Creating and integrating simple models with SpaDES' help

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 distribution of the values and rasters for each year;
4. In the end, we ant to build a simple linear model to check if and how the abundance is changing though time.

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, with one raster of abundance values per year (`abundaRas`), and the simple linear model we created to check the change in abundance through time (`modAbund`). 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 (`.studyAreaName`), 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:

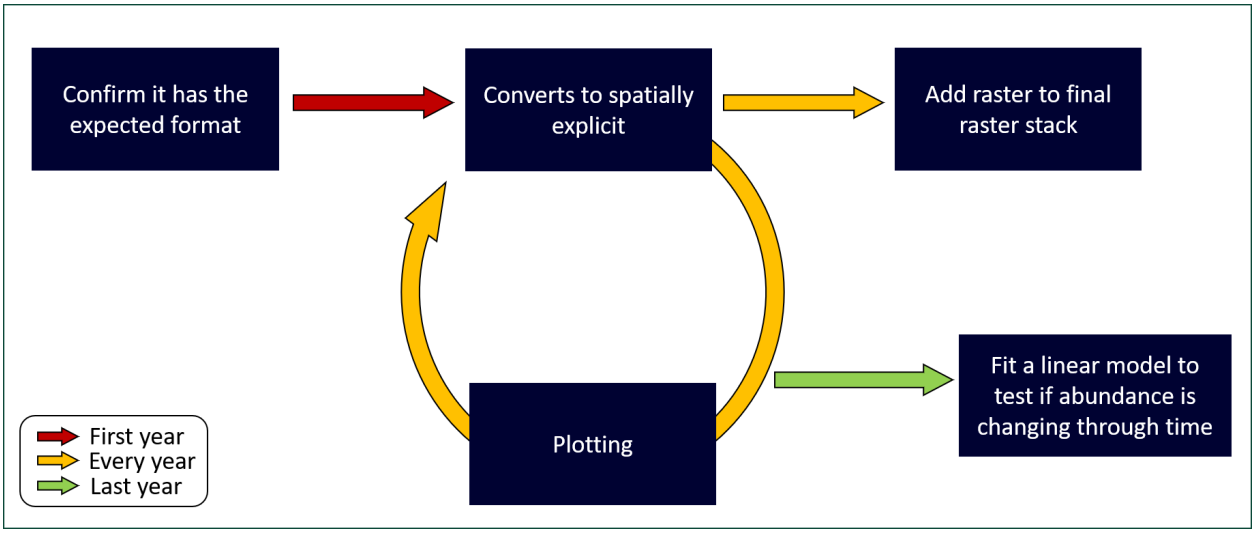

Figure 1: Module scheme presenting all important events detailing what happens to the original data.

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`, which 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 and load it.

```
if(!require("Require")){
  install.packages("Require")
}
library("Require")
```

After loading `Require`, we install and/or load SpaDES.

```
Require("SpaDES")
```

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`.

```
newModule(name = "speciesAbundance", path = "~/EFI")
```

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/Tati/Documents/EFI
* 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/Tati/Documents/EFI/speciesAbundance/speciesAbundance.R')
Using RStudio; may need to open manually e.g., with file.edit or file.show
file.edit('C:/Users/Tati/Documents/EFI/speciesAbundance/speciesAbundance.Rmd')
```

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 prefilled 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 infoirmation 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 `.`.
  Example: in our example case, we will use as parameters two of the pre-defined examples provided by the template: `.plotInitialTime` and `.studyAreaName`. These will mainly be used for the plotting event 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 three 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`. The third one is `modAbund`, a fitted model (of the `lm` class) of abundance through time. Outputting a model object could, for example, allow for posterior forecasts.

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(".plotInitialTime", "numeric", start(sim), start(sim), end(sim),
                  paste0("Describes the simulation time at which the first plot event",
                         "should occur.")),
  defineParameter(".studyAreaName", "character", "Riparian Woodland Reserve", NA, NA,
                  paste0("Human-readable name for the study area used - e.g., a hash ",
                         "of the study area obtained using ",
                         "`reproducible::studyAreaName()`"))
),
inputObjects = bindrows(
  expectsInput(objectName = "abund",
```

```
                objectClass = NA,
                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 = "") #<~~~~~~~~~~~~~~~ FILL IT UP WHEN DATA HAS BEEN UPLOADED!
  ),
  outputObjects = bindrows(
    createsOutput(objectName = "abundaRas",
                  objectClass = "spatRaster",
                  desc = paste0("A raster object of spatially explicit abundance data ",
                                "for a given year")),
    createsOutput(objectName = "allAbundaRas",
                  objectClass = "spatRaster", #<~~~~~~~ DOUBLE CHECK THE CLASS UPLOADED!
                  desc = "a raster stack of all `abundaRas`"),
    createsOutput(objectName = "modAbund", objectClass = "lm",
                  desc = paste0("A fitted model (of the `lm` class) of abundance through",
                                "time"))
  )
```

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.

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 `.studyAreaName` for the plot title. This plotting function will be dependent on the parameter `.plotInitialTime`, which will used to schedule the start of plotting events.

Finally, at the last year of the simulation, we will have an event named `abundanceThroughTime`, which will build a simple linear model to identify any trends in abundance though time.

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")
      sim <- scheduleEvent(sim, end(sim), "speciesAbundance", "abundanceThroughTime")
    },
    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 ----- ! #
    },
    abundanceThroughTime = {
      # ! ----- EDIT BELOW ----- ! #
      # do stuff for this event

      # e.g., call your custom functions/methods here
      # you can define your own methods below this `doEvent` function

      # schedule future event(s)
      # No need to schedule further events as this one happens at the end of the
      # simulations
```

```
      # ! ----- 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 original data from the first year up to the current year (histogram with distribution);

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

abundanceThroughTime:

(5) build a simple linear model using the abundance data (`abund`) to identify any trends in abundance though time.

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.

IMPORTANT Avoid using the same names for functions and events. It might get confusing to debug and might in some instances fail.

THE simList

THINGS TO PRESENT:

Changing start time of the module (start time) Change plotting time of the module (change parameter)