My first SpaDES module - from R to SpaDES

Ceres Barros, Eliot McIntire & Tati Micheletti

2019 October

Contents

Loops: what are they and why use them?

- fundamental programming units (most languages have them)
- used to repeat a set of instructions, without "duplicating" code
- often used in ecology to iterate something over "time"
  - usually, each 'time-step' depends on the previous one. Note in the examples below, how we start an `age` object at 1, and increment it at each step of `time`, or `t`. In both cases `time` and `t` are the loop *counters*, but only in the second case is the counter being effectively "used" inside the loop.

```
## Simple case:
age <- 1
for (time in 1:10) {
  age <- age + 1
}
```

Let's tear it apart and understand each individual part:

- Setting up (sometimes required), **initializing**

```
age <- 1
```

- deciding on **bounds** and **intermediates** or "step"
  - lower bound = 1 to upper bound = 10
  - here step is 1 (i.e., 1, 2, 3, 4, . . . 10)

```
for (time in 1:10)
```

- **content** (what is this loop going to do?)

```
age <-  age + 1
```

In both cases, we have to: - **initialise** a storage vector - define time **boundaries** - define the **step**, or incremental unit (in this case implicit) - define the **content** of the for-loop that is going to be iterated

From loops to events

- In R, we try to avoid loops because:
  - they are slow
  - they make code harder to share amongst users
- It's easy to think about "events" in ecology

In SpaDES, events are first *defined*, then *scheduled* to happen at a particular point in time:

Loops in R

```
## initializing
age <- 1

## boundaries
times = list(start = 1, end = 10)

## event definition (content), define what the event will do
aging <- age + 1

## event execution and scheduling - note the step definition
events <- {
  doEvent("aging")
  scheduleEvent("aging", when = now + 1)
}
```

As you can see, event execution and scheduling in SpaDES have the same fundamental components of a for-loop: **initialize**, **bounds**, **step**, **content**

Loops (AKA events) in SpaDES

```
## initialisation
age <- 1

## boundaries
times = list(start = 1, end = 10)

## event definition (content)
aging <- function(age) {
  age <- age + 1
}

## event execution and scheduling
events <- {
  doEvent("aging")
  scheduleEvent("aging", when = now + 1)
}
```

- This creates a queue – a list of events that need to occur

What would that sequence look like?

1. what is happening now
2. What are the next thing(s) in the queue

```
#     eventTime moduleName  eventType
#  1:         0       loop       init
#  2:         0       loop addOneYear
#  3:         1       loop addOneYear
#  4:         2       loop addOneYear
#  5:         3       loop addOneYear
#  6:         4       loop addOneYear
```

```
#  7:         5       loop addOneYear
#  8:         6       loop addOneYear
#  9:         7       loop addOneYear
# 10:         8       loop addOneYear
# 11:         9       loop addOneYear
# 12:        10       loop addOneYear
```

Why contain the iterated code in a function?

- iterated code remains isolated from execution and scheduling - cleaner, more organised
- easier to share
- easier to change and update - modular!

Exercise 1. Make your first module

Exercise 1.1. Creating a new module, understanding module scripts

1. Start by opening a new *.R* script, where you load the necessary libraries and that will serve as your "user-interface" or SpaDES "controller" script - I like to call it *global.R*
2. Define the directories
3. Create a new module in the module path

```
library(SpaDES)

## set/create directories
setPaths()    ## default temporary directories

setPaths(cachePath = "~/SpaDES_myModule/cache",
         inputPath = "~/SpaDES_myModule/inputs",
         modulePath = "~/SpaDES_myModule/modules",
         outputPath = "~/SpaDES_myModule/outputs")

## get paths
getPaths()

newModule("loop", path = getPaths()$modulePath)
```

- Two template scripts were created: an *.R* (will contain module code) and *.Rmd* (will contain module documentation and examples)
- Folders for data, R code and test code were also created, along with citation, license and README files

/!\ Atention: running `newModule` twice will overwrite any changes! /!

4. Now manually open the *.R* and *.Rmd* scripts

- First part of *loop.R* contains metadata, the second part contains event execution and scheduling functions, and the third part contains event functions (content).
- The *loop.Rmd* file is a template for documenting the module.

Exercise 1.2. Coding a module

We will first built the module "skeleton" and then define its parameters and eventual inputs/outpupts.

1. Skip to the `doEvent` function

- `doEvent` is the core of any SpaDES module
- It is where events are executed and scheduled

- When modules are created with `newModule`, `doEvent` is automatically suffixed with the module name (in this case "loop", so `doEvent.loop`) - /!\ this is **very** important /!\

2. Add event code and remove unnecessary events

- the template contains event "slots" for 5 different events: `init`, `plot`, `save`, `event1` and `event2`
- `init` is **mandatory** - /!\ never EVER remove it, or change its name /!\

```
doEvent.loop = function(sim, eventTime, eventType) {
  switch(
    eventType,
    init = {
      ## event content
      sim$age <- 1

      ## schedule event
      sim <- scheduleEvent(sim, start(sim), "loop", "addOneYear")
    },

    addOneYear = {
      ## event content:
      sim$age <- sim$age + 1

      ## schedule event
      sim <- scheduleEvent(sim, time(sim) + P(sim)$Step, "loop", "addOneYear")

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

Can you see where **initialize**, **bounds**, **step**, **content** are?

3. Define parameters

- In SpaDES, parameters can be "global" (of type `.<param_name.`) or module specific
- Parameters do not participate in the flow of information/data between modules
- Parameters can be changed by the user at the higher level (i.e. without changing the module code in the *.R* script)
- What do you think can be a parameter in our case?
- Parameters are defined in `definedModule`, using the `defineParameter` function
- This part of the module is the metadata, containing important information about the module
- It also indicates to other modules what to expect as its inputs and outputs
- Time boundaries do not need to be defined as parameters - they have their own special objects

```
defineModule(sim, list(
  name = "loop",
  description = NA, #"insert module description here",
  keywords = NA, # c("insert key words here"),
  authors = person("First", "Last", email = "first.last@example.com", role = c("aut", "cre")),
  childModules = character(0),
  version = list(SpaDES.core = "0.2.2.9006", loop = "0.0.1"),
```

```
  spatialExtent = raster::extent(rep(NA_real_, 4)),
  timeframe = as.POSIXlt(c(NA, NA)),
  timeunit = "year",
  citation = list("citation.bib"),
  documentation = list("README.txt", "loop.Rmd"),
  reqdPkgs = list(),
  parameters = rbind(
    #defineParameter("paramName", "paramClass", value, min, max, "parameter description"),
    defineParameter(".plotInitialTime", "numeric", NA, NA, NA, "This describes the simulation time at wh
    defineParameter(".plotInterval", "numeric", NA, NA, NA, "This describes the simulation time interval
    defineParameter(".saveInitialTime", "numeric", NA, NA, NA, "This describes the simulation time at wh
    defineParameter(".saveInterval", "numeric", NA, NA, NA, "This describes the simulation time interval
    defineParameter(".useCache", "logical", FALSE, NA, NA, "Should this entire module be run with cachin
  )
))
```

4. Define inputs/outputs

- Inputs and outputs, unlike parameters, are objects that establish links between modules, and between the user and modules
- They are **always** contained in the `simList` object
- A good way of thinking about what input and output *objects* are is: `sim$outputs <- sim$inputs`
- do we have any inputs? What about outputs?
- Input and output objects are also defined in `defineModule` using the `expectsInput` and `createsOutput` functions

```
 inputObjects = bind_rows(
    #expectsInput("objectName", "objectClass", "input object description", sourceURL, ...),
    expectsInput(objectName = NA, objectClass = NA, desc = NA, sourceURL = NA)
 )

  outputObjects = bind_rows(
    #createsOutput("objectName", "objectClass", "output object description", ...),
    createsOutput(objectName = NA, objectClass = NA, desc = NA)
  )
```

5. Complete metadata, and define the parameter, expectedInputs and createdOutputs

- Don't forget to complete the remaining metadata like authorship, essential keywords, time units, etc.
- /!\ time units need to be correctly defined, as they will affect how modules are linked /!\
- /!\ remember to declare package dependecies /!\
- When you are done, don't forget to save the *loop.R* file!

```
defineModule(sim, list(
  name = "loop",
  description = "For-loop in SpaDES",
  keywords = c("loops", "age", "simple"),
  authors = person("John", "Doe", email = "john.doe@example.com", role = c("aut", "cre")),
  childModules = character(0),
  version = list(SpaDES.core = "0.1.1.9005", loop = "0.0.1"),
  spatialExtent = raster::extent(rep(NA_real_, 4)),
  timeframe = as.POSIXlt(c(NA, NA)),
  timeunit = "year",
  citation = list("citation.bib"),
```

```
  documentation = list("README.txt", "loop.Rmd"),
  reqdPkgs = list(),
  parameters = rbind(
    defineParameter(name = "Step", class = "numeric", default = 1, min = NA, max = NA, desc = "Time step
  ),
  inputObjects = bind_rows(
    #expectsInput("objectName", "objectClass", "input object description", sourceURL, ...),
    expectsInput(objectName = NA, objectClass = NA, desc = NA, sourceURL = NA)
  ),
  outputObjects = bind_rows(
    #createsOutput("objectName", "objectClass", "output object description", ...),
    createsOutput(objectName = "age", objectClass = "integer", desc = "Age vector")
  )
))
```

Exercise 1.3. Run simulations, check the event queue and module diagrams

Now let's give our *loop.Rmd* an example - let's set up the "simulation" run. 1. Check the event queue before and after running `spades` 2. Produce module diagrams before running `spades` 3. Run the "simulation" 4. Compare with outputs produced by the "normal" loop

```
## Simulation setup
paths <- getPaths()
modules <- list("loop")
times <- list(start = 1, end = 10)
parameters <- list(loop = list(Step = 1L))

## SpaDES Events
mySim <- simInit(paths = paths, modules = modules,
                 times = times, params = parameters)   ## remove the "L" from Step and see what happens
events(mySim)   ## shows scheduled events

mySimOut <- spades(mySim, debug = TRUE)   ## execute events
events(mySimOut)      ##
completed(mySimOut)   ## shows completed events

mySimOut$age

## Loop version
age <- 1
for (time in 1:10) {
  age <- age + 1
}

## Compare outputs
mySimOut$age
age
```

Note that mySimOut is a *pointer* to the updated/changed `mySim` not a true new `simList` object

Exercise 1.4. Make it even more `SpaDESy`

Notice that below the `doEvent.loop` function there are templates for other funcitons that can be used inside the events. Keeping the code inside these functions increases modularity and flexibility, as functions are self-contained.

1. Make separate functions to be used in the `init` and the `addOneYear` events.

```
### Initialisation function
loopInit <- function(sim) {
  sim$age <- 1
  return(invisible(sim))
}

### Aging event function
aging <- function(age = sim$age) {
  age <- age + 1
  return(age)
}
```

NOTE: We present above two different ways of specifying a function. One always passed the `sim` object to the function and return the `sim` oject modified. The second returns the results of a function to the `sim` object as a new object "in" it.

2. Now you'll need to adapt the code inside `doEvent.loop` so that the appropriate functions are called inside their respective events

```
doEvent.loop = function(sim, eventTime, eventType) {
  switch(
    eventType,
    init = {
      ## event content
      # sim$age <- 1
      ## OR
      sim <- loopInit(sim)

      ## schedule event
      sim <- scheduleEvent(sim, start(sim), "loop", "addOneYear")
    },
    addOneYear = {
      ## event content:
      # sim$age <- sim$age + 1
      ## OR:
      sim$age <- aging(age = sim$age)

      ## schedule event
      sim <- scheduleEvent(sim, time(sim) + P(sim)$Step, "loop", "addOneYear")
    },
    warning(paste("Undefined event type: '", current(sim)[1, "eventType", with = FALSE],
                  "' in module '", current(sim)[1, "moduleName", with = FALSE], "'", sep = ""))
  )
  return(invisible(sim))
}
```