Creating and integrating simple models in SpaDES

Tati Micheletti

2024-03-21

Introduction

This guide provides a simple and yet comprehensive guide to creating and then integrating SpaDES compatible modules (hereby "SpaDES modules"). It is divided in two parts. PART I focuses on the creation of a new SpaDES module from scratch, while PART II focuses on developing a new project and integrating the module created in PART I with two other SpaDES compatible modules hosted online.

PART I: creating a SpaDES compatible module from scratch

1. Planning the New Module

The first thing we will do is step away from `R` and list all needed `inputs`, desired `outputs` and `parameters` (arguments a user can control) the module will need to run. This will depend on the objective you have in mind. In our example, we want to:
1. Download abundance data (with locations and year of counts);
2. For each year, we want to convert the table into a raster file;
3. We want to check the values and rasters for each year.

For that, we will need as an `input` the dataset with counts, locations (in lat/long format) and year of survey (`abund`). We also want to `output` two objects, a raster stack of abundance through time (`allAbundaRas`), with one raster of abundance values per year (`abundaRas`). Moreover, we want to be able to define two `parameters`, which in this simple example, are mostly related to plotting: the name of the study area (`areaName`), and the first year for plotting (`.plotInitialTime`).

Once we have out list of `inputs`, `outputs` and `parameters`, we should hash out how our module will work. This means, drawing a conceptual model of (1) which `events` will happen in our analysis or simulation, (2) in which order will they happen, (3) how are `inputs`, `outputs` and `parameters` related to these events, and among themselves. This is extremely helpful in the early days of using SpaDES, while you familiarize yourself with the template. Figure 1 details what is envisioned for this module:

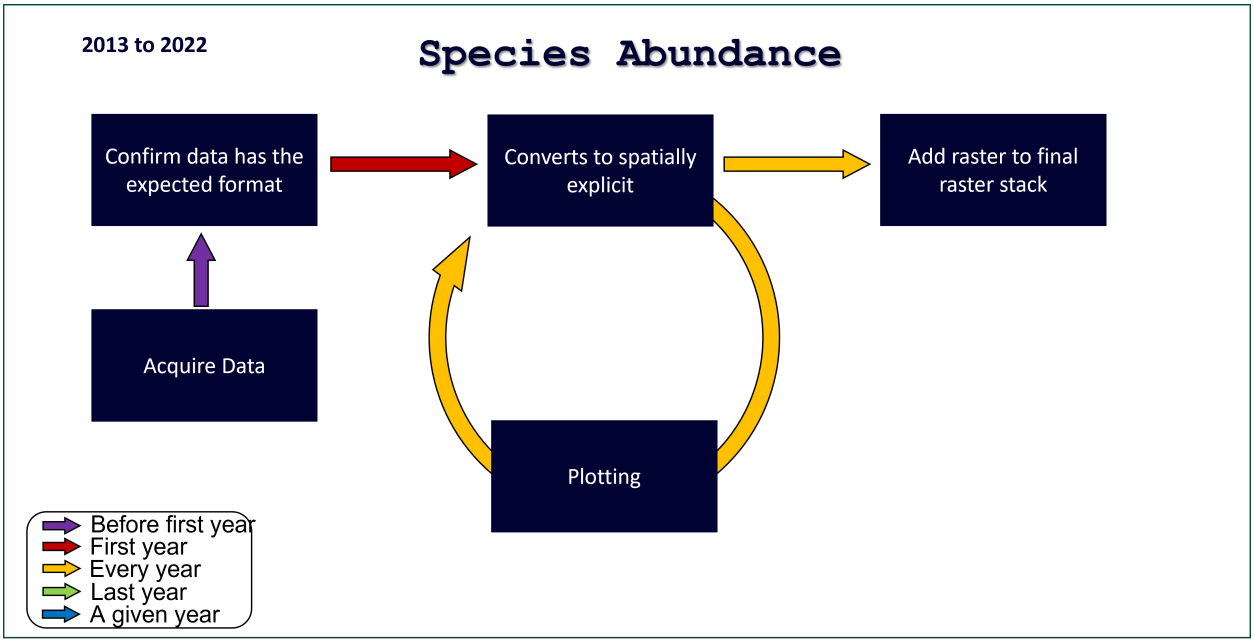


Figure 1: Module scheme presenting all important events detailing what happens to the original data.

Once this has been done, it is time to start working on the new module.

2. Installing and Loading Libraries

We will start by installing `Require`, `SpaDES.core`(to run SpaDES), and `SpaDES.project` (which will be used in PART II, to help with the workflow). `Require` is a package which tries to resolve all possible package dependencies and versioning to ensure reproducibility. This package is used in the background by SpaDES modules, but can also be used as a general approach to installing and loading libraries from both CRAN and github. We first test if `Require` has already been installed, and if not, we install it. As we are slowly transitioning `SpaDES.project` (which we will use in PART II) to CRAN, we will need some of the most up-to-date versions of these two packages. Therefore, we will install them from GitHub using `remotes`. It is good practice to restart your session after installing packages, so please proceed to it if `remotes`, `Require`, `SpaDES.project` and/or `SpaDES.core` were installed.

```
getOrUpdatePkg <- function(p, minVer = "0") {
  if (!isFALSE(try(packageVersion(p) < minVer, silent = TRUE) )) {
    repo <- c("predictiveecology.r-universe.dev", getOption("repos"))
    install.packages(p, repos = repo)
  }
}
getOrUpdatePkg("remotes")
remotes::install_github("PredictiveEcology/Require", ref = "a2c60495228e3a73fa513435290e84854ca51907",
getOrUpdatePkg("SpaDES.project", "0.0.8.9040")
install.packages("SpaDES.core")
```

3. Creating a New Module

Once SpaDES has been loaded, we use the function `newModule()` to create a new module. The function takes two arguments: `name`, the name of the module, and `path`, which is the directory where you want to create this new module. The new module is a simply a folder containing the following files:

- **Templated Module's Manual**: where the module's usage is detailed - `module.Rmd`.
- **Functions' folder**: where functions may be saved (alternatively, they can also be stored in the module's R file) - `R/`.
- **Tests folder**: where the unit tests are hosted - `tests/`.
- **Data folder**: where potential data for the module may be saved (alternatively, data can also be saved in a common inputs folder on the project level, if it may be shared across models) - `data/`.
- **README**: where basic information about the module can be added - `README.md`.
- **NEWS**: where improvements from previous module versions can be communicated - `NEWS.md`.
- **Citation**: where you define how your module should be cited - `citation.bib`.
- **LICENSE**: where you establish which type of license your module will have - `LICENSE.md`.
- **Module**: Templated script constituted of 3 Main Parts (plus potential functions) - `module.Rmd`.

Now we define our working directory and create the module's template on it.

```
wd <- reproducible::checkPath("~/SpaDES_Modules", create = TRUE)
SpaDES.core::newModule(name = "speciesAbundance", path = wd)
```

When we create the module, we get the following message and the module file (`speciesAbundance.R`), is opened in the source window. The message repeats what we discussed above.

```
New module speciesAbundance created at C:/Users/tatim/Documents/SpaDES_Modules
* edit module code in speciesAbundance.R
* write tests for your module code in tests/
* describe and document your module in speciesAbundance.Rmd
* tell others how to cite your module by editing citation.bib
* choose a license for your module; see LICENSE.md
Using RStudio; may need to open manually e.g., with file.edit or file.show
file.edit('C:/Users/tatim/Documents/SpaDES_Modules/speciesAbundance/speciesAbundance.R')
Using RStudio; may need to open manually e.g., with file.edit or file.show
file.edit('C:/Users/tatim/Documents/SpaDES_Modules/speciesAbundance/speciesAbundance.Rmd')
Warning message:
In newModuleTests(name = name, path = path, open = !isFALSE(open),  :
  The `testthat` package is required to run unit tests on modules.
```

4. Editing the New Module

Now we can start working on the new module's file. We will skip the module documentation (manual, citation, readme and license) and tests due to time constraints, but you should never skip these when developing your modules, so they can be used by others.

4.1. Descriptive Metadata

We will start by filling out the descriptive metadata for our module. This part of the module is composed of the following parts:

- **name**: Module's name, which was pre-filled when the module was created.
- **description**: Module's description, where details about the module can be found, such as why the module was created and its basic working mechanism. Almost as an abstract of the modules' usefulness. If you want to avoid long lines, you can wrap the description with the function `paste0()`, as seen below.
- **keywords**: Keywords for helping the module getting properly indexed. If more than one keyword is added, it is important to remember to concatenate all words using a `c("keyword 1", "keyword 2", "keyword 3")`.
- **authors**: Add information on author(s), including contact. This is the best way to ensure someone can contact you if they want to discuss any specific information about your module.
- **childModules**: This is used when the current module represents a parent (i.e., grouping) module. It's a mechanism to simplify getting a group of modules which are expected to work together. We won't be using it in this simple example. Although in the descriptive metadata part, this is also functional.
- **version**: We follow a software development versioning system for the modules, combining the Numeric status and the Numeric 90+ scheme. This system has 4 codes, separated by `.`. The first number indicates the major structureal changes to the module, the second indicates minor changes, but generally in more than one place, the third indicates bug fixes, and the last indicates punctual enhancements or bugfixes. With such system, we hope modules will be continuously updated and the exact version used for publications can be backtraced.
- **timeframe**: Indicates the original or exemplified time frame of the module (i.e., from 2013 to 2022).
- **timeunit**: This is an important information and also functional when modules have divergent time units. SpaDES converts all times internally to seconds in order to allow the integration of modules happening at different time scales.
- **citation**: Indicates the file where the citation for the module can be found. The citation template format is BibTeX.
- **documentation**: Indicates all files that are related to the module in terms of documentation. If, for example, the user decides to create a vignette for the module apart from the manual, it can be added here.

Once the descriptive metadata has been updated for our example module `speciesAbundance`, it should look similar to this:

```
  name = "speciesAbundance",
  description = paste0("This is a simple example module on how SpaDES work. It uses made",
                       " up data and is partially based on the example publised by ",
                       "Barros et al., 2022 (DOI: 10.1111/2041-210X.14034)"),
  keywords = c("example", "SpaDES tutorial"),
  authors = structure(list(list(given = "Tati", family = "Micheletti",
                                role = c("aut", "cre"),
                                email = "tati.micheletti@gmail.com",
                                comment = NULL)),
                      class = "person"),
  childModules = character(0),
  version = list(speciesAbundance = "0.0.0.9000"),
  timeframe = as.POSIXlt(c(2013, 2022)),
  timeunit = "year",
  citation = list("citation.bib"),
  documentation = list("NEWS.md", "README.md", "speciesAbundance.Rmd"),
```

4.2. Functional Metadata

An important part of a SpaDES module is the functional metadata. This is a hybrid approach between human-readable information and code used by SpaDES to schedule the events happening across modules. This part of the module is composed of the following parts:
- **reqdPkgs**: names of the packages (potentially with version) needed for the module to run. If the packages are not specified, SpaDES will not load them for the module to use and any functions dependent on the omitted packages will fail.

Example: in our example, we will need the following packages:
`data.table` (for organizing and working on our dataset), `terra` (for converting our dataset to a spatially explicit dataset) and `ggplot2` (which is default in the template, so there is no need to add it again, similarly to `SpaDES.core`, the package that orchestrates the whole system).

- **parameters**: this is one of the crucial parts of the functional metadata, together with `inputObjects` and `outputObjects`. It defines the parameters a user can pass to the module (e.g., arguments to a function), a default value when the user doesn't provide one, the expected range a parameter can take if e.g. numeric and the description of the parameter. This is defined by the function `defineParameter()` and the template provides several potential parameters of interest for the module developer. These predefined parameters are generally preceded by `.`, but are not required to be used nor removed from the module if not used. Module developers are also expected to add their own parameters of interest which do not have to be preceded by `.`. It is necessary to have at least one parameter declared in this session, even if it is not used. We suggest therefore, that the predefined parameters are not removed.

Example: in our example case, we will use as parameters two pre-defined example provided by the template, and we will add one parameter ourselves: `.plotInitialTime`, `.plotInterval`, and `areaName`. These will mainly be used for the plotting event and saving in this simple case.

- **inputObjects**: input objects are the objects expected to be present for the module to run. These are similar, for example to a dataset or a spatial object that will be passed as a function argument. These are generally provided by the user, but should have a default in case the user does not provide it, so that the module can run independently of providing data, similarly to a test-run. The importance of providing

such default (which will be discussed below) cannot be overstated: this is crucial for users of your module to be able to see the module functioning and understand its mechanisms. The input objects are defined by the function `expectsInput()` with all arguments in quotes: object name (`objectName`), the class of the expected input (`objectClass`), a description of the expected object (`desc`) and, if available online, the source address in the form of a URL (`sourceURL`). The last can be used by SpaDES to retrieve the object using the function `prepInputs()` from the package `reproducible` (which is a part of the SpaDES metapackage)‘.

Example: in our example, the module expects only a data frame with the following columns: `counts` (abundance in a numeric form), `years` (year of the data collection in numeric form) and coordinates in latlong system (two columns, `lat` and `long`, indicating latitude and longitude, respectively).

- **outputObjects**: these are the objects created by the module. Similarly to the input objects, here we should also provide object name, object class and description. Failing to provide the `outputObjects` will result in the simulation not returning these at the end.

Example: in our example, we will create two outputs. The first one, is named `abundaRas`, which is a raster object of spatially explicit abundance data for a given year, compatible with `terra` (i.e., `SpatRaster` object). The second one, is named `allAbundRas`, which is a raster stack of all `abundaRas`.

Once the functional metadata has been updated for our example module `speciesAbundance`, it should look similar to this (note that the parameters not used were removed just to improve clarity):

```
parameters = bindrows(
  defineParameter(paramName = ".plotInitialTime",
                  paramClass = "numeric",
                  value = start(sim),
                  min = start(sim),
                  max = end(sim),
                  desc = paste0("Describes the simulation time at which the first plot event",
                         "should occur.")),
      defineParameter(paramName = ".plotInterval",
                      paramClass = "numeric",
                      value = 5,
                      min = NA,
                      max = NA,
                      desc = "Describes the simulation time interval between plot events."),
    defineParameter(paramName = "areaName",
                    paramClass = "character",
                    value = "Riparian_Woodland_Reserve",
                    min = NA,
                    max = NA,
                    desc = paste0("Name for the study area used"))
),
inputObjects = bindrows(
  expectsInput(objectName = "abund",
               objectClass = "data.frame",
               desc = paste0("data frame with the following columns: `counts` (abundance in a",
                             "numeric form), `years` (year of the data collection in numeric",
                             "form) and coordinates in  latlong system (two columns, `lat` and",
                             "`long`, indicating latitude and longitude, respectively)"),
```

```
                  sourceURL = "https://zenodo.org/records/10885997/files/abundanceData.csv")
  ),
  outputObjects = bindrows(
    createsOutput(objectName = "abundaRas",
                  objectClass = "SpatRaster",
                  desc = "A raster object of spatially explicit abundance data for a given year"),
    createsOutput(objectName = "allAbundaRas",
                  objectClass = "SpatRaster",
                  desc = "a raster stack of all `abundaRas`")
  )
```

Note that for the parameter `.plotInitialTime` we add the `start()` function instead of a number indicating the start of the simulation. This helps maintain flexibility in the module (i.e., avoid hardcoding the time) in case we decide to start our simulation at a different point in time as presented later. The `start()`, as well as the `end()` and the `time()` functions are "shortcuts" SpaDES module developers can use to access the provided start, end and current time (i.e., year) of the simulation. They are extremely handy to use during coding of the module as we will demonstrate below. The argument of these functions is `sim`, which is a special class of list-type object (i.e., `simList`) which we will detail below.

4.3. Adding default values to `.inputObjects`

As mentioned, an important step to help modules be used is to provide default objects to allow for a potential user to test the module. This can be done inside the module structure itself, under the function `.inputObjects()` (located at the end of the template). As SpaDES is modular and an object might be provided by the user or by another module, or may not be. It is useful, therefore, to know to which case the object pertains to and a function that can help with that is `suppliedElsewhere()`. This function can be used as a check to determine whether the module needs to proceed in getting and assigning its default value if the object is not being supplied by the user or by another module.
Example: in the case of our example, we should add a default to our dataset `abund`. As mentioned before, the function `prepInputs()` can be very useful here to download and prepare the data expected by the module. This is a very versitile function that will return an `R` object that will have resulted from the running of `preProcess()` (function that identifies the source of the data, and download it), and `postProcess()` or `postProcessTo()` (which are functions that deal with loading the data with the specified or needed function for the data type). For example, if the data to be downloaded is a GIS object, it may have been cropped, reprojected, "fixed", and masked to a provided study area before it is assigned to an object. If it is a table, as in our case, it will be loaded with the specified function, in this case, `data.frame()`. We also add a warning so the user can know that the data was not supplied and is being retrieved by the provided url in the metadata. This is what the loading of this dataset will look like when added as a default inside `.inputObjects()`:

```
if (!suppliedElsewhere(object = "abund", sim = sim)) {
  sim$abund <- prepInputs(url = extractURL("abund"),
                                  targetFile = "abundanceData.csv",
                                  destinationPath = dPath,
                                  fun = "data.frame",
                                  header = TRUE)
  warning(paste0("abund was not supplied. Using example data"), immediate. = TRUE)
}
```

The function `prepInputs()` generally expects a url address where it can find the object of interest. As we

provided the url where the abundance dataset can be found in the metadata (i.e., the argument `sourceURL` in `expectsInput()`), we can here use the function `extractURL()` to get the url where the object is stored from the metadata. This avoids copy and paste in several places, as well as typos in the url, by keeping the metadata the canonical source for this information. Another interesting feature of `prepInputs()` is that it allows files to be hosted in Google Drive, provided the user has the package `googledrive` installed. The function can also usually extract the file name and function to load the object correctly, in case these are not provided by the user. It is a highly recommended function to be used throughout due to its flexibility.

One last important detail which can be noted here: the assignment of `abund` to `sim`. We will now discuss the meaning of `simList`, a very special form of list.

4.3.1. The heart of SpaDES: the `simList`

The simList (named `sim` in the templates) is a list containing the minimum components of a SpaDES simulation. It is created by initializing a simulation using `simInit()` and returned when running a simulation via a `spades()` call. This list contains all information needed for organizing the events (described below), and contains all parameters, inputs and outputs created. Through time and across modules, only objects created and stored in this special list are available for the simulations.

4.4. Events

The events define what will be done by the module (Figure 1). Each module may contain an arbitrary number of events, and each event consists of two parts:

(1) *what* to do right now: we generally execute a function returning an object that is appended to `sim`;

(2) *when* to do it again: we generally schedule the same event in the event itself using the function `scheduleEvent()`.

The first and only event that is mandatory to have in all modules is called `init`. This event is the one responsible for scheduling all other events, and happens for all modules before any other event. It can be used to make assertions and checks, for example, or even just schedule the next events. In our example the `init` event will be used to confirm the data has the expected columns, and a desired format (`data.table`), as well as schedule the next events. Note that for the last event, we create an object named `lastYearOfData` and use it for the scheduling of the event. This way, when we integrate other modules, we still have this event happening at the "end of this module", as opposed to "at the end of all simulations", as these might not coincide, as we will see.

The next event will then happen in all years, and is going to be named `tableToRasters`. This event will convert the table into rasters using the GIS information provided on the table (`lat` and `long` columns). In the same event, the raster recently created will be appended to the already existing ones, incrementing the raster stack that will hold the abundance rasters of all years.

Followed by this, we will `plot` both the original data from the first year up to the current year (histogram with its distribution) and the newly created raster, using the parameter `areaName` for the plot title. This plotting function will be dependent on the parameter `.plotInitialTime`, which will used to schedule the start of plotting events.

Our events will (temporarily) have the following format:

```
doEvent.speciesAbundance = function(sim, eventTime, eventType) {
  switch(
    eventType,
    init = {
      ### check for more detailed object dependencies:
      ### (use `checkObject` or similar)

      # do stuff for this event
      # Check the data
      if (!is(sim$abund, "data.table"))
        sim$abund <- data.table(sim$abund)

      if (!all("abundance" %in% names(abund),
               "years" %in% names(abund),
               "lat" %in% names(abund),
               "long" %in% names(abund)))
        stop("Please revise the column names in the abundance data")

      # schedule future event(s)
      sim <- scheduleEvent(sim, time(sim), "speciesAbundance", "tableToRasters")
      sim <- scheduleEvent(sim, P(sim)$.plotInitialTime, "speciesAbundance", "plot")
    },
    tableToRasters = {
      # ! ----- EDIT BELOW ----- ! #
      # do stuff for this event


      # schedule future event(s)
      sim <- scheduleEvent(sim, time(sim) + 1, "speciesAbundance", "tableToRasters")

      # ! ----- STOP EDITING ----- ! #
    },
    plot = {
      # ! ----- EDIT BELOW ----- ! #
      # do stuff for this event

      # schedule future event(s)
      sim <- scheduleEvent(sim, time(sim) + 1, "speciesAbundance", "plot")

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

Now that we have filled the `init` event and created and scheduled the other ones, we need to define the functions to do the following tasks that are still missing for each event:

tableToRasters:

(1) convert the table into rasters (named `abundaRas` as defined in the `createdOutputs()`) using the GIS information provided on the table;

(2) append the recently created raster to the full raster stack (named `allAbundaRas`)

plot:

(3) plot the newly created raster (`abundaRas`), using the parameter `areaName` and year for title.

(4) save the raster stack to the `outputs` folder.

Each one of these numbered tasks will be converted into a function. At this point, there are two options: (1) keep all functions in the module's file as the template suggests (see the functions between `doEvent.speciesAbundance` and `.inputObjects`) or (2a) save all functions or (2b) each one of them separately in the module's `R/` folder, which gets sourced at the beginning of each module run. Although this might be a matter of personal choice, in the case of complex modules, keeping each function in a separate file named after the function in the R folder might prove useful to keep the module's organization, easier to debug and easier for others to read through the module.

4.5. Event functions

IMPORTANT: Avoid using the same names for functions and events. It might get confusing to debug.

We will now create all functions for the module, except for function (4), which we will use a pre-existing function from the package `terra`. We will opt, due to the module's simplicity, to host the functions below in the same script. However, a good common practice for more complex modules is to store the functions in the `R/` module folder. This helps keeping the main code of the module cleaner and easier to debug and improve. As our `init` event will not need any functions, we will start with the next event, `tableToRasters`:

(1) convert the table into rasters (named `abundaRas` as defined in the `createdOutputs()`) using the GIS information provided on the table: `convertToRaster(dataSet, currentTime)`

```
convertToRaster <- function(dataSet, currentTime){
  ras <- rast(dataSet[time == currentTime, c("lat", "lon", "abundance")], type="xyz")
  crs(ras) <- "GEOGCRS[\"WGS 84 (CRS84)\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOI
return(ras)
}
```

(2) append the recently created raster to the full raster stack (named `allAbundaRas`):`appendRaster(allAbundanceRasters, newRaster)`

```
appendRaster <- function(allAbundanceRasters, newRaster){
  if (is.null(allAbundanceRasters)){
  # This would happen in the first time we are appending the raster
  allAbundanceRasters <- newRaster
} else {
```

```
  # This would happen in the next times
  allAbundanceRasters <- c(allAbundanceRasters, newRaster)
}
return(allAbundanceRasters)
}
```

Next, we will move to the event `plot`:

(3) plot the newly created raster (`abundaRas`), using the parameter `areaName` and year for title. This function is not going to be built in the module. It is, instead, a function from the package `terra`. It is here written only to show how we will use it in the `plot` event: `plot(x, main)`

```
plot(ras, main = paste0(P(sim)$areaName,": ", time(sim)))
```

(4) save the rasters (`allAbundaRas`) to the outputs folder at the last year: `saveAbundRasters(allAbundanceRasters, savingName, savingFolder)`

```
saveAbundRasters <- function(allAbundanceRasters, savingName, savingFolder){
  terra::writeRaster(x = allAbundanceRasters,
                     filetype = "GTiff",
                     filename = file.path(savingFolder, paste0(savingName, ".tif")),
                     overwrite = TRUE)
  message(paste0("All rasters saved to: \n",
                 file.path(savingFolder, paste0(savingName, ".tif"))))
}
```

Now that we have all functions written, we add them to the `speciesAbundance.R` module file. The functions part of the file should then look like this, after replacing the example functions that come in the template:

```
convertToRaster <- function(dataSet, currentTime){
  ras <- rast(dataSet[time == currentTime, c("lat", "lon", "abundance")], type="xyz")
  crs(ras) <- "GEOGCRS[\"WGS 84 (CRS84)\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOI
  return(ras)
}

appendRaster <- function(allAbundanceRasters, newRaster){
  if (is.null(allAbundanceRasters)){
    # This would happen in the first time we are appending the raster
    allAbundanceRasters <- newRaster
  } else {
    # This would happen in the next times
```

```
    allAbundanceRasters <- c(allAbundanceRasters, newRaster)
  }
  return(allAbundanceRasters)
}

saveAbundRasters <- function(allAbundanceRasters, savingName, savingFolder){
  terra::writeRaster(x = allAbundanceRasters,
                     filetype = "GTiff",
                     filename = file.path(savingFolder, paste0(savingName, ".tif")),
                     overwrite = TRUE)
  message(paste0("All rasters saved to: \n",
                 file.path(savingFolder, paste0(savingName, ".tif"))))
}
```

4.5.1. Completing the Events with the needed functions

At last, we will add call the functions in the specific events needed and revise the scheduling. As we want our module to be integrated with any other modules that may have, for example, longer time series, we need to make sure that our funtions will only run while our module has data. Therefore, we will add a conditional scheduling of future events to check for the data availability before scheduling future actions for both `tableToRasters` and `plot` events. Here is what the module code should look like in the end, after all editing done:

```
## 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 = "speciesAbundance",
  description = paste0("This is a simple example module on how SpaDES work. It uses made up data",
                       "and is partially based on the example published by Barros et al., 2022",
                       "(https://besjournals.onlinelibrary.wiley.com/doi/full/10.1111/2041-210X.14034)"
  keywords = c("example", "SpaDES tutorial"),
  authors = structure(list(list(given = "Tati", family = "Micheletti", role = c("aut", "cre"),
                                email = "tati.micheletti@gmail.com", comment = NULL)),
                      class = "person"),
  childModules = character(0),
  version = list(speciesAbundance = "1.0.0"),
  timeframe = as.POSIXlt(c(2013, 2022)),
  timeunit = "year",
  citation = list("citation.bib"),
  documentation = list("NEWS.md", "README.md", "speciesAbundance.Rmd"),
  reqdPkgs = list("SpaDES.core (>= 2.0.3)", "terra", "reproducible", "ggplot2"),
  parameters = bindrows(
    defineParameter(".plotInitialTime", "numeric", start(sim), start(sim), end(sim),
                    "Describes the simulation time at which the first plot event should occur."),
    defineParameter(".plotInterval", "numeric", 1, NA, NA,
                    "Describes the simulation time interval between plot events."),
    defineParameter("areaName", "character", "Riparian_Woodland_Reserve", NA, NA,
                    "Name for the study area used")
```

```
  ),
  inputObjects = bindrows(
    expectsInput(objectName = "abund",
                 objectClass = "data.frame",
                 desc = paste0("data frame with the following columns: `counts` (abundance in a",
                               "numeric form), `years` (year of the data collection in numeric",
                               "form) and coordinates in  latlong system (two columns, `lat` and",
                               "`long`, indicating latitude and longitude, respectively).",
                               "In this example, we use the data v.2.0.0"),
                 sourceURL = "https://zenodo.org/records/10877463/files/abundanceData.csv")
  ),
  outputObjects = bindrows(
    createsOutput(objectName = "abundaRas", objectClass = "SpatRaster",
                  desc = "A raster object of spatially explicit abundance data for a given year"),
    createsOutput(objectName = "allAbundaRas", objectClass = "SpatRaster",
                  desc = "a raster stack of all `abundaRas`")
  )
))

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

doEvent.speciesAbundance = function(sim, eventTime, eventType) {
  switch(
    eventType,
    init = {
      ### check for more detailed object dependencies:
      ### (use `checkObject` or similar)

      # do stuff for this event
      # Check the data
      if (!is(sim$abund, "data.table"))
        sim$abund <- data.table(sim$abund)

      if (!all("abundance" %in% names(sim$abund),
               "years" %in% names(sim$abund),
               "lat" %in% names(sim$abund),
               "long" %in% names(sim$abund)))
        stop("Please revise the column names in the abundance data")

      # schedule future event(s)
      sim <- scheduleEvent(sim, time(sim), "speciesAbundance", "tableToRasters")
      sim <- scheduleEvent(sim, P(sim)$.plotInitialTime, "speciesAbundance", "plot")
    },
    tableToRasters = {
      # ! ----- EDIT BELOW ----- ! #

      # do stuff for this event
      sim$abundaRas <- convertToRaster(dataSet = sim$abund,
                                       currentTime = time(sim),
                                       nameRaster = paste0(P(sim)$areaName, ": ", time(sim)))
      sim$allAbundaRas <- appendRaster(allAbundanceRasters = sim$allAbundaRas,
                                       newRaster = sim$abundaRas)
```

```
      # schedule future event(s)
      if (time(sim) < max(as.numeric(sim$abund[, years])))
        sim <- scheduleEvent(sim, time(sim) + 1, "speciesAbundance", "tableToRasters")

      # ! ----- STOP EDITING ----- ! #
    },
    plot = {
      # ! ----- EDIT BELOW ----- ! #
      # do stuff for this event
      terra::plot(sim$abundaRas, main = paste0(P(sim)$areaName, ": ", time(sim)))

      if (time(sim) == max(as.numeric(sim$abund[, years]))){
        saveAbundRasters(allAbundanceRasters = sim$allAbundaRas,
                         savingName = P(sim)$areaName,
                         savingFolder = Paths$output)
      }

      # schedule future event(s)
      if ((time(sim) + P(sim)$.plotInterval) < max(as.numeric(sim$abund[, years])))
        sim <- scheduleEvent(sim, time(sim) + P(sim)$.plotInterval,
                             "speciesAbundance", "plot")

      # ! ----- STOP EDITING ----- ! #
    },
    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 below.

convertToRaster <- function(dataSet, currentTime, nameRaster){
  ras <- rast(dataSet[years == currentTime, c("lat", "long", "abundance")], type="xyz")
  terra::crs(ras) <- "GEOGCRS[\"WGS 84 (CRS84)\",\n    DATUM[\"World Geodetic System 1984\",\n        E
  names(ras) <- nameRaster
  return(ras)
}

appendRaster <- function(allAbundanceRasters, newRaster){
  if (is.null(allAbundanceRasters)){
    # This would happen in the first time we are appending the raster
    allAbundanceRasters <- newRaster
  } else {
    # This would happen in the next times
    allAbundanceRasters <- c(allAbundanceRasters, newRaster)
  }
  return(allAbundanceRasters)
}

saveAbundRasters <- function(allAbundanceRasters, savingName, savingFolder){
  terra::writeRaster(x = allAbundanceRasters,
```

```
                    filetype = "GTiff",
                    filename = file.path(savingFolder, paste0(savingName, "_abundance.tif")),
                    overwrite = TRUE)
  message(paste0("All rasters saved to: \n",
                 file.path(savingFolder, paste0(savingName, ".tif"))))
}


.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 defineModule.
  # 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 function.
  # Note: the module developer can check if an object is 'suppliedElsewhere' to
  # selectively skip unnecessary steps because the user has provided those inputObjects in the
  # 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 file from url in source

  # }

  #cacheTags <- c(currentModule(sim), "function:.inputObjects") ## uncomment this if Cache is being use
  dPath <- asPath(getOption("reproducible.destinationPath", dataPath(sim)), 1)
  message(currentModule(sim), ": using dataPath '", dPath, "'.")

  # ! ----- EDIT BELOW ----- ! #
  if (!suppliedElsewhere(object = "abund", sim = sim)) {
    sim$abund <- reproducible::prepInputs(url = extractURL("abund"),
                            targetFile = "abundanceData.csv",
                            destinationPath = dPath,
                            fun = "read.csv",
                            header = TRUE)
    warning(paste0("abund was not supplied. Using example data"), immediate. = TRUE)
  }
  # ! ----- STOP EDITING ----- ! #
  return(invisible(sim))
}
```

One important note is that all new objects being created (i.e., assigned to `sim` in the form of `sim$object <- function1(args)`) are defined in the `createdOutputs()`. If an object created is not assigned to sim or described in `createdOutputs()` it will not be available at the end of the simulation. This is, for example, the case of the `lastYearOfData` object created in the `init` but not assigned to the `simList`.

PART II: creating a SpaDES project to control your module(s)

1. Scripting the project

SpaDES has additional packages that allow, for example, for the creation of a project. The `SpaDES.project` package was designed with a PERFICT approach in mind (McIntire et al. 2022). The advantage of having such a project to run your module as opposed to scripts sourcing your module through the `simInitAndSpades()` or `simInit()` followed by `spades()` call is that the the whole workflow is done in one script, where all needed components need to be present for the simulation to run. There is no pre-installing or loading libraries, setting up paths, creating objects to be passed to the script, or manipulating data. This eliminates "secret handshakes" and allow for your project to be tested, run and improved by others.

Here we provide enough information to set up a project with our example in mind. More detailed guides on using git and GitHub, installing dependencies, R and RStudio, finding available SpaDES modules, and setting up a `SpaDES project` can be found in the vignettes Session of the Predictive Ecology website.

The main function in the `SpaDES.project` package is called `setupProject()`. The primary five objectives of this function are:

1. Preparation for `SpaDES.core::simInit`: This functions is designed to set the stage for a smooth transition into `SpaDES.core::simInit`, i.e., the initiation of a collection of SpaDES modules. After a `out <- setupProject()` call, the return can be passed directly to `do.call(SpaDES.core::simInitAndSpades, out)`.

2. Simplicity for Beginners, Versatility for Experts: The functions are crafted to be approachable for beginners, offering simplicity, while at the same time, providing the power and flexibility needed to address the demands of highly intricate projects, all within the same structural framework.

3. Handling Package Complexity: These functions address the complexities associated with R package installation and loading, especially when working with modules created by various users.

4. Creating a Standardized Project Structure: They facilitate the establishment of a consistent SpaDES project structure. This uniformity eases the transition from one project to another, regardless of project complexity, promoting a seamless and efficient workflow.

5. Minimizing .GlobalEnv Assignments: An important goal is to encourage best practices by reducing the need for assignments to the .GlobalEnv. This practice fosters clean, maintainable code that remains reproducible even as project complexity grows.

This function performs the following tasks:
- `paths` - Sets standardized paths for the simulation;
- `modules` - Either downloads them from a cloud repository or uses locally available modules;
- `packages` - Installs and/or loads (using the `require` argument) packages not already identified in the metadata of the `modules`;
- `params` - Sets parameter values for any of the `modules`;
- `options` - Configures basic `R options`.

More specifically, this function orchestrates a series of operations in the following order: `setupPaths`, `setupModules`, `setupPackages`, `setupOptions`, `setupSideEffects`, `setupParams`, and `setupGitIgnore`. This sequence accomplishes several tasks, including the creation of folder structures, installation of missing packages listed in either the `packages` or `require` arguments, loading of packages (limited to those

specified in the `require` argument), configuration of options, and the download or validation of modules. Additionally, it returns elements that can be directly passed to `simInit` or `simInitAndSpades`, specifically `modules`, `params`, `paths`, `times`, and any named elements passed to ... (`dots`). If desired, this function can also modify the `.Rprofile` file for the project, ensuring that each time the project is opened, it adopts specific `.libPaths()`. This sequence of events allows users to leverage settings (i.e., objects) that are established before others as explained below.

For user convenience, there are several auxiliary elements, as described in the help file of the function: `?setupProject()`. The output from `setupProject()` is a list containing several named elements. These elements are designed to be passed directly to `simInit` to initialize a `simList` and, potentially, a `spades` call.

Details on why this function is so useful are described in details in its vignette, but in general, it accepts Different Argument Types, such as `url` and local file paths, it performs Sequential Argument Processing, which means that prior value can be leveraged by a subsequent one *within the same function call*, and it Handles Missing Values, enabling users to source a script passing a required argument, which can be helpful for submitting jobs. Setting up a project in this fashion also creates a separate library for the project, which also improves reproducibility and avoids package versioning clashes. More advantages and details on the function can be found in it's documentation by running `?setupProject` in the `R` console.

As there isn't a specific template, as `setupProject()` is a function, we will provide the templates for our example here. It is good practice to restart your session frequently to improve reproducibility. In order to demonstrate the flexibility of SpaDES, we will establish a couple of different `runs`. These can, in other cases, be seen as different replicates, or different scenarios. Each run has an unique name (`runName`) and a unique set of parameters (and usually, a unique `outputs` folder, as defined by the user). We will save each run with different `setupProject()` arguments. The first one will be the most straight forward case using all default parameters and objects.

First, we will restart our session and then set our desired directory. This is where a folder with the project will be created, with the name specified in the argument `paths = list(projectPath = 'projectName')`. This should either be created on a disk that has enough space, depending on the simulation needs, or scratch paths should be provided (i.e., `terra::terraOptions(tempdir = "~/scratch/terra")` before `setupProject()` call, and in `paths = list(scratchPath = "~/scratch/")` of `setupProject()`).

```
wd <- reproducible::checkPath("~/SpaDES_Projects", create = TRUE)
setwd(wd)
```

Once the home directory has been set, we will call the `setupProject()` function. At first, we will use the default values provided in the module. We will only need to provide the (1) `paths` to the project and outputs (as we want each call to the project with variations being saved to a different folder), the (2) `modules` that will be used in the project, and the (3) `times` for which the analysis or simulations will happen. We can restart our session before setting up the project, and don't need to load any libraries at the moment. All needed libraries will be installed and loaded during the simulation call.

At this moment, we have two options to setup where the `modules` of our project will be hosted, depending on user's preferences: (1) local module repository. Mixing GitHub and locally provided modules is also supported; the function will download the repositories to your provided `modulePath`, or (2) cloud-based module repository.

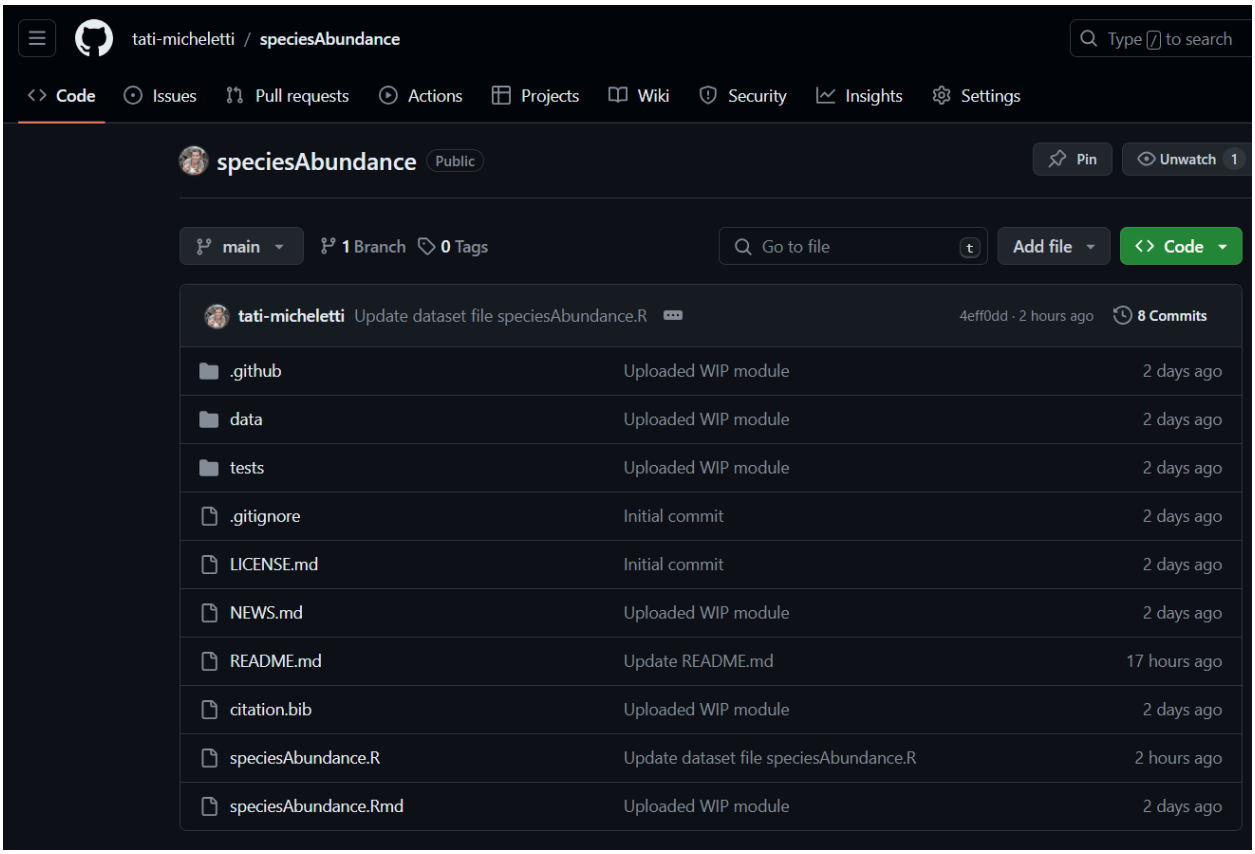
OPTION 1. *Local module repository*: The first option is to keep and use your modules locally. As we already have an existing module (i.e., `speciesAbundance`), we can either (A) pass the correct location of

the existing module in the argument such as `paths = list(modulePath = "~/Documents/GitHub/")`; or (B) manually create the folder specified in `projectPath` (i.e., `integratingSpaDESmodules`) and move the module directory to a folder named `modules`. The advantage of the *Option B* is that you keep all modules used in a given project in one place, facilitating any improvements and debugging. In Option 2 (A), the function call will be similar to this. Note that if the dependency packages have not yet been installed, they will now. Please run one function at a time to ensure any potential warnings or errors are seen. Make sure you watch the Plots window!

```
runName <- "defaults"
out <- SpaDES.project::setupProject(
  runName = runName,
  paths = list(projectPath = "integratingSpaDESmodules",
               modulePath = file.path(dirname(wd), "SpaDES_Modules"),
               outputPath = file.path("outputs", runName)),
  modules = c("speciesAbundance"),
  times = list(start = 2013,
               end = 2022),
  updateRprofile = FALSE)

snippsim <- do.call(SpaDES.core::simInitAndSpades, out)
```

OPTION 2. *Cloud-based module repository*: The first option is **strongly suggested**. Once the module is ready, the user may uploads it to GitHub and uses the web address to indicate in the project function where this module can be found. This is the preferred option, as it grants other users access to the modules and increases reproducibility. In this case, the user will have to setup an account in GitHub, create a repository with the name of the module (i.e., `speciesAbundance`), and push the module we created to it. Please note that the repository should contain all files and folder structure as these were in the original module's folder, not the folder containing these (Figure 2):



Note on using git submodules: For users who wish to have their projects uploaded to GitHub, but maintain modularity (i.e., keep each module in its own repository to inherit module's updates), there is an experimental argument in `setupProject()` that allows the modules to be setup as `gitsubmodules`. This argument is `useGit = "sub"` and will work if the project (i.e., `integratingSpaDESmodules`) is setup as a git repository. This can be done manually by the user, or interactively from R. The function guides the user on the steps needed, although basic knowledge on how `git` works is required.

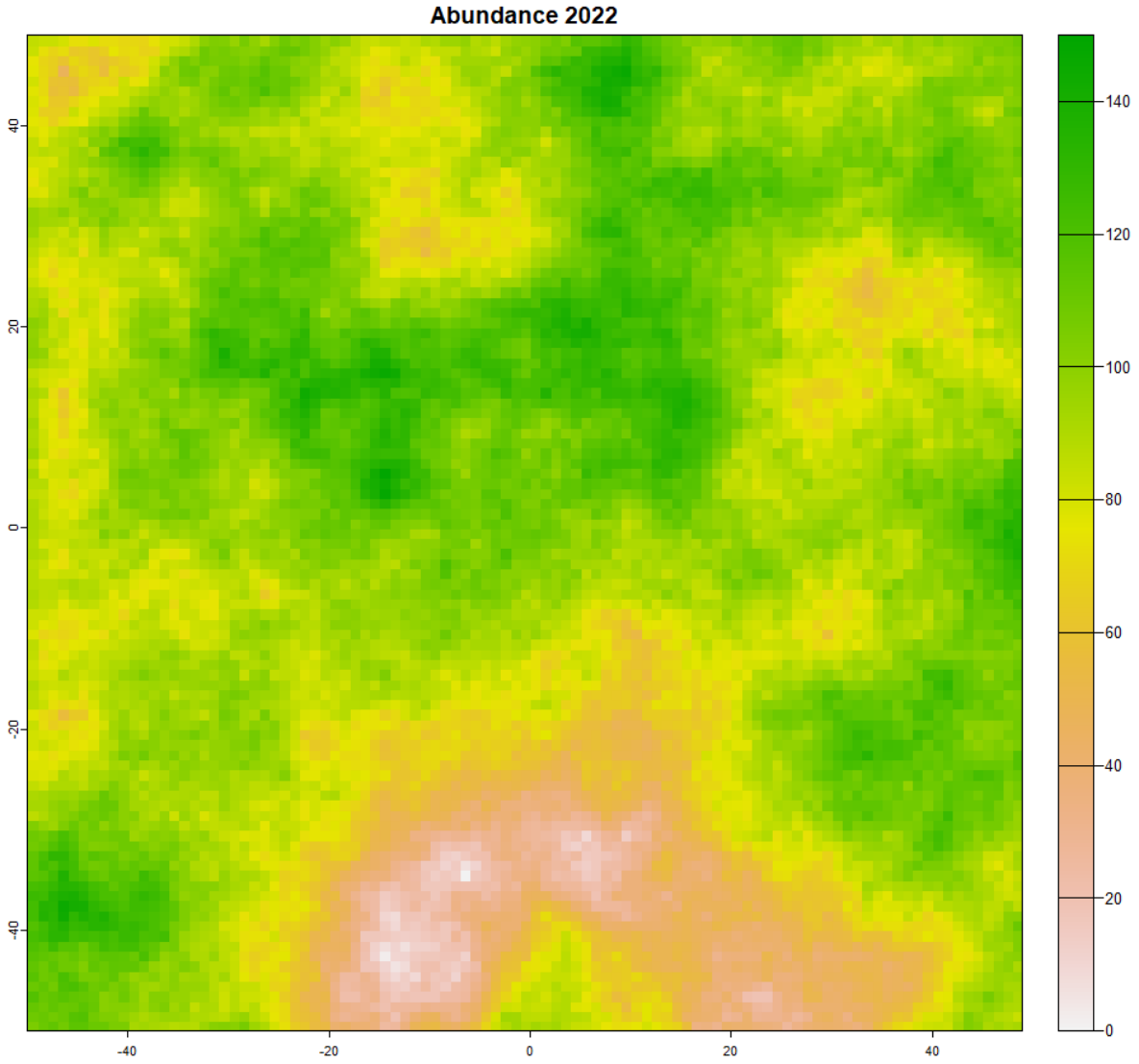
For those who choose OPTION 2, `setupProject()` should look similar to this code (i.e., replacing the GitHub user). For now, however, we will use `OPTION 1` to keep working with the module just created, as you likely did not have the chance to setup the GitHub repository (which can also be done from within `setupProject()`!). So please, don't run the `OPTION 2` code.:

```
runName <- "defaults"
out <- SpaDES.project::setupProject(
  runName = runName,
  paths = list(projectPath = "integratingSpaDESmodules",
               outputPath = file.path("outputs", runName)),
  modules = c("tati-micheletti/speciesAbundance@main"),
  times = list(start = 2013,
               end = 2022),
  updateRprofile = FALSE)

snippsim <- do.call(SpaDES.core::simInitAndSpades, out)
```

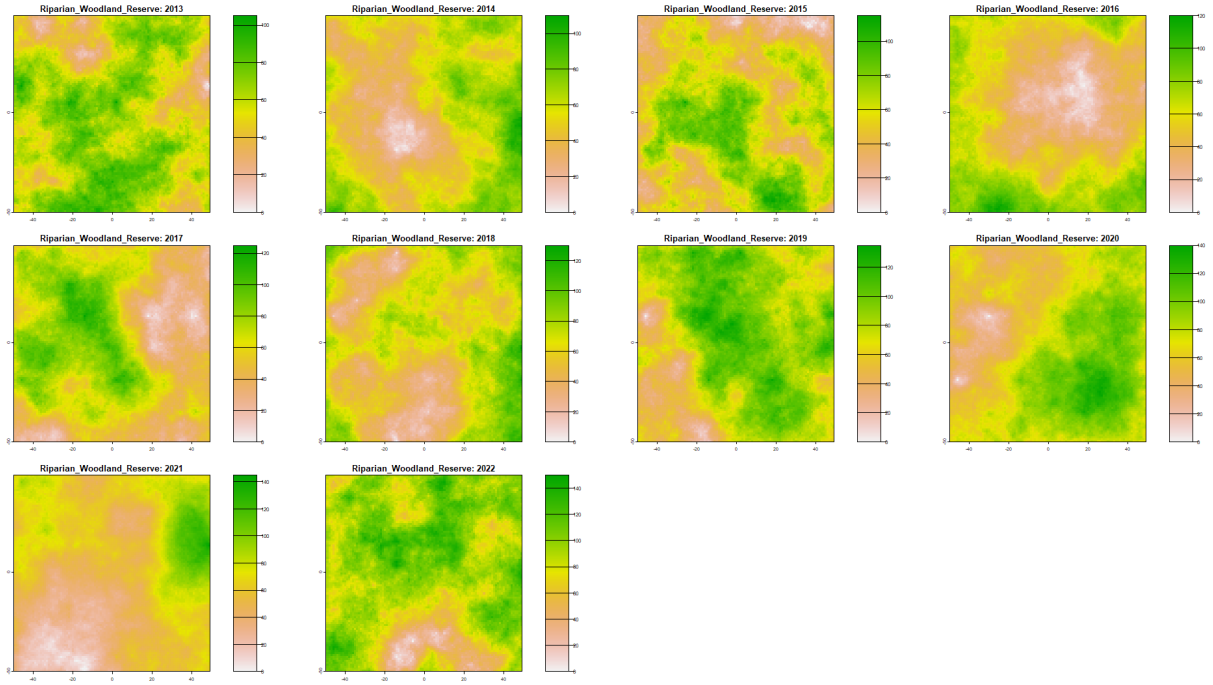
After the simulation has ran, the objects created in the simulation can be checked. First, we look at the `abundaRas`. To check it, just call it as an object in a list. This is the last year's abundance raster:

```
snippsim$abundaRas
terra::plot(snippsim$abundaRas, main = "Abundance 2022")
```



The next object of interest is the `allAbundaRas`. This is a raster stack of `abundaRas` for all years:

```
snippsim$allAbundaRas
terra::plot(snippsim$allAbundaRas)
```



The first thing we will change is the starting date of our model. We will start our project in 2018 instead of 2013. For that, we will change the `time` parameter and re-run our module. As always, restart your session to make sure the workflow can be ran from the start:

```
wd <- reproducible::checkPath("~/SpaDES_Projects", create = TRUE)
setwd(wd)
runName <- "lateStart"
out <- SpaDES.project::setupProject(
  runName = runName,
  paths = list(projectPath = "integratingSpaDESmodules",
               modulePath = file.path(dirname(wd), "SpaDES_Modules"),
               outputPath = file.path("outputs", runName)),
  modules = c("speciesAbundance"),
  times = list(start = 2018,
               end = 2022),
  updateRprofile = FALSE)

snippsim <- do.call(SpaDES.core::simInitAndSpades, out)
```

At last, we will change one of the parameters of the module using our project script. We will start plotting in 2018 instead of 2013. For that, we will add to `setupProject()` the `params` argument and pass to it a list of which module we want to change the parameter for, and for this, a list of which parameter and value it should take in. In this case, we change the parameter `.plotInitialTime`. We should then see that plotting will only start happening at year 2018:

```
wd <- reproducible::checkPath("~/SpaDES_Projects", create = TRUE)
setwd(wd)
runName <- "latePlotting"
out <- SpaDES.project::setupProject(
  runName = runName,
  paths = list(projectPath = "integratingSpaDESmodules",
               modulePath = file.path(dirname(wd), "SpaDES_Modules"),
               outputPath = file.path("outputs", runName)),
  modules = c("speciesAbundance"),
  params = list(speciesAbundance = list(.plotInitialTime = 2018)),
  times = list(start = 2013,
               end = 2022),
  updateRprofile = FALSE)

snippsim <- do.call(SpaDES.core::simInitAndSpades, out)
```

As we can start to grasp, `setupProject()` is a very powerful and flexible function. Exploring examples of how to set it may help understand its full potential. Now that we have explored how to set it up, we will add to our project two modules created by other developers, which we can integrate to ours to expand the possibilities in our analysis.

The first module we will integrate with our `speciesAbundance` is called `temperature`, and provides for the same region we have abundance data, temperature data. This way, could be able to add a second module (i.e., `speciesAbundTempLM`) that could create a model of `abundance ~ temperature` and potentially forecast species abundance if the `temperature` module provides a dataset that goes beyond species abundance's time frame.

It is natural that a basic knowledge of the modules to be added is needed. The module integration happens at the `expectedInputs` and `createdOutputs` level and, functionally, the only knowledge required to integrate these modules is the **name(s) of the object(s) of interest**, which can easily be found in the functional metadata part of the module, if not well described already in the module's manual.

So first, we will run both `speciesAbundance` and `temperature` modules. To do that using the defaults provided by the `temperature` module, the only thing needed in the `setupProject()` is the address to where the module is hosted. Note that we extend our `end time` of the simulation as the new module has a longer time series than out original model.

```
wd <- reproducible::checkPath("~/SpaDES_Projects", create = TRUE)
setwd(wd)
runName <- "defaultTemp"
out <- SpaDES.project::setupProject(
  runName = runName,
  paths = list(projectPath = "integratingSpaDESmodules",
               modulePath = file.path(dirname(wd), "SpaDES_Modules"),
               outputPath = file.path("outputs", runName)),
  modules = c("speciesAbundance",
              "tati-micheletti/temperature@main"),
  times = list(start = 2013,
               end = 2032),
  updateRprofile = FALSE)
```

```
snippsim <- do.call(SpaDES.core::simInitAndSpades, out)
```

In our example, there is no direct integration between the `speciesAbundance` and the `temperature` modules. These become integrated with the third module, `speciesAbundTempLM`, which uses inputs from both of these and creates a model that can be forecasted. Interesting to note on the new module is that the use of the option of saving functions in the `R/` folder. These are automatically parsed at the beginning of the simulation call (`simInitAndSpades()`). Moreover, we can see that as the `temperature` module has a longer time series of data, the simulation doesn't stop at the end of the time series from `speciesAbundace` module, but keeps going until the defined `end time`. This is due to our preemptive conditional scheduling of the events in the `speciesAbundance` module.

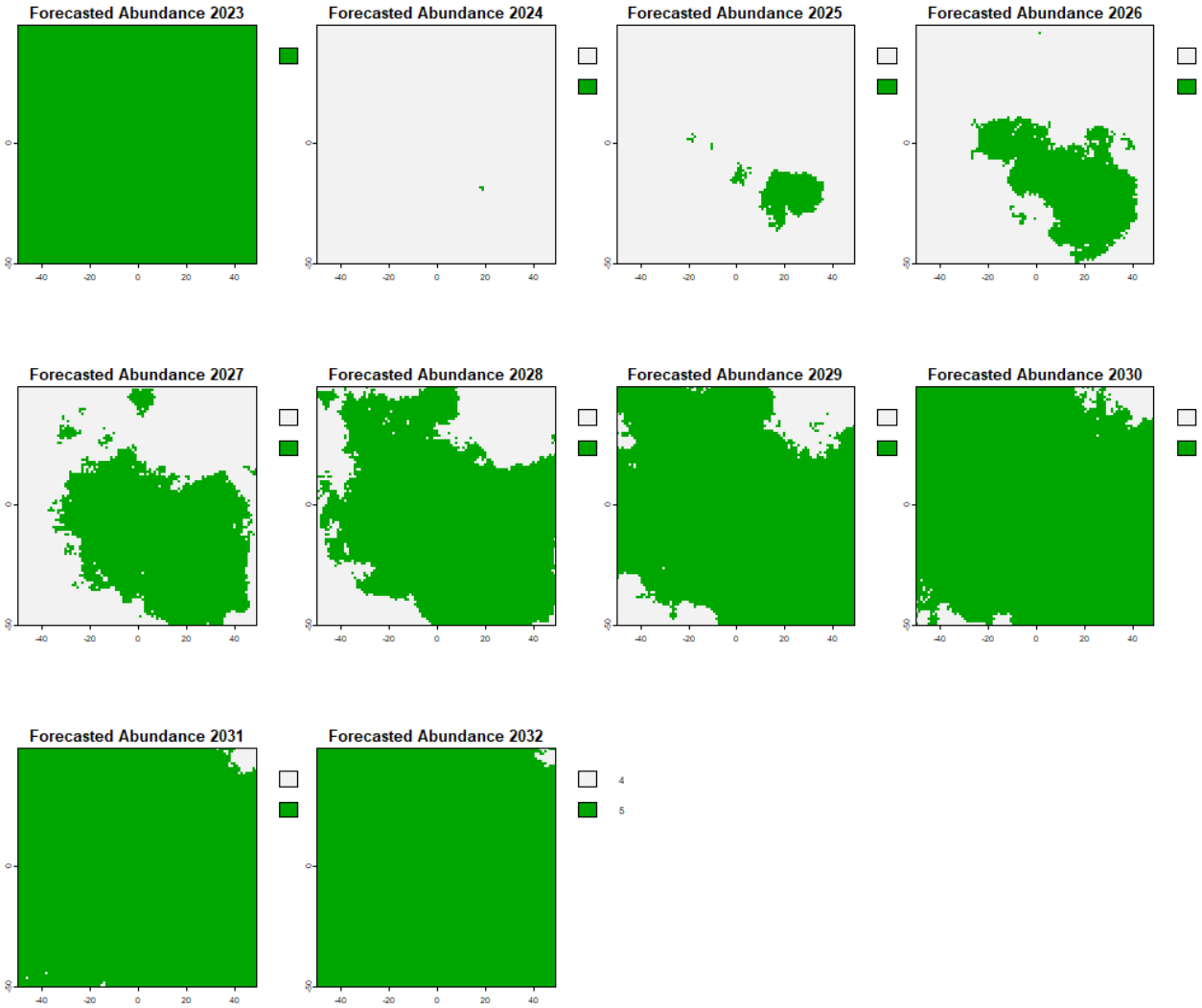
Now we will add a third module, which aims at fitting a linear model (LM) to help understand and forecast the relationship between species abundance and temperature. This module was carefully created, taking into account the objects created by `speciesAbundance` and `temperature`, more specifically, `abundaRas` and `tempRas`, respectively. These objects outputted by `speciesAbundance` and `temperature` are inputs in the `speciesAbundTempLM` module. The module was planned to identify the point in time when the data from the `speciesAbundance` module is no longer available, and forecast from that point on the abundance based on the provided temperature. The time for fitting of the LM is automatically detected, when all the data from the `speciesAbundance` is available. Until then, the module only stores the provided data in a `data.table`. The predictions are plotted and saved in the `outputs` folder.

```
wd <- reproducible::checkPath("~/SpaDES_Projects", create = TRUE)
setwd(wd)
runName <- "integratedDefault"
out <- SpaDES.project::setupProject(
  runName = runName,
  paths = list(projectPath = "integratingSpaDESmodules",
               modulePath = file.path(dirname(wd), "SpaDES_Modules"),
               outputPath = file.path("outputs", runName)),
  modules = c("speciesAbundance",
              "tati-micheletti/temperature@main",
              "tati-micheletti/speciesAbundTempLM@main"),
  times = list(start = 2013,
               end = 2032),
  loadOrder = c("speciesAbundance",
              "temperature",
              "speciesAbundTempLM"),
  updateRprofile = FALSE)

snippsim <- do.call(SpaDES.core::simInitAndSpades, out)
```

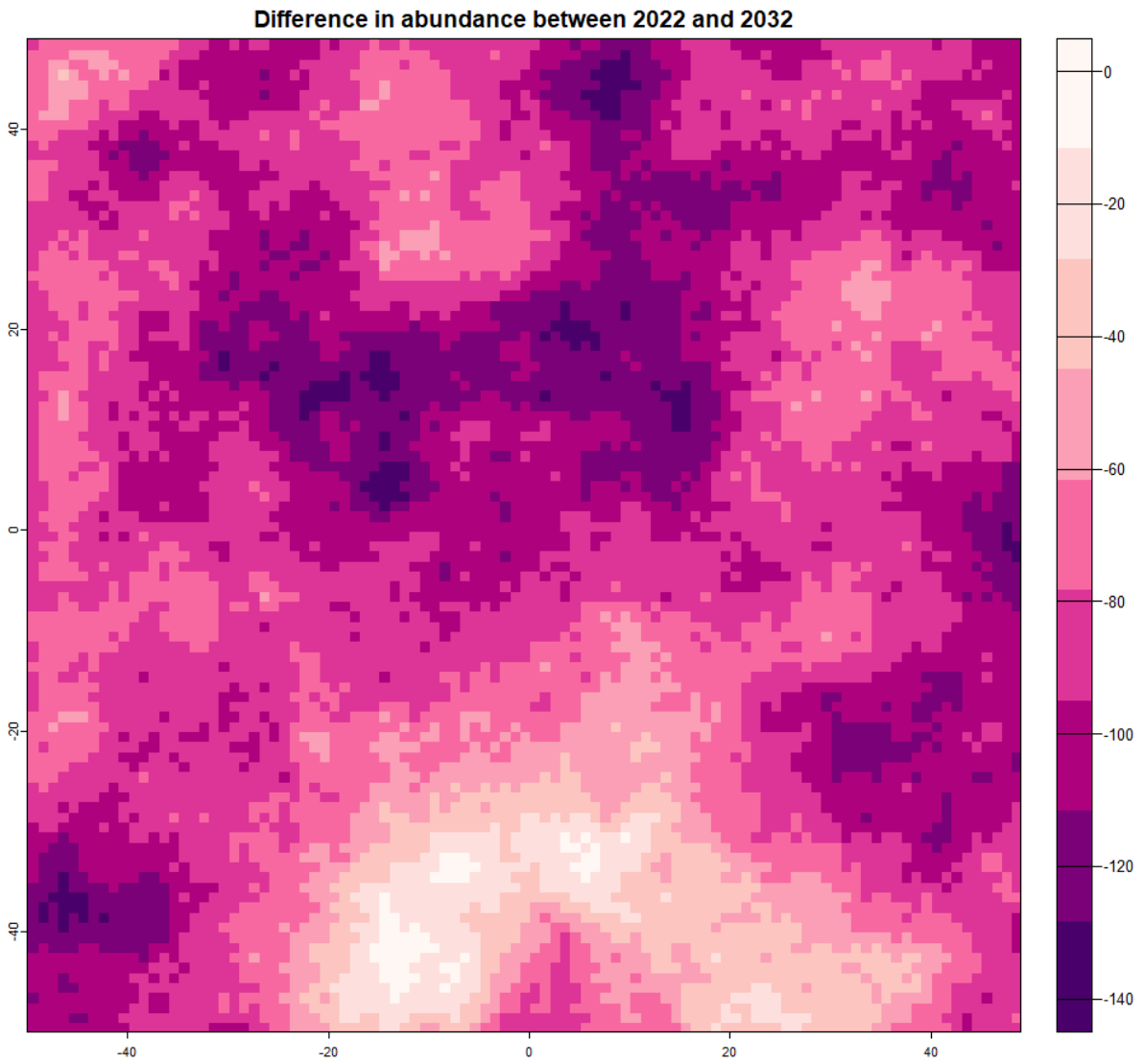
Now we can see some results. Accessing the forecasted abundances and the difference raster is also possible. This is done by calling the object name from the results list.

```
terra::plot(rast(snippsim$forecasts))
```



This is done by calling the object name from the results list.

```
terra::plot(snippsim$forecastedDifferences, col = c("#49006A", "#7A0177", "#AE017E",
                                                    "#DD3497", "#F768A1", "#FA9FB5",
                                                    "#FCC5C0", "#FDE0DD", "#FFF7F3"),
            main = "Difference in abundance between 2022 and 2032")
```



If you explore the last added module, you may notice that the event `abundanceForecasting` intentionally depicts the full range of actions done in the event, not wrapped in a function. Although we recommend all actions to be wrapped in functions to avoid cluttering the module, this is an alternative way to write the module and it is up to the module developer to decide on how to do it.

This is the end of the tutorial. I hope you can start working on your own SpaDES modules. If something is failing, below are some common mistakes when setting up modules. I also provide a list of further resources to improve even more your SpaDES skills!

Most common mistakes

The most common mistakes a user makes when starting working with SpaDES are:
- Forgetting to declare inputs and/or outputs
- Forgetting the correct usage of parameters: `P(sim)$paramName\` - Forgetting to schedule or scheduling events incorrectly

- Forgetting to assign objects that will be used by other events and/or modules to the `simList`
- Forgetting comas and parenthesis in the Module file

Further resources:

- **Recent Publications Supporting or Using SpaDES:**
1. Bauduin et al., 2019
2. Micheletti et al,. 2021
3. Barros et al., 2022
4. McIntire et al. 2022
5. Micheletti et al., 2023
6. Stewart and Micheletti et al., 2023
7. Raymundo et al., 2024

- **Issues in GitHub:** https://github.com/PredictiveEcology (and then to the specific package)

- **SpaDES weekly meeting:** meet.google.com/smq-bwnu-nhx (Weekly on Tuesday, 12 pm MST / 7pm UTC – for an invite, please send me an email at tati.micheletti@gmail.com. All participants are encouraged to add items to the agenda for discussion: https://docs.google.com/document/d/1Sq8YJoNTCu_kkjoweFf4W3IjokK0ZK6J7qq-CMRkR5c/edit#heading=h.y0z0my8yjzmi)/
- **SpaDES user group:** https://groups.google.com/g/spades-users/
- **Live SpaDES best practices (live document):** https://docs.google.com/document/d/19QmQ5sErqbXF_mgv3M50SnRQJBciFvCV_LuJDsj0qKA/edit?usp=sharing/
- **SpaDES.core vignettes:** https://spades-core.predictiveecology.org/articles/i-introduction.html/
- **All SpaDES.core functions:** https://spades-core.predictiveecology.org/reference/index.html/
- **Workshop Material:** https://spades-workshops.predictiveecology.org/index.html/
- **Published Material:** https://ceresbarros.github.io/SpaDES4Dummies/ (Complement to Barros et al., 2023)

Happy SpaDESing!