sim$projClimateRas <- projClimateRas
sim$climateDT <- rbindlist(list(baselineClimateData, projClimateData), use.names = TRUE)

return(invisible(sim))
}

## Plotting event function
climatePlot <- function(sim) {
  ## plot climate rasters
  allRasters <- rast(list(sim$baselineClimateRas, sim$projClimateRas))
  lapply(sim$baselineClimateURLs$vars, function(var, allRasters) {
    lrs <- grep(paste0(var, "_"), names(allRasters))
  })
}

```

```

file_name <- paste0("climateRas_", var)
Plots(allRasters[[lrs]],
      fn = plotSpatRasterStk, types = P(sim)$plots,
      usePlot = FALSE,
      filename = file.path(outputPath(sim), "figures", file_name),
      xlab = "Longitude", ylab = "Latitude")
}, allRasters = allRasters)

return(invisible(sim))
}

.inputObjects <- function(sim) {
#cacheTags <- c(currentModule(sim), "function:.inputObjects") ## uncomment this if Cache
dPath <- asPathgetOption("reproducible.destinationPath", dataPath(sim)), 1)
message(currentModule(sim), ": using dataPath '", dPath, "'.")

# ! ----- EDIT BELOW ----- !
if (!suppliedElsewhere(sim$studyAreaRas)) {
  ## code check: did the user supply a study area?
  stop("Please supply a 'studyAreaRas' SpatRaster")
}

if (!is(sim$studyAreaRas, "SpatRaster")) {
  sim$studyAreaRas <- rast(sim$studyAreaRas)
}

if (!suppliedElsewhere(sim$baselineClimateURLs)) {
  sim$baselineClimateURLs <- data.table(
    vars = c("BI01", "BI04", "BI012", "BI015"),
    URL = c("https://biogeo.ucdavis.edu/data/worldclim/v2.1/base/wc2.1_2.5m_bio.zip",
            "https://biogeo.ucdavis.edu/data/worldclim/v2.1/base/wc2.1_2.5m_bio.zip",
            "https://biogeo.ucdavis.edu/data/worldclim/v2.1/base/wc2.1_2.5m_bio.zip",
            "https://biogeo.ucdavis.edu/data/worldclim/v2.1/base/wc2.1_2.5m_bio.zip"),
    targetFile = c("wc2.1_2.5m_bio_1.tif", "wc2.1_2.5m_bio_4.tif",
                 "wc2.1_2.5m_bio_12.tif", "wc2.1_2.5m_bio_15.tif"),
    year = rep(1L, 4)
  )
}

```

```

}

if (!suppliedElsewhere(sim$projClimateURLs)) {
  sim$projClimateURLs <- data.table(
    vars = rep(c("BIO1", "BIO4", "BIO12", "BIO15"), times = 4),
    URL = rep(c("https://geodata.ucdavis.edu/cmip6/2.5m/CanESM5/ssp585/wc2.1_2",
               "https://geodata.ucdavis.edu/cmip6/2.5m/CanESM5/ssp585/wc2.1_2",
               "https://geodata.ucdavis.edu/cmip6/2.5m/CanESM5/ssp585/wc2.1_2",
               "https://geodata.ucdavis.edu/cmip6/2.5m/CanESM5/ssp585/wc2.1_2",
               each = 4),
    targetFile = rep(c("wc2.1_2.5m_bioc_CanESM5_ssp585_2021-2040.tif",
                      "wc2.1_2.5m_bioc_CanESM5_ssp585_2041-2060.tif",
                      "wc2.1_2.5m_bioc_CanESM5_ssp585_2061-2080.tif",
                      "wc2.1_2.5m_bioc_CanESM5_ssp585_2081-2100.tif"),
                     each = 4),
    year = rep(2L:5L, each = 4)
  )
}

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

```

We draw your attention to a few particular aspects of the data modules:

- How we took care to define the data classes of parameters, expected inputs and module outputs in their respective metadata sections;
- How we added additional R packages necessary to run the module to the metadata;
- How we added default values for parameters and inputs explicitly used by the modules (others like `.plotInterval` were left as NA). The exception was the `studyAreaRas` input object for which we do not provide a default. However, we added a code check in `.inputObject` that stops interrupts R if this object is not in `sim` (see [Protect yourself and others from common mistakes/problems](#))
- How we use the function `prepInputs` to do most of the heavy-lifting of downloading data and spatial pre-processing. This function is able to recognize whether the data has already been downloaded, and can

cache all spatial processing tasks (see [Caching](#)). In some cases, we wrapped `prepInputs` in a `Map` call to loop through several URLs and download and pre-process many data layers. This `Map` call can also be cached with `Cache`.

- How we use the function `Plots` to control plotting to the screen device and/or save to image files depending on the `P(sim)$plots` argument. Note that `Plots` works best with functions that output `ggplot` objects, or that are compatible with `quickPlot::Plot`.
- The fact that neither module depends on the other. This is not a required feature of data modules, but just so happens to be the case in this example. In fact, in more complex modelling frameworks, like the LandR model (Barros et al. [n.d.](#)), we often have several data modules that depend on each other (e.g., `LandR Biomass_speciesData` sources and processes tree species percent cover data that is used by `LandR Biomass_borealDataPrep` to estimate several parameters for the forest landscape simulation model `LandR Biomass_core`).
- How we export objects created within the module functions to `sim`. Without doing so, these objects are lost after the function is executed.

### 2.2.2 Prediction module

We show below the .R script for the `projectSpeciesDist` module. This module depends entirely on the other two, as we did not provide any default input objects in the `.inputObjects` function. This is, of course, not good practice, but again we warn the user early on (in the `.inputObjects` function) if the module cannot find the necessary inputs.

This module fits a machine learning SDM using the MaxEnt algorithm implemented in the `dismo` package. We recommend having a look at [this guide](#) to learn about fitting SDMs with `dismo` and more. Before fitting the SDM, the module converts any non-binary species data into presences and absences.

The main outputs are species distribution projections in the form of plots and a stacked raster layer (`sppDistProj`) and the fitted SDM object.

```
## Everything in this file and any files in the R directory are sourced during `simInit()`
## all functions and objects are put into the `simList`.
## To use objects, use `sim$xxx` (they are globally available to all modules).
## Functions can be used inside any function that was sourced in this module;
## they are namespaced to the module, just like functions in R packages.
```

```

## If exact location is required, functions will be: `sim$.mods$<moduleName>$Fun
defineModule(sim, list(
  name = "projectSpeciesDist",
  description = "",
  keywords = "",
  authors = structure(list(list(given = c("Ceres"), family = "Barros",
                                 role = c("aut", "cre"), email = "ceres.barros@ubc.ca")),
    childModules = character(0),
    version = list(projectSpeciesDist = "1.0.0"),
    timeframe = as.POSIXlt(c(NA, NA)),
    timeunit = "year",
    citation = list("citation.bib"),
    documentation = list("README.md", "projectSpeciesDist.Rmd"), ## same file
    reqdPkgs = list("SpaDES.core (>=2.0.2)",
                    "caret", "data.table", "dismo",
                    "ggplot2", "rJava", "rasterVis"),
    parameters = bindrows(
      #defineParameter("paramName", "paramClass", value, min, max, "parameter description")
      defineParameter("predVars", "character", c("BI01", "BI04", "BI012", "BI015"),
                      "Predictors used in statistical model."),
      defineParameter("presThresh", "numeric", 10, 0, NA,
                      paste("Minimum threshold for the species to be considered present in the sppAbundanceDT table. It is used to filter sppAbundanceDT to remove non binary species data (e.g. presence/absence). By default 10% cover."),
      defineParameter("statModel", "character", "MaxEnt", NA, NA,
                      paste("What statistical algorithm to use. Currently only 'MaxEnt' and 'GLM' are supported. 'MaxEnt' will fit a MaxEnt model using dismo package. 'GLM' will fit a generalised linear model with a logit link function (glm(..., family = 'binomial')). In both cases all predictors are included in the model. Note that for GLM only additive effects are considered." ))
      defineParameter(".plots", "character", "screen", NA, NA,
                      "Used by Plots function, which can be optionally used here")
      defineParameter(".plotInitialTime", "numeric", start(sim), NA, NA,
                      "Describes the simulation time at which the first plot event occurs")
      ## .seed is optional: `list('init' = 123)` will `set.seed(123)` for the `initial` seed
      defineParameter(".seed", "list", list(), NA, NA,
                      "Named list of seeds to use for each event (names)."),
      defineParameter(".useCache", "logical", FALSE, NA, NA,
                      "Should caching of events or module be used?")
    )
  )
)

```

```

),
inputObjects = bindrows(
  #expectsInput("objectName", "objectClass", "input object description", sourceURL, ...),
  expectsInput("climateDT", "data.table",
    desc = paste("A data.table with as many columns as the climate covariates",
                "used in the species distribution model and 'year' column denotes",
                "the simulation year to which the data corresponds.")),
  expectsInput("sppAbundanceDT", "data.table",
    desc = paste("A species abundance data. Converted to presence/absence data",
                "By default a table with % species cover.")),
  expectsInput("studyAreaRas", objectClass = "SpatRaster",
    desc = "A binary raster of the study area")
),
outputObjects = bindrows(
  #createsOutput("objectName", "objectClass", "output object description", ...),
  createsOutput(objectName = "sppDistProj", objectClass = "SpatRaster",
    desc = paste("Species distribution projections - raw predictions.",
                "Each layer corresponds to a prediction year")),
  createsOutput(objectName = "evalOut", objectClass = "ModelEvaluation",
    desc = paste("`sdmOut` model evaluation statistics. Model evaluated on",
                "the data. See `?dismo::evaluation`.")),
  createsOutput(objectName = "sdmData", objectClass = "data.table",
    desc = "Input data used to fit `sdmOut`."),
  createsOutput(objectName = "sdmOut", objectClass = c("MaxEnt", "glm"),
    desc = paste("Fitted species distribution model. Model fitted on 80%",
                "of `sdmData`, with remaining 20% used for evaluation.")),
  createsOutput(objectName = "thresh", objectClass = "numeric",
    desc = paste("Threshold of presence that maximises the sum of the sensitivity",
                "(true positive rate) and specificity (true negative rate)",
                "See `dismo::threshold(..., stat = 'spec_sens')`.")))
)

## event types
# - type `init` is required for initialization

doEvent.projectSpeciesDist = function(sim, eventTime, eventType) {
  switch(
    eventType,

```

```

init = {
  ### check for more detailed object dependencies:
  ### (use `checkObject` or similar)

  # do stuff for this event
  sim <- SDMInit(sim)

  # schedule future event(s)
  sim <- scheduleEvent(sim, start(sim), "projectSpeciesDist", "fitSDM")
  sim <- scheduleEvent(sim, start(sim), "projectSpeciesDist", "evalSDM",
                       eventPriority = .normal() + 1)
  sim <- scheduleEvent(sim, start(sim), "projectSpeciesDist", "projSDM",
                       eventPriority = .normal() + 2)
  sim <- scheduleEvent(sim, P(sim)$plotInitialTime, "projectSpeciesDist", "plotProjSDM",
                       eventPriority = .normal() + 3)

  },
  fitSDM = {
    # ! ----- EDIT BELOW ----- !
    sim <- fitSDMEvent(sim)
    # ! ----- STOP EDITING ----- !
  },
  evalSDM = {
    # ! ----- EDIT BELOW ----- !
    sim <- evalSDMEvent(sim)
    # ! ----- STOP EDITING ----- !
  },
  projSDM = {
    # ! ----- EDIT BELOW ----- !
    sim <- projSDMEvent(sim)

    sim <- scheduleEvent(sim, time(sim) + 1L, "projectSpeciesDist", "projSDM")
    # ! ----- STOP EDITING ----- !
  },
  plotProjSDM = {
    # ! ----- EDIT BELOW ----- !
    plotProjEvent(sim)

    sim <- scheduleEvent(sim, time(sim) + 1L, "projectSpeciesDist", "plotProjSDM")
  }
}

```

```

    eventPriority = .normal() + 1)

    # ! ----- STOP EDITING ----- ! #
  },
  warning(paste("Undefined event type: \\'", current(sim)[1, "eventType", with = FALSE],
                "\\' in module \\'", current(sim)[1, "moduleName", with = FALSE], "\\'", se
  )
  return(invisible(sim))
}

## event functions
# - keep event functions short and clean, modularize by calling subroutines from section

#### template initialization
SDMInit <- function(sim) {
  # # ! ----- EDIT BELOW ----- !
  ## at this point we can only have the following columns
  if (!identical(sort(names(sim$sppAbundanceDT)), sort(c("cell", "x", "y", "sppAbund", "ye
    stop(paste("sim$sppAbundanceDT can only have the following columns at the start of ye
      paste(c("cell", "x", "y", "sppAbund", "year"), collapse = ", ")))
  }

  if (length(setdiff(sim$climateDT$cell, sim$sppAbundanceDT$cell)) > 0 || 
      length(setdiff(sim$sppAbundanceDT$cell, sim$climateDT$cell)) > 0) {
    stop("'cell' columns in `climateDT` and `sppAbundanceDT` have different values")
  }

  if (!P(sim)$statModel %in% c("MaxEnt", "GLM")) {
    stop("'statModel' parameter must be 'MaxEnt' or 'GLM'")
  }

  ## a few data cleaning steps to make sure we have presences and absences:
  sppAbundanceDT <- copy(sim$sppAbundanceDT)
  if (min(range(sppAbundanceDT$sppAbund)) < 0) {
    sppAbundanceDT[sppAbund < 0, sppAbund := 0]
  }

  if (!all(unique(sppAbundanceDT$sppAbund) %in% c(0, 1))) {
    message("Species data is not binary.")
  }
}

```

```

message("Converting values >= P(sim)$presThresh to presences, and < P(sim)$pr
sppAbundanceDT[sppAbund >= P(sim)$presThresh, presAbs := 1]
sppAbundanceDT[sppAbund < P(sim)$presThresh, presAbs := 0]
}

## join the two datasets - note that there are no input species abundances bey
sim$sdmData <- merge(sim$climateDT, sppAbundanceDT[, .(cell, sppAbund, presAbs
by = c("cell", "year"), all = TRUE)

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

fitSDMEvent <- function(sim) {
  # ! ----- EDIT BELOW ----- !
  ## break data into training and testing subsets
  dataForFitting <- sim$sdmData[year == time(sim)]

  if (nrow(dataForFitting) == 0) {
    stop(paste("No data for year", time(sim), "provided to fit the model"))
  }

  group <- createDataPartition(dataForFitting$presAbs, p = 0.8, list = FALSE)
  ## save the split datasets as internal objects to this module
  mod$trainData <- dataForFitting[group]
  mod$testData <- dataForFitting[-group]

  if (!any(mod$trainData$presAbs == 0)) {
    stop("Training dataset contains no absences.")
  }

  predVars <- P(sim)$predVars
  if (P(sim)$statModel == "MaxEnt") {
    sim$sdmOut <- maxent(x = as.data.frame(mod$trainData[, ..predVars]),
                           p = mod$trainData$presAbs)
  } else {
    ## make an additive model with all predictors - avoid using as.formula, which
    form <- enquote(paste("presAbs ~", paste(predVars, collapse = "+")))
    sim$sdmOut <- glm(formula = eval(expr = parse(text = form)),
}

```

```

                family = "binomial", data = mod$trainData)
}
# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

evalSDMEvent <- function(sim) {
# ! ----- EDIT BELOW ----- !
## validate model
predVars <- P(sim)$predVars
sim$evalOut <- evaluate(p = mod$testData[presAbs == 1, ..predVars],
                        a = mod$testData[presAbs == 0, ..predVars],
                        model = sim$sdmOut)
## save the threshold of presence/absence in an internal object to this module
sim$thresh <- threshold(sim$evalOut, 'spec_sens')

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

projSDMEvent <- function(sim) {
# ! ----- EDIT BELOW ----- !
## predict across the full data and make a map
dataForPredicting <- sim$sdmData[year == time(sim)]

if (nrow(dataForPredicting) == 0) {
  stop(paste("No data for year", time(sim), "provided to calculate predictions"))
}

predVars <- P(sim)$predVars
preds <- predict(sim$sdmOut, as.data.frame(dataForPredicting[, ..predVars]),
                 progress = '')
sppDistProj <- replace(sim$studyAreaRas, which(!is.na(sim$studyAreaRas[])), preds)
names(sppDistProj) <- paste0("year", time(sim))

if (is.null(sim$sppDistProj)) {
  sim$sppDistProj <- sppDistProj
} else {
  sim$sppDistProj <- rast(list(sim$sppDistProj, sppDistProj))
}
}
```

```

}

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

plotProjEvent <- function(sim) {
# ! ----- EDIT BELOW ----- !
checkPath(file.path(outputPath(sim), "figures"), create = TRUE)

if (any(!is.na(P(sim)$plots))) {

## response plot
## we can't use Plots to plot and save SDM predictions with dismo.
## these are only saved to disk
fileSuffix <- paste0(P(sim)$statModel, ".png")

notScreen <- setdiff(P(sim)$plots, "screen")
if (any(notScreen != "png")) {
  warning(paste(currentModule(sim), "only saves to PNG at the moment."))
}
png(file.path(outputPath(sim), "figures", paste0("SDMresponsePlot_", fileSuf
response(sim$sdmOut)
dev.off()

## species projections
fileSuffix <- paste0(P(sim)$statModel, "_Year", time(sim))
clearPlot()
rawValsPlot <- sim$sppDistProj[[paste0("year", time(sim))]]
Plots(rawValsPlot, fn = plotSpatRaster, types = P(sim)$plots,
      usePlot = TRUE, filename = file.path(outputPath(sim), "figures", paste0("Projected raw values -", "year", time(sim))),
      plotTitle = paste("Projected raw values -", "year", time(sim)),
      xlab = "Longitude", ylab = "Latitude")

PAsPlot <- terra::as.int(sim$sppDistProj[[paste0("year", time(sim))]] > sim$sdmOut)
Plots(PAsPlot, fn = plotSpatRaster, types = P(sim)$plots,
      usePlot = TRUE, filename = file.path(outputPath(sim), "figures", paste0("Projected presence/absence -", "year", time(sim))),
      plotTitle = paste("Projected presence/absence -", "year", time(sim)),
      xlab = "Longitude", ylab = "Latitude")
}

```

```

}

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

.inputObjects <- function(sim) {
#cacheTags <- c(currentModule(sim), "function:.inputObjects") ## uncomment this if Cache
dPath <- asPathgetOption("reproducible.destinationPath", dataPath(sim)), 1)
message(currentModule(sim), ": using dataPath '", dPath, "'.")

# ! ----- EDIT BELOW ----- !
## check that necessary objects are in the simList or WILL BE supplied by another module
if (!suppliedElsewhere("climateDT") | !suppliedElsewhere("sppAbundanceDT")) {
  stop("Please provide `climateDT` and `sppAbundanceDT`")
}

# ! ----- STOP EDITING ----- !
return(invisible(sim))
}

```

We draw your attention to:

- As we said earlier, we could have added yearly projected values to the `sppAbundanceDT` table. In this case we probably would have changed this object's name, since MaxEnt is not modelling species abundance, but probability of occurrence. We suggest this as an exercise to do on your own
- How links with the data modules are established by declaring data modules' output objects as expected inputs for this module.
- How the `fitSDM` event does not schedule itself (the SDM only needs to be fitted once).
- How, unlike `fitSDM`, the `projSDM` event schedules itself so that model projections are executed for each year of the simulation, provided that there is corresponding environmental data – notice how the functions `fitSDMEvent` and `projSDMEvent` both check that there is data for the current year of the simulation (`time(sim)`).
- How the fitted model object (`sdmOut`) and it's evaluation (`evalOut`)

are both module outputs. This way these objects can not only be used by other events, but also inspected by the user after the simulation is finished (see [Transparent models](#)).

You will notice that this module performs model fitting (i.e., calibration), predictions and model validation. These three components could be broken into three separate modules. As an exercise, we recommend trying to do so on your own.

### 2.2.3 Additional module scripts and functions

All functions needed by a module can be coded within the module script. Often they will be located after all the event functions and before the `.inputObjects` function; however, their exact location is up to the developer.

For long, complex modules involving many (potentially long) functions, including all functions within a module’s script quickly makes the script long and hard to maintain. Instead, we recommend that additional (i.e., non-event functions) are put into separate scripts and in the module’s `R/` folder. `SpaDES` will source any scripts contained in this folder and associate its objects/functions with the respective module.

*Note that, these functions/objects will only be available to the module where the `R/` is located.*

Hence, if one or more functions are meant to be shared across modules they will need to either:

- be defined in `.GlobalEnv` (the global environment – see `?GlobalEnv`) before running `simInit`;
- be put it in an `R` package that is included in the modules’ metadata and installed as other packages – hence the package is in an online repository like GitHub or CRAN;
- be put it in a *local* `R` package that is included in the modules’ metadata and installed (e.g., `devtools::install_local`) or loaded (e.g. `devtools::load_all("<pathToPackage>")`) “separately” by the user before, as `Require` is not able to install local packages.

In this example, we put the plotting functions called by `Plots` into separate scripts for each module where they are used. Modules `speciesAbundanceData` and `projectSpeciesDist` both have the function

`plotSpatRaster` in a `plotSpatRaster.R` script saved in their respective R/folders:

```
#' Function used to plot SpatRaster
#'
#' To be used with Plots
#'
#' @param ras a SpatRaster layer
#' @param title character. Plot title
#' @param xlab character. X-axis title
#' @param ylab character. Y-axis title
#'
#' @importFrom rasterVis gplot
#' @importFrom ggplot2 geom_tile scale_fill_brewer coord_equal theme_bw

plotSpatRaster <- function(ras, plotTitle = "", xlab = "x", ylab = "y") {
  gplot(ras) + geom_tile(aes(fill = value)) + scale_fill_distiller(palette = "Blues",
    direction = 1, na.value = "grey90", limits = c(0, 1)) +
    theme_classic() + coord_equal() + labs(title = plotTitle,
    x = xlab, y = ylab)
}
```

The `climateData` module uses the function `plotSpatRasterStk`, which is in the `plotSpatRasterStk.R` in its R/ folder:

```
#' Function used to plot SpatRaster Stacks
#'
#' To be used with Plots
#'
#' @param stk a SpatRaster stack.
#' @param title character. Plot title
#' @param xlab character. X-axis title
#' @param ylab character. Y-axis title
#'
#' @importFrom rasterVis gplot
#' @importFrom ggplot geom_tile facet_wrap scale_fill_brewer coord_equal

plotSpatRasterStk <- function(stk, plotTitle = "", xlab = "x",
  ylab = "y") {
  gplot(stk) + geom_tile(aes(fill = value)) + scale_fill_distiller(palette = "Blues",
    direction = 1, na.value = "grey90") + theme_classic() +
```

```

    coord_equal() + facet_wrap(~variable) + labs(title = plotTitle,
      x = xlab, y = ylab)
}

```

Go ahead and create these scripts with the functions above and put them in the right R/ folders.

---

## 2.3 Running the model

### 2.3.1 Ensuring all packages are installed

After the modules are created, we go back to the `Part2_SDMs.R` script to set up and run the simulation. We first ensure that all module dependencies (and their dependencies and so on) are installed in `.libPaths()`.

Only then do we load packages necessary to run the simulation.

Note that the `dismo` package (a dependency of the `projectSpeciesDist` module) needs `rJava` to run `maxent`. In turn, `rJava` needs a working Java installation (Java can be downloaded [here](#)). Below you will see that we attempt to warn the user about problems loading `rJava`, which are likely related to Java not being found on the system.

```

outs <- SpaDES.project::packagesInModules(modulePath = simPaths$modulePath) ## 
Require::Require(c(unname(unlist(outs)),
  "DiagrammeR"),
  require = FALSE,    ## don't load packages
  upgrade = FALSE,   ## don't upgrade dependencies
  standAlone = TRUE, ## install all dependencies in proj-lib (ign
  purge = TRUE)

## now load packages - SpaDES.core may have been loaded already, which is fine
Require::Require(c("reproducible", "SpaDES.core", "SpaDES.experiment"),
  install = FALSE)

## dismo needs a few tweaks to run MaxEnt
out <- reproducible::preProcess(targetFile = "maxent.jar",
  url = "https://github.com/mrmaxent/Maxent/blob/m
  destinationPath = simPaths$inputPath,

```

```

            fun = NA)
file.copy(out$targetFilePath, file.path(system.file("java", package="dismo"), "maxent.jar"
                                         overwrite = TRUE)

out <- require(rJava)
if (!out) {
  stop(paste("Your Java installation may have problems, please check.\n",
            "See https://www.java.com/en/download/manual.jsp for Java installation"))
}

```

/!\ ATTENTION /!\  
*Windows can present problems when many packages are installed and further package installations fail. If you see errors like this after restarting R:*

Installing: glue Detaching is fraught with many potential problems; you may have to restart your session if things aren't working some packages don't seem to unload their dlls correctly.  
These will not be unloaded: ellipsis, vctrs Unloading package bit64 --  
Installing glue -- (1 of 1. Estimated time left: ...; est. finish: ...calculating)  
Installing package into '\~/R/win-library/4.0' (as 'lib' is unspecified) trying URL  
'<https://cran.rstudio.com/bin/windows/contrib/4.0/glue\_1.6.2.zip>' Content type  
'application/zip' length 171858 bytes (167 KB) downloaded 167 KB  
  
package 'glue' successfully unpacked and MD5 sums checked  
Error in unpackPkgZip(foundPkgs[okp, 2L], foundPkgs[okp, 1L], lib, libs\_only, :  
ERROR: failed to lock directory '\~/R/win-library\4.0' for modifying  
Try removing '\~/R/win-library\4.0/00LOCK'

*If you encounter this error, delete the problematic file/folder and try again.*

*Sometimes `Require` may still unable to automatically install a package and a manual installation from a clean session is the only solution. In the error above, `Require` only detected that `glue` was missing during the `simInit` call, which meant that other packages had been loaded already causing failure when `Require` attempted the installation. This problem persisted even after we avoided loading `terra` before running `simInit` (we make the study area objects prefixing functions with `terra:::`), so `glue` had to be manually installed.*

### 2.3.2 Simulation set-up

The simulation folder directories were already set up before creating the modules (see above), but it is still necessary to create a few lists that will be passed to the `simInit` function, which initializes the simulation. These lists define the modules used in the simulation (`simModules`), the start and end of the simulation (`simTimes`), the parameters passed to each module (`simParams`) and external input objects (`simObjects`) like the study area (`studyAreaRas`).

The `studyAreaRas` is created from a random polygon drawn in SW Alberta, Canada, using `SpaDES.tools::randomStudyArea`. (Figure 2.1).

We also define a few useful global options:

- `reproducible.cachePath` and `reproducible.destinationPath` define the cache directory and the directory where downloaded and processed data will be stored;
- `reproducible.useCache` and `reproducible.useTerra`, which will activate caching and the use of the `terra` package across all Cache and `prepInputs` function calls.

```
## a few important options:
options(reproducible.useCache = TRUE,
        reproducible.cachePath = simPaths$cachePath,
        reproducible.destinationPath = simPaths$inputPath, ## all downloaded and
        reproducible.useTerra = TRUE, ## we want to use the terra R package
        spades.moduleCodeChecks = FALSE,
        spades.useRequire = FALSE)

## list the modules to use
simModules <- list("speciesAbundanceData", "climateData", "projectSpeciesDist")

## Set simulation and module parameters
simTimes <- list(start = 1, end = 5, timeunit = "year")

## we create two lists of parameters, one using the default MaxEnt
## the other a GLM
simParamsMaxEnt <- list(
  "speciesAbundanceData" = list(
    ".plots" = c("png"),
```

```

  ".useCache" = FALSE
),
"climateData" = list(
  ".plots" = c("png"),
  ".useCache" = FALSE
),
"projectSpeciesDist" = list(
  "statModel" = "MaxEnt",
  ".plots" = c("png"),
  ".useCache" = FALSE
)
)
simParamsGLM <- simParamsMaxEnt
simParamsGLM$projectSpeciesDist$statModel <- "GLM"

## make a random study area.
## Here use seed to make sure the same study area is always generated
studyArea <- SpaDES.tools::randomStudyArea(size = 1e10, seed = 123)
studyAreaRas <- terra::rasterize(studyArea,
                                    terra::rast(extent = terra::ext(studyArea),
                                                crs = terra::crs(studyArea, proj = TRUE),
                                                resolution = 1000))
simObjects <- list(
  "studyAreaRas" = studyAreaRas
)

## Simulation setup - create two simulations, one for MaxEnt another for GLM
## SpaDES.experiment::experiment2, will take care of subdirectories to store outputs
mySimMaxEnt <- simInit(times = simTimes, params = simParamsMaxEnt,
                        modules = simModules, objects = simObjects,
                        paths = simPaths)
mySimGLM <- simInit(times = simTimes, params = simParamsGLM,
                      modules = simModules, objects = simObjects,
                      paths = simPaths)

```

Before running the simulation we look at the module linkage diagrams produced by `moduleDiagram` (Figure 2.2)) and `objectDiagram` (Figure 2.3) to assess whether modules are linked as expected.

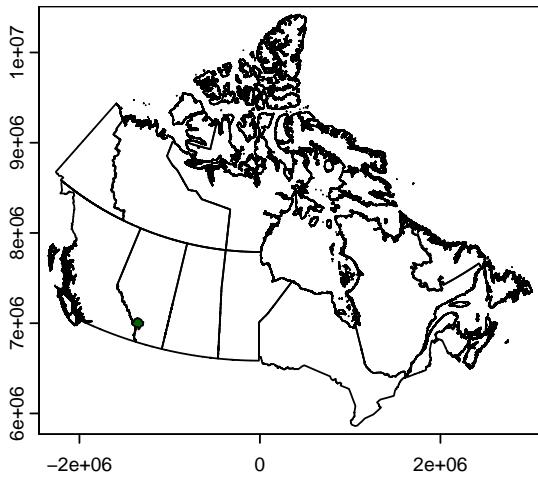


Figure 2.1: Study area within Canada.

```
moduleDiagram(mySimMaxEnt)
objectDiagram(mySimMaxEnt)
```

### 2.3.3 Simulation runs

To run the simulation, we can call `spades` on the output `simLists` (called `mySimMaxEnt` and `mySimMaxGLM` here) generated by `simInit`, or use `experiment2` from the `SpaDES.experiment` package. `experiment2` will run as many simulations as `simLists` and organise outputs into sub-folders within the `simPaths$outputs` directory. It can also repeat simulations (`rep` argument) and parallelise across replicates using `future`. See `?experiment2` for examples.

We advise using `spades` when running the model for the first time. Passing the argument `debug = TRUE` will print the progress of the simulation in detail. This helps diagnosing problems when the simulation fails, but also seeing which events are being executed and when particular cache calls are activated.

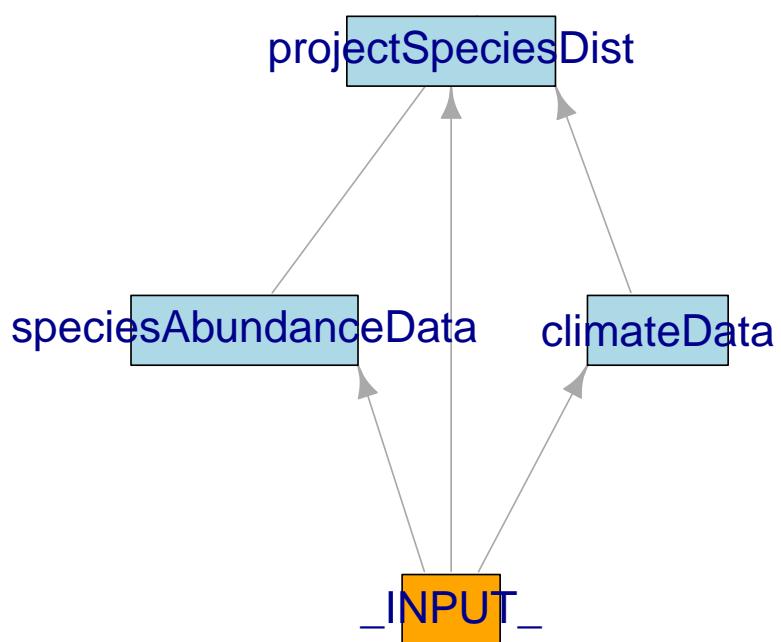


Figure 2.2: Module network diagram.

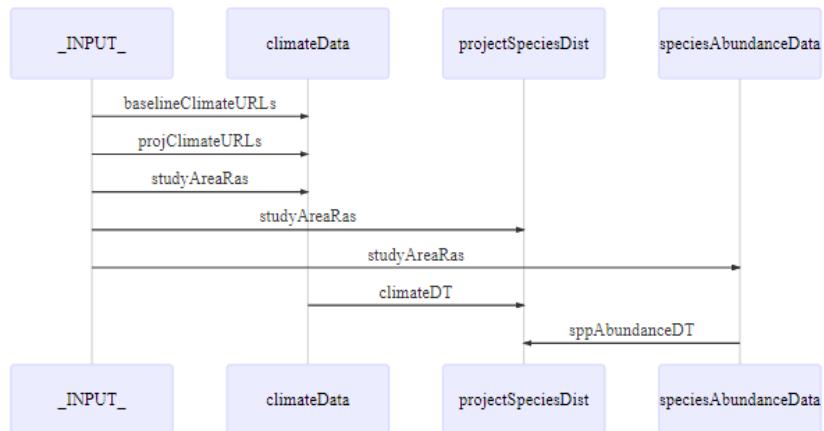


Figure 2.3: Module diagram showing module inter-dependencies with object names.

```

## run simulation
clearPlot(force = TRUE) ## this forces wiping the graphics device and opening a new window

## This runs one simulation and stores outputs in the main
## 'outputs' folder - not what we want, but good for
## testing mySimOut <- spades(mySimMaxEnt, debug = TRUE)

## Better to use when spades runs error-free on the
## simLists
myExperiment <- experiment2(MaxEnt = mySimMaxEnt, GLM = mySimGLM,
    debug = TRUE, replicates = 1, clearSimEnv = FALSE) ## prevent removing objects from
## save outputs
qs::qsave(myExperiment, file.path(simPaths$outputPath, paste0("myExperiment",
    ".qs")))

```

Try to execute the `spades` call twice to see how much faster it runs after many of the operations have been cached (see also [Caching](#)).

By default the data modules (`speciesAbundanceData` and `climateData`) save figures of the input species and climate layers (Figure 2.4 and Figure 2.5 , respectively).

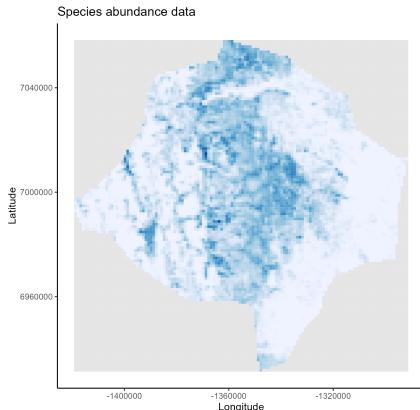


Figure 2.4: Prediction plots. Input *Picea glauca* percent cover across the landscape. Note that values are converted to presence/absence.

The prediction module also outputs the projections for each climate period

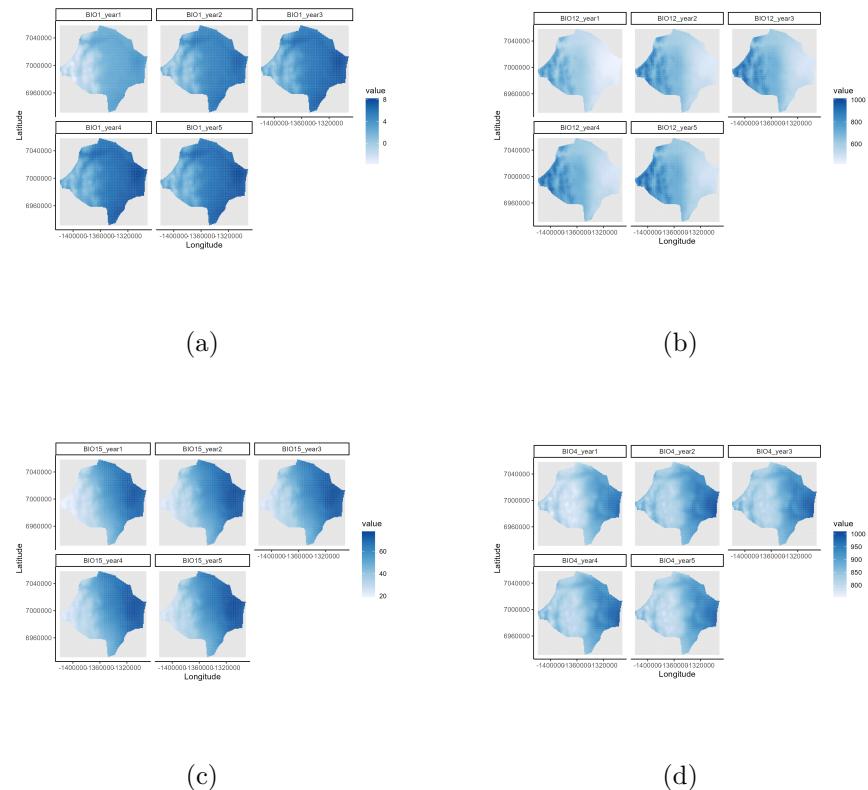


Figure 2.5: Prediction plots. Bioclimatic variables under baseline (year 1) and future conditions.

automatically ( Figure 2.6 and Figure 2.7 ).

The projected layers can also be accessed and plotted via the `simList` object, as can the model validation results.

From the results we can see that the MaxEnt and GLM predictions do not seem to agree, indicating a potential problem. We may be missing important covariates, interactions, or simply more appropriate algorithms.

Peruse each model's estimated coefficients and residuals, and validation results will be a good first step to diagnosing the problem.

```
myExperiment$MaxEnt_rep1$sdmOut ## this links to an html page

sets <- par(mfrow = c(2, 2))
plot(myExperiment$GLM_rep1$sdmOut)
par(sets)

## check validation results for the two models
myExperiment$MaxEnt_rep1$evalOut
myExperiment$GLM_rep1$evalOut
```

### 2.3.3.1 Adding a new climate scenario

Because data were linked to the modules (and the forecasting) via the modules' metadata and inputs, adding a new climate scenario and re-running forecasts is easy.

To do so, we need only to change the URLs for the climate layers, by passing a custom `projClimateURLs` `data.table` to the `climateData` module. `SpaDES` will take care of downloading and processing the new layers, as well as forecasting. Model fitting will also be repeated, even if the baseline data did not change, because the `kfold` function we use to partition the data into the training and testing subsets randomly assigns cases to each group. If this was not desired, we could set a random seed before running the fitting event (`fitSDM`) by passing the `.seed` parameter to the `projectSpeciesDist` module (e.g., `.seed = list("fitSDM" = 123)`).

```
## Run with another climate scenario - the most contrasting
## scenario to SSP 585 get the original table from one of
## the simulations and replace the climate scenario
projClimateURLs <- copy(mySimMaxEnt$projClimateURLs)
projClimateURLs[, `:=` (URL = gsub("ssp585", "ssp126", URL), targetFile = gsub("ssp585",
```

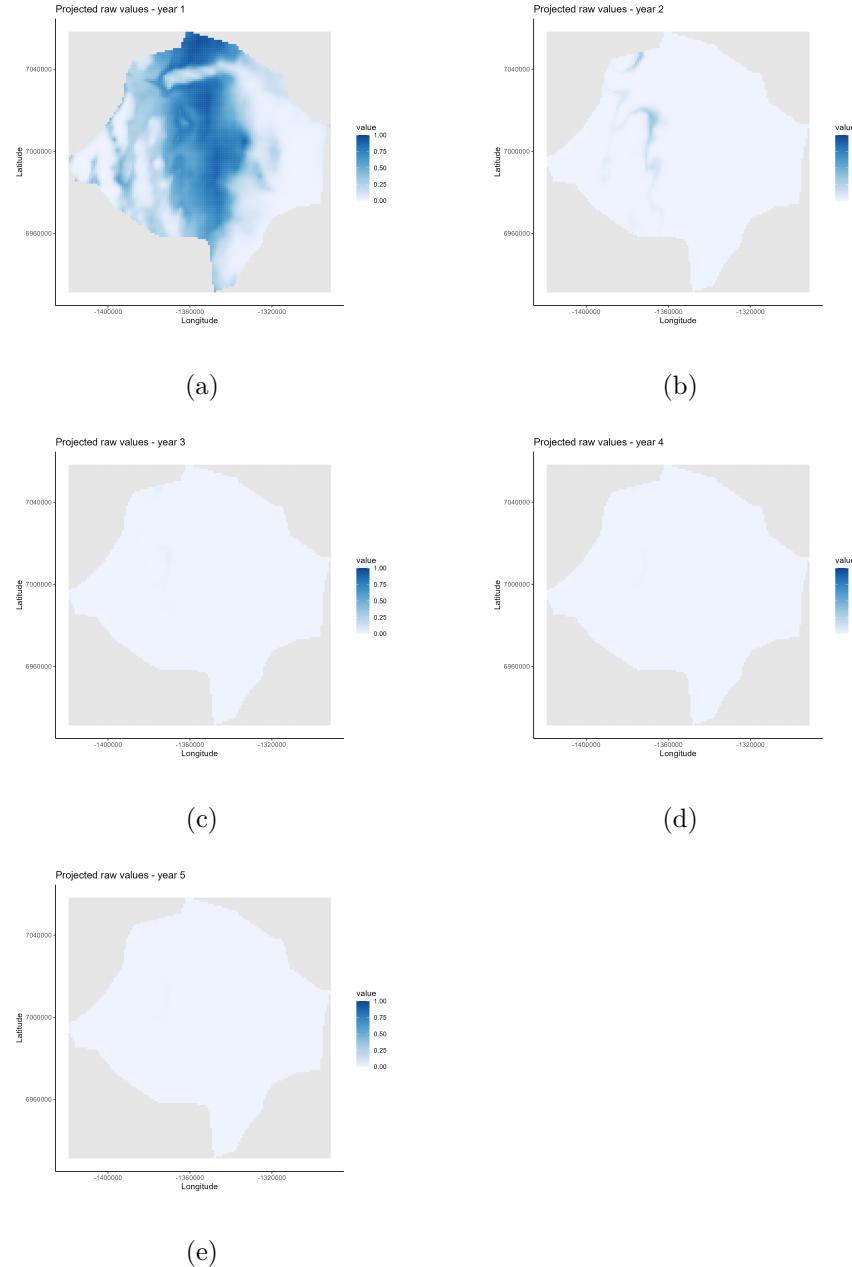


Figure 2.6: Prediction plots: Raw predicted values of species probability of occurrence under (left to right) baseline climate conditions (first year of simulation), 2021-2040, 2041-2060, 2061-2080 and 2081-2100 climate conditions (second to fifth years of simulation) - using MaxEnt.

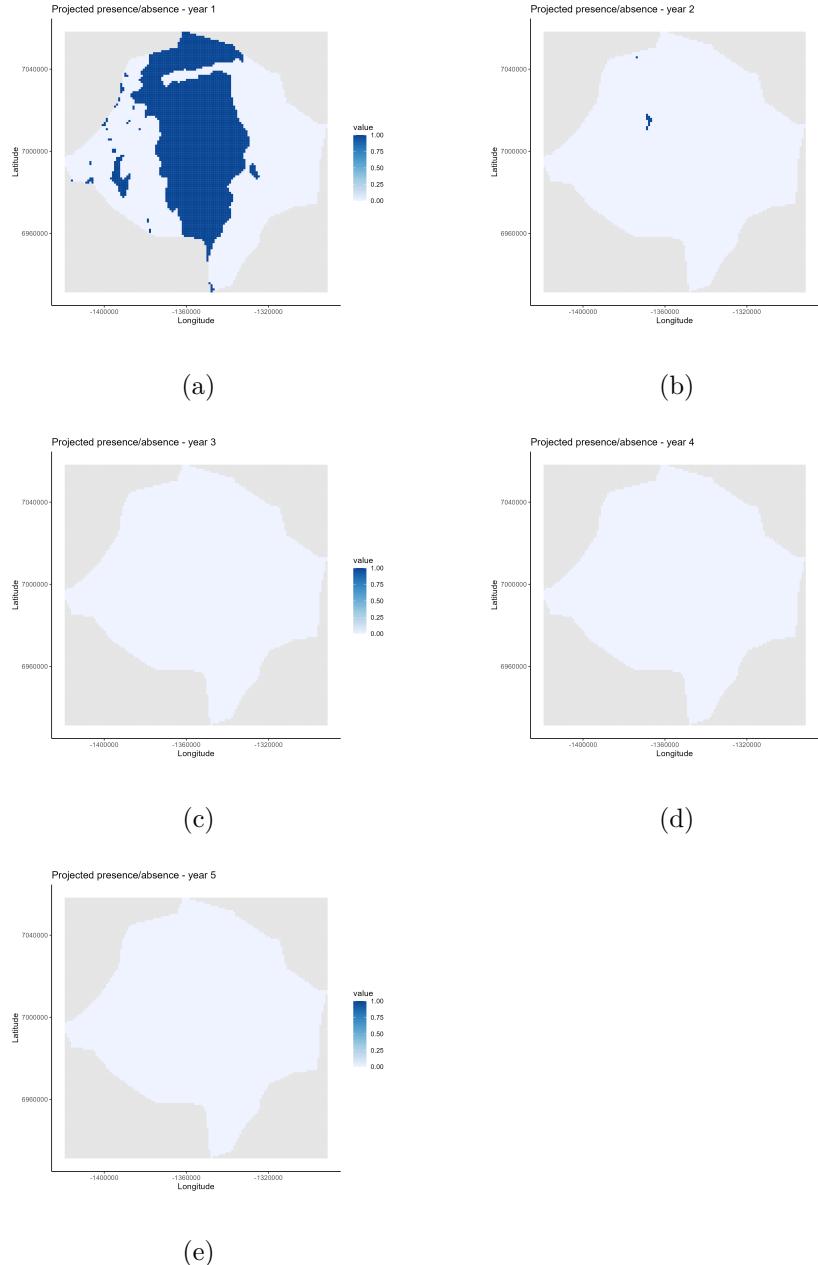


Figure 2.7: Prediction plots: Predictions of *Picea glauca* presence/absence under (left to right) baseline climate conditions (first year of simulation), 2021-2040, 2041-2060, 2061-2080 and 2081-2100 climate conditions (second to fifth years of simulation) - using MaxEnt.

```

    "ssp126", targetFile)])]

## this time we pass the new table or URLs to the modules,
## so that climate layers are changed
simObjects2 <- list(studyAreaRas = studyAreaRas, projClimateURLs = projClimateURLs)

mySimMaxEnt2 <- simInit(times = simTimes, params = simParamsMaxEnt,
    modules = simModules, objects = simObjects2, paths = simPaths)
mySimGLM2 <- simInit(times = simTimes, params = simParamsGLM,
    modules = simModules, objects = simObjects2, paths = simPaths)

myExperiment2 <- experiment2(MaxEnt = mySimMaxEnt2, GLM = mySimGLM2,
    debug = TRUE, replicates = 1, clearSimEnv = FALSE)
## save outputs
qs::qsave(myExperiment2, file.path(simPaths$outputPath, paste0("myExperiment2",
    ".qs")))

```

### 2.3.3.2 Proposed exercises

1. try changing the climate layers (e.g., use different climate scenarios or General Circulation models) and rerunning predictions;
2. try adding other statistical algorithms;
3. try breaking up the prediction module into three modules: a calibration module, a prediction module and a validation module.

Have fun!

### 2.3.3.3 Making use of `simList` for reporting

Another advantage of having all simulation parameters, inputs and outputs centralised in one object, is that we can easily inspect and manipulated them afterwards, without the need to load separate objects back into R.

Here we show how we capitalize on this SpADES feature to create figures of the outputs (Figure 2.8).

```

## MaxEnt predictions across time and for each climate
## scenario ----- combine plots from two distinct
## simulations in a single figure (the same can be done to
## compare MaxEnt and GLM, or plot all projections)

```

```

## fetch the internal plotting function instead of
## repeating code here
plotFun <- myExperiment$GLM_rep1@.envir$.mods$climateData$plotSpatRasterStk

## raw predictions exported by the module
sppDistProjMaxEnt <- myExperiment$MaxEnt_rep1$sppDistProj
sppDistProjMaxEnt2 <- myExperiment2$MaxEnt_rep1$sppDistProj

## we convert the raw predictions into presence absence
## using exported threshold
sppDistProjMaxEnt_PA <- as.int(myExperiment$MaxEnt_rep1$sppDistProj >
  myExperiment$MaxEnt_rep1$thresh)
sppDistProjMaxEnt2_PA <- as.int(myExperiment2$MaxEnt_rep1$sppDistProj >
  myExperiment2$MaxEnt_rep1$thresh)

## rename layers from plotting
names(sppDistProjMaxEnt) <- names(sppDistProjMaxEnt2) <- c("2001",
  "2021-2040", "2041-2060", "2061-2080", "2081-2100")
names(sppDistProjMaxEnt_PA) <- names(sppDistProjMaxEnt2_PA) <- c("2001",
  "2021-2040", "2041-2060", "2061-2080", "2081-2100")

## for a simpler plot choose only years 2001, 2041-2060 and
## 2081-2100
yrs <- c("2001", "2041-2060", "2081-2100")
plotMaxEnt <- plotFun(sppDistProjMaxEnt[[yrs]], xlab = "Longitude",
  y = "Latitude", plotTitle = "MaxEnt raw predictions - SSP 585") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0, 1),
  begin = 0.25)
plotMaxEnt2 <- plotFun(sppDistProjMaxEnt2[[yrs]], xlab = "Longitude",
  y = "Latitude", plotTitle = "MaxEnt raw predictions - SSP 126") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0, 1),
  begin = 0.25)
plotMaxEnt_PA <- plotFun(sppDistProjMaxEnt_PA[[yrs]], xlab = "Longitude",
  y = "Latitude", plotTitle = "MaxEnt presence/absence - SSP 585") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0, 1),
  begin = 0.25)
plotMaxEnt2_PA <- plotFun(sppDistProjMaxEnt2_PA[[yrs]], xlab = "Longitude",
  y = "Latitude", plotTitle = "MaxEnt presence/absence - SSP 126") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0, 1),
  begin = 0.25)

```

```

begin = 0.25)

## organise the plots with mildest scenario first It is
## clear that MaxEnt and GLM do not agree in their
## prediction
plotAll <- ggpubr::ggarrange(plotMaxEnt2 + labs(title = expression(bold("Scenario
y = expression(atop(bold("Raw predictions"), "Latitude"))) +
theme(legend.title = element_blank(), legend.key.height = unit(3,
"lines"), plot.title = element_text(hjust = 0.5), plot.margin = margin(0
0, 0, 0)), plotMaxEnt + labs(title = expression(bold("Scenario - SSP 585
y = expression(atop(bold(""), ""))) + theme(plot.title = element_text(hjust =
plot.margin = margin(0, 0, 0, 0)), plotMaxEnt_PA + labs(title = expression(
y = expression(atop(bold("Presence/absence"), "Latitude"))) +
theme(plot.margin = margin(0, 0, 0, 0)), plotMaxEnt_PA +
labs(title = expression(bold(""))), y = expression(atop(bold(""), ""))
+ theme(plot.margin = margin(0, 0, 0, 0)), legend = "right",
common.legend = TRUE, labels = c("a)", "b)", "c)", "d)"))

## save figure:
figDir <- checkPath(file.path(simPaths$outputPath, "generalFigures"),
create = TRUE)
ggsave(file.path(figDir, "MaxEntPredictions.png"), width = 13.5,
height = 5.5, units = "in", dpi = 300)

```

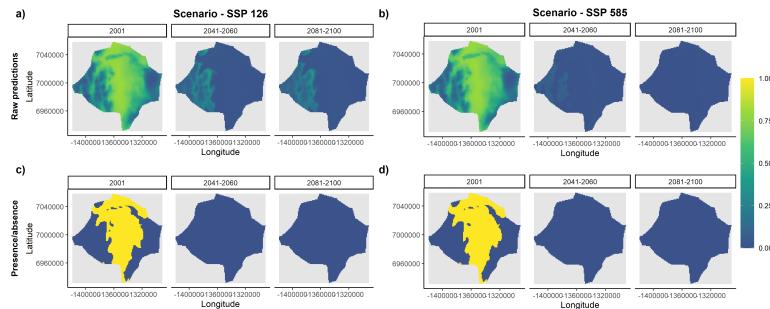


Figure 2.8: Adding a new scenario: Predictions of *Picea glauca* probabilities of presences and presence/absence under (left to right) baseline climate conditions, 2041-2060, and 2081-2100 climate projections under two emission scenarios (SSP 136 and SSP 585, the default) – showing MaxEnt forecasts only.

## 2.4 Caching

In this example, we relied on caching to avoid having to repeat computationally intensive operations. Running the `simInit` and `spades` calls a second time (even after restarting R session) was faster and SpaDES informed us of instances where cached objects were being retrieved:

```
(...)
Mar05 19:56:53 clmtDt 1 climateData init 1\
Mar05 19:56:53 clmtDt ... (Object to retrieve (a7816e2d0deb3b29.rds)) Mar05 19:56:53 clmtD
(...)
```

Caching in SpaDES is managed by the `reproducible` package, and can be generally broken down into two types: explicitly coded by the module developer, or internal to SpaDES functions.

### 2.4.1 Explicitly caching operations

Throughout the data modules we explicitly cached several data preparation operations using the functions `Cache` and `prepInputs` from the `reproducible` package.

In brief, `Cache` searches for a stored (i.e. cached) output of a given function call; if it does not find it, `Cache` executes the function call, saves its output and saves information about the function inputs and the function's code. If it does find it, `Cache` compares the present inputs and function code against their cached counterparts. In case of a mismatch, the function call is executed again and re-cached.

`prepInputs` calls `Cache` internally at several points, notably to cache spatial processing tasks (e.g. projecting and cropping spatial layers to a study area raster). Another great feature of `prepInputs` is that when it has a source URL for the target file (as when we used `prepInputs` to download species % cover and climate layers), it first checks whether the data have already been downloaded (and potentially extracted from an archive folder – `.zip` file). This is not the same thing as caching, but also avoids unnecessary downloads that can be time consuming.

Note that caching operations involving stochasticity should be avoided, as it will prevent new random outputs from being generated.

We recommend exploring the examples available in the `Cache` and `prepInputs` R documentation to learn more about their capabilities. In particular, read about `showCache`, `clearCache` and the argument `userTags`, which allow consulting and deleting cached files.

#### /!\ ATTENTION /!\

*Cache does not deal well with the `apply` family of functions, which is why we used `Map` (instead of `mapply`) to iteratively apply `prepInputs` to several climate layer URLs.*

### 2.4.2 Implicit caching of events

SpaDES offers implicit caching of events via the global parameter `.useCache`, which comes in the template modules generated by `newModule`. We call this “implicit” caching, because the developer does not need to add any caching mechanisms to the module code. SpaDES automatically reads the value of the `.useCache` parameter and activates caching in the module accordingly.

This parameter can be used to cache (or not) all or some module events (in their entirety). In our example, we cached have not cached events but passing `.useCache = ".inputObjects"` or `.useCache = "init"` would cache these events.

Notice how loading cached events (last line below) produces a slightly different message from loading other cached operations (second line below):

```
Mar05 19:58:34 spcsbn 1 speciesAbundanceData init 1
Mar05 19:58:34 spcsbn ... (Object to retrieve (bffbc48cc055c846.rds))
Mar05 19:58:35 spcsbn loaded cached copy of init event in speciesAbundanceData m
```

We have noted that `terra` objects have issues with caching due to the fact that they rely on pointers. This is a behaviour that we started observing in September 2022 and may be resolved by `terra` developers in the future.

### 2.4.3 Controlling caching without changing module code

In addition to the , which controls caching at the module level.

The user can turn caching on/off without caching module code via three different mechanisms:

- via the `.useCache` parameter – as explained above ([Implicit caching of events](#)), setting this parameter controls event caching inside a module;

- via `options("reproducible.useCache")` – setting this option to TRUE or FALSE in the global environment (`.GlobalEnv`) will affect *all* caching (inside and outside `SpaDES` modules and the simulation);
  - via the argument `spades(.useCache = ...)` – this argument behaves in the same way as the `.useCache` module parameter, but supersedes it across *all* modules (i.e. if `spades(..., .useCache = FALSE)`, caching will be turned off even if a module's `.useCache` is TRUE).
- 

## 2.5 Best practices

### 2.5.1 Reproducible package installation

When sharing code, it is good practice to provide other users with a list of necessary packages (e.g. by listing the sequence of `library` calls at the start of a script). We go a step further and advise users to provide code that automatically installs all necessary packages at the start of their controller script. In addition all modules should contain a full list of packages that they depend on, and any particular versions necessary. If `options("spades.useRequire")` is set to TRUE (the default), `SpaDES` will automatically attempt to install any packages listed across all modules if they are not installed in `.libPaths()`, or if the installed version (or branch if installing from GitHub) does not correspond to what is listed in the module `.R` script. Users can also use `Require::pkgSnapshot()` to save a list of installed packages that can be used later by `Require` to install all necessary packages in another machine (see example below).

Please beware that package installation should be done as much as possible from a clean R session especially in the context of a `SpaDES`-based project, where each module can potentially have many different dependencies, which have dependencies of their own (see, for instance, how we delayed package loading until after all modules were in place and had their dependencies checked in `Part2_SDMs.R`)

```
Require::pkgSnapshot("pkgsnapshot.txt", libPaths = .libPaths()[1]) ## you should make sure

## on another machine:
Require::Require(packageVersionFile = "pkgsnapshot.txt")

## See ?Require::pkgSnapshot() for more examples.
```

### 2.5.2 Protect yourself and others from common mistakes/problems

A developer should put in place code checks, warnings and messages that protect and warn the user against common mistakes or issues. Some of these fall in the category of *code assertions* – small tests that verify a snippet of code. More complex tests that assess whether the module (or a group of modules) is producing expected results for, e.g., an ecological point of view fall in the category of *integration tests*. Here, we only talk about code assertions.

A common assertion is to verify that input format and class conform to what the function expects. If this is not the case, the developer may add a mechanism to correct the faulty inputs (potentially with a warning or message telling the user it did so) or simply stop the computations with a meaningful error. We provide two examples in the `climateData` module, where the `climateInit` function checks whether the bioclimatic variable names are consistent between the baseline and projected climate data, and whether their raster layers match.

Other assertions can prevent undesirable function behaviours, such as the `if` statement protecting the `newModule` call in `Part2_SDMs.R`, or warn the user that something is missing early on, such as the check for `studyAreaRas` existence in the `.inputObjects` of the data modules).

Bear in mind that these are just examples assertions and integration tests are as diverse as the code they test.

### 2.5.3 Readable code

There are several guides on how to write reader-friendly code. Even if the developer is forever the sole reader of their own code, there are benefits to writing readable code. First, working on it is less tiresome. Second, we quickly forget why we wrote code in a certain away. Code that is well documented and readable is easier to “come back to” and adapt.

We follow many of the [recommendations by Hadley Wickham](#), and highlight below those that we find particularly important:

- spacing around operators;

- spacing before left parenthesis, except in a function call;
- adding curly braces after `if`, `else`, `for` and `function`, unless they are very short statements;
- thoroughly commenting the code;
- naming functions meaningfully and avoiding to re-use function names (e.g. avoid `c <- function (...) {}`, as `c` is already a `base` function).

You can automatically cleanup and format your code using the `styler` package. This package provides an Rstudio addin to easily style a block of selected code, or an entire file.

#### 2.5.4 Module documentation – module .Rmd

When modules are created using `newModule`, this function provides a template module `.Rmd` file that is meant to document the module. The template suggests a few key sections that should be part of any module's documentation. Notably, an overview of the module and of its inputs, parameters, outputs and general event flow, together with more in-depth descriptions of each of these sections.

The documentation may also contain reproducible examples of how a module can be used, although this is not always relevant. For instance, data modules are often meaningless without downstream modules that use their outputs.

We invite the reader to see the manual of our forest landscape simulation model [LandR \*Biomass\\_core\*](#), as an example of how we document some of our `SpaDES` modules.

#### 2.5.5 Coding for the future

We often make coding decisions that we regret a few months down the line. This is why as module developers, it is a good idea to think about other possible applications of a module or potential expansion avenues. For instance, trying to imagine if the module can be scaled up or transferred to different study areas, may influence the format of expected inputs and of outputs. In our example, we exported the same type of information (species % cover and climate data) as raster layers and as tables, because we could foresee that the tables could be used to store several projections in a more compact format.

### 2.5.6 Transparent models

Model transparency is not only about using open source code and making it available. Providing easy access to model data, parameters and outputs is also important. For instance, in our example we deliberately exported the fitted statistical model `sdmOut`, data (`sdmData`) and evaluation statistics (`evalOut`) so that they can be more easily inspected by the user, without needing to “dive in” the code.

SpaDES also offers the ability to save any objects that are exported to the `simList` object *without having to change module code*. To do so, the user passes a `data.frame` of object names and (potentially) the simulation times when they should be saved to the `simInit(outputs = ...)` argument. Because objects are saved as `.rds` files by default, any object class can be saved to disk (see `?outputs` for more information).

---

## 2.6 Additional notes

SpaDES is an extremely powerful family of R packages, whose potential goes well beyond what has been discussed here. We recommend going to the [SpaDES webpage](#) to find out more about the SpaDES R modelling platform, upcoming workshops and publications. See also the [Predictive Ecology Github repository](#) for a list of all available SpaDES modules and SpaDES-related packages that we maintain.

We wish to acknowledge the World Climate Research Programme, which coordinated and promoted CMIP6, and thank the climate modelling groups for producing and making available their model output, the Earth System Grid Federation (ESGF) for archiving the data and providing access, and WorldClim for downscaling and sharing climate projections and preparing bioclimatic variables.

---

**Happy SpaDESing!**

# References

- [1] Ceres Barros et al. “Empowering ecologists with a PERFECT workflow: seamlessly linking data, parameterization, prediction, validation and visualization”.
- [2] Stephen E. Fick and Robert J. Hijmans. “WorldClim 2: new 1-km spatial resolution climate surfaces for global land areas”. In: *International Journal of Climatology* 37.12 (Oct. 2017), pp. 4302–4315. ISSN: 0899-8418, 1097-0088. doi: [10.1002/joc.5086](https://doi.org/10.1002/joc.5086). URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/joc.5086>.
- [3] Robert J. Hijmans et al. “dismo: Species Distribution Modeling”. In: (2021). URL: <https://CRAN.R-project.org/package=dismo>.
- [4] Neil Cameron Swart et al. *CCCma CanESM5 model output prepared for CMIP6 ScenarioMIP*. Version Number: 20220228 Type: dataset DOI: 10.22033/ESGF/CMIP6.1317. 2019. doi: [10.22033/ESGF/CMIP6.1317](https://doi.org/10.22033/ESGF/CMIP6.1317). URL: <http://cera-www.dkrz.de/WDCC/meta/CMIP6/CMIP6.ScenarioMIP.CCCma.CanESM5>.
- [5] Wilfried Thuiller et al. “biomod2: Ensemble Platform for Species Distribution Modeling”. In: (2021).



## Appendix A

# Complete R script to Chapter 1

```
options(repos = c("https://predictiveecology.r-universe.dev/",
                 CRAN = "https://cloud.r-project.org"))

## decide where you're working
mainPath <- "~/SpaDES4Dummies_Part1"
pkgPath <- file.path(mainPath, "packages", version$platform,
                      paste0(version$major, ".", strsplit(version$minor, "[.]")[[1]][1]))
dir.create(pkgPath, recursive = TRUE)
.libPaths(pkgPath, include.site = FALSE) ## install packages in project library (proj-lib)

if (!"remotes" %in% installed.packages(lib.loc = pkgPath))
  install.packages("remotes")

if (!"Require" %in% installed.packages(lib.loc = pkgPath) ||
    packageVersion("Require", lib.loc = pkgPath) < "0.3.1") {
  remotes::install_github("PredictiveEcology/Require@2788b023ad191c29346ef8c64df71b937be3",
                         upgrade = FALSE)
}

Require::Require(c("SpaDES", "DiagrammeR"),
                require = FALSE, upgrade = FALSE,
                dependencies = TRUE, standAlone = TRUE) ## automatically downloads all p
```

```

library(SpaDES)

setPaths(cachePath = file.path(mainPath, "cache"),
         inputPath = file.path(mainPath, "inputs"),
         modulePath = file.path(mainPath, "modules"),
         outputPath = file.path(mainPath, "outputs"))

## make a list of directory paths and check paths are ok
simPaths <- getPaths()

## Let's create a self-contained module that will simulate the species' abundance
if (!dir.exists(file.path(simPaths$modulePath, "speciesAbundance"))) {
  newModule(name = "speciesAbundance", path = simPaths$modulePath)
}

if (!dir.exists(file.path(simPaths$modulePath, "temperature"))) {
  newModule(name = "temperature", path = simPaths$modulePath)
}

if (!dir.exists(file.path(simPaths$modulePath, "speciesTempLM"))) {
  newModule(name = "speciesTempLM", path = simPaths$modulePath)
}

## list the modules to use
simModules <- list("speciesAbundance", "temperature", "speciesTempLM")

## Set simulation and module parameters
simTimes <- list(start = 1, end = 10, timeunit = "year")
simParams <- list(
  speciesAbundance = list(simulationTimeStep = 1,
                          .plotInitialTime = 1),
  temperature = list(simulationTimeStep = 1,
                     .plotInitialTime = 1),
  speciesTempLM = list(statsTimestep = 5)
)

## Simulation setup
mySim <- simInit(times = simTimes, params = simParams,
                  modules = simModules, paths = simPaths)

```

```
moduleDiagram(mySim)

objectDiagram(mySim)

## run simulation
dev() # on Windows and Mac, this opens external device if using Rstudio, it is faster
clearPlot()
mySim2 <- spades(mySim, debug = TRUE)
```



## Appendix B

# Complete R script to Chapter 2

```
options(repos = c("https://predictiveecology.r-universe.dev/",
                 CRAN = "https://cloud.r-project.org"))

if (getRversion() < "4.2.1") {
  warning(paste("dismo::maxent may create a fatal error",
                "when using R version < v4.2.1 and from RStudio.\n",
                "Please upgrade R, or run this script outside of RStudio.\n",
                "See https://github.com/rspatial/dismo/issues/13"))
}

## decide where you're working
mainPath <- file.path("~/SpaDES4Dummies_Part2")
pkgPath <- file.path(mainPath, "packages", version$platform,
                      paste0(version$major, ".", strsplit(version$minor, "[.]")[[1]][1]))
dir.create(pkgPath, recursive = TRUE)
.libPaths(pkgPath, include.site = FALSE) ## install packages in project library (proj-lib)

if (!"remotes" %in% installed.packages(lib.loc = pkgPath))
  install.packages("remotes")

if (!"Require" %in% installed.packages(lib.loc = pkgPath) ||
    packageVersion("Require", lib.loc = pkgPath) < "0.3.1.9015") {
```

```

remotes::install_github("PredictiveEcology/Require@55ec169e654214d86be62a0e13e9",
                        upgrade = FALSE)
}

## Notes:
## 1) if you are working from RStudio and have an older version of base packages
## (and others) installed, you may need to run the following lines (and code above)
## in order to update these base packages
## 2) Please ensure the appropriate Rtools version is installed (see)

## there seems to be a problem with `ragg` and a forced install solves it
if (!"ragg" %in% installed.packages(lib.loc = pkgPath)) {
  install.packages("ragg")
}

Require::Require(c("PredictiveEcology/SpaDES.project@transition (HEAD)",
                  "PredictiveEcology/SpaDES.core@master (HEAD)",
                  ## these will be needed later on:
                  "ggpubr",
                  "SpaDES.tools",
                  "PredictiveEcology/SpaDES.experiment@75d917b70b892802fed0bbdb2",
                  require = FALSE, ## don't load packages yet
                  upgrade = FALSE, standAlone = TRUE))

Require::Require("SpaDES.core", install = FALSE) ## load only
setPaths(cachePath = file.path(mainPath, "cache"),
         inputPath = file.path(mainPath, "inputs"),
         modulePath = file.path(mainPath, "modules"),
         outputPath = file.path(mainPath, "outputs"))

simPaths <- getPaths() ## check that this is what you wanted

## Let's create a self-contained module that will simulate the species' abundance
if (!dir.exists(file.path(simPaths$modulePath, "speciesAbundanceData"))) {
  newModule(name = "speciesAbundanceData", path = simPaths$modulePath)
}

if (!dir.exists(file.path(simPaths$modulePath, "climateData"))) {

```

```

    newModule(name = "climateData", path = simPaths$modulePath)
}

if (!dir.exists(file.path(simPaths$modulePath, "projectSpeciesDist"))) {
  newModule(name = "projectSpeciesDist", path = simPaths$modulePath)
}

## now, let's pretend you've created your modules and each sources a series of other packages
## it's a good idea to always make sure all necessary module dependencies are installed
## this is a particularly useful line when sharing your packages with someone else.
outs <- SpaDES.project::packagesInModules(modulePath = simPaths$modulePath) ## gets list
Require::Require(c(unname(unlist(outs)),
  "DiagrammeR"),
  require = FALSE, ## don't load packages
  upgrade = FALSE, ## don't upgrade dependencies
  standAlone = TRUE,
  purge = TRUE) ## install all dependencies in proj-lib (ignore user/system)

## now load packages - SpaDES.core may have been loaded already, which is fine
Require::Require(c("reproducible", "SpaDES.core", "SpaDES.experiment"),
  install = FALSE)

## dismo needs a few tweaks to run MaxEnt
out <- preProcess(targetFile = "maxent.jar",
  url = "https://github.com/mrmaxent/Maxent/blob/master/ArchivedReleases/",
  destinationPath = simPaths$inputPath,
  fun = NA)
file.copy(out$targetFilePath, file.path(system.file("java", package="dismo"), "maxent.jar",
  overwrite = TRUE)

out <- require(rJava)
if (!out) {
  stop(paste("Your Java installation may have problems, please check.\n",
  "See https://www.java.com/en/download/manual.jsp for Java installation"))
}
## a few important options:
options(reproducible.useCache = TRUE,
  reproducible.cachePath = simPaths$cachePath,
  reproducible.destinationPath = simPaths$inputPath, ## all downloaded and pre-processed
  reproducible.verbose = TRUE)

```

```

reproducible.useTerra = TRUE, ## we want to use the terra R package
spades.moduleCodeChecks = FALSE,
spades.useRequire = FALSE)

## list the modules to use
simModules <- list("speciesAbundanceData", "climateData", "projectSpeciesDist")

## Set simulation and module parameters
simTimes <- list(start = 1, end = 5, timeunit = "year")

## we create two lists of parameters, one using the default MaxEnt
## the other a GLM
simParamsMaxEnt <- list(
  "speciesAbundanceData" = list(
    ".plots" = c("png"),
    ".useCache" = FALSE
  ),
  "climateData" = list(
    ".plots" = c("png"),
    ".useCache" = FALSE
  ),
  "projectSpeciesDist" = list(
    "statModel" = "MaxEnt",
    ".plots" = c("png"),
    ".useCache" = FALSE
  )
)
simParamsGLM <- simParamsMaxEnt
simParamsGLM$projectSpeciesDist$statModel <- "GLM"

## make a random study area.
## Here use seed to make sure the same study area is always generated
studyArea <- SpaDES.tools::randomStudyArea(size = 1e10, seed = 123)
studyAreaRas <- terra::rasterize(studyArea,
                                   terra::rast(extent = terra::ext(studyArea),
                                               crs = terra::crs(studyArea, proj =
                                               resolution = 1000))

simObjects <- list(
  "studyAreaRas" = studyAreaRas
)

```

```

)

## Simulation setup - create two simulations, one for MaxEnt another for GLM
## SpADES.experiment::experiment2, will take care of subdirectories to store outputs
mySimMaxEnt <- simInit(times = simTimes, params = simParamsMaxEnt,
                        modules = simModules, objects = simObjects,
                        paths = simPaths)
mySimGLM <- simInit(times = simTimes, params = simParamsGLM,
                      modules = simModules, objects = simObjects,
                      paths = simPaths)

moduleDiagram(mySimMaxEnt)
objectDiagram(mySimMaxEnt)

## run simulation
clearPlot(force = TRUE)    ## this forces wiping the graphics device and opening a new window

## This runs one simulation and stores outputs in the main 'outputs' folder
## - not what we want, but good for testing
# mySimOut <- spades(mySimMaxEnt, debug = TRUE)

## Better to use when spades runs error-free on the simLists
myExperiment <- SpADES.experiment::experiment2(MaxEnt = mySimMaxEnt,
                                                GLM = mySimGLM,
                                                debug = TRUE,
                                                replicates = 1,
                                                clearSimEnv = FALSE)    ## prevent removing

## save outputs
qs::qsave(myExperiment, file.path(simPaths$outputPath, paste0("myExperiment", ".qs")))

## check models
tryCatch(myExperiment$MaxEnt_rep1$sdmOut)    ## this links to an html page
sets <- par(mfrow = c(2,2))
plot(myExperiment$MaxEnt_rep1$sdmOut)
par(sets)

## check validation results for the two models
myExperiment$MaxEnt_rep1$evalOut
myExperiment$GLM_rep1$evalOut

```

```

## Run with another climate scenario - the most contrasting scenario to SSP 585
## get the original table from one of the simulations and replace the climate scenario
projClimateURLs <- copy(mySimMaxEnt$projClimateURLs)
projClimateURLs[, `:=` (URL = gsub("ssp585", "ssp126", URL),
                      targetFile = gsub("ssp585", "ssp126", targetFile))]

## this time we pass the new table or URLs to the modules, so that climate layers are updated
simObjects2 <- list(
  "studyAreaRas" = studyAreaRas,
  "projClimateURLs" = projClimateURLs
)

mySimMaxEnt2 <- simInit(times = simTimes, params = simParamsMaxEnt,
                         modules = simModules, objects = simObjects2,
                         paths = simPaths)
mySimGLM2 <- simInit(times = simTimes, params = simParamsGLM,
                      modules = simModules, objects = simObjects2,
                      paths = simPaths)

myExperiment2 <- experiment2(MaxEnt = mySimMaxEnt2,
                               GLM = mySimGLM2,
                               debug = TRUE,
                               replicates = 1,
                               clearSimEnv = FALSE)

## save outputs
qs::qsave(myExperiment2, file.path(simPaths$outputPath, paste0("myExperiment2", ".rds")))

## MaxEnt predictions across time and for each climate scenario -----
## combine plots from two distinct simulations in a single figure
## (the same can be done to compare MaxEnt and GLM, or plot all projections)

## fetch the internal plotting function instead of repeating code here
plotFun <- myExperiment$GLM_rep1@.envir$.mods$climateData$plotSpatRasterStk

## raw predictions exported by the module
sppDistProjMaxEnt <- myExperiment$MaxEnt_rep1$sppDistProj
sppDistProjMaxEnt2 <- myExperiment2$MaxEnt_rep1$sppDistProj

```

```

## we convert the raw predictions into presence absence
## using exported threshold
sppDistProjMaxEnt_PA <- as.int(myExperiment$MaxEnt_rep1$sppDistProj > myExperiment$MaxEnt)
sppDistProjMaxEnt2_PA <- as.int(myExperiment2$MaxEnt_rep1$sppDistProj > myExperiment2$MaxEnt)

## rename layers from plotting
names(sppDistProjMaxEnt) <- names(sppDistProjMaxEnt2) <- c("2001", "2021-2040", "2041-2060")
names(sppDistProjMaxEnt_PA) <- names(sppDistProjMaxEnt2_PA) <- c("2001", "2021-2040", "2041-2060")

## for a simpler plot choose only years 2001, 2041-2060 and 2081-2100
yrs <- c("2001", "2041-2060", "2081-2100")
plotMaxEnt <- plotFun(sppDistProjMaxEnt[[yrs]],
                       xlab = "Longitude", y = "Latitude",
                       plotTitle = "MaxEnt raw predictions - SSP 585") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0,1), begin = 0.25)
plotMaxEnt2 <- plotFun(sppDistProjMaxEnt2[[yrs]],
                        xlab = "Longitude", y = "Latitude",
                        plotTitle = "MaxEnt raw predictions - SSP 126") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0,1), begin = 0.25)
plotMaxEnt_PA <- plotFun(sppDistProjMaxEnt_PA[[yrs]],
                           xlab = "Longitude", y = "Latitude",
                           plotTitle = "MaxEnt presence/absence - SSP 585") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0,1), begin = 0.25)
plotMaxEnt2_PA <- plotFun(sppDistProjMaxEnt2_PA[[yrs]],
                           xlab = "Longitude", y = "Latitude",
                           plotTitle = "MaxEnt presence/absence - SSP 126") +
  scale_fill_viridis_c(na.value = "grey90", limits = c(0,1), begin = 0.25)

## organise the plots with mildest scenario first
## It is clear that MaxEnt and GLM do not agree in their prediction
plotAll <- ggpubr::ggarrange(plotMaxEnt2 + labs(title = expression(bold("Scenario - SSP 126"))),
                               y = expression(atop(bold("Raw predictions"),
                               theme(legend.title = element_blank(), legend.key.height = 10),
                               plot.title = element_text(hjust = 0.5), plot.margin = margin(0, 0, 0, 0)),
                               plotMaxEnt + labs(title = expression(bold("Scenario - SSP 585"))),
                               y = expression(atop(bold(""), ""))) +
                               theme(plot.title = element_text(hjust = 0.5), plot.margin = margin(0, 0, 0, 0)),
                               plotMaxEnt2_PA + labs(title = expression(bold(""))),
                               y = expression(atop(bold("Presence/absence"),
                               theme(plot.title = element_text(hjust = 0.5), plot.margin = margin(0, 0, 0, 0)))) +
                               theme(plot.title = element_text(hjust = 0.5), plot.margin = margin(0, 0, 0, 0)))

```

```
        theme(plot.margin = margin(0,0,0,0)),
        plotMaxEnt_PA + labs(title = expression(bold("")),
                               y = expression(atop(bold(""), "))),
        theme(plot.margin = margin(0,0,0,0)),
        legend = "right", common.legend = TRUE, labels = c(
figDir <- checkPath(file.path(simPaths$outputPath, "generalFigures"), create = TRUE)
ggsave(file.path(figDir, "MaxEntPredictions.png"), width = 13.5, height = 5.5, unit = "in")
```