Introduction to SpaDES

Alex M. Chubaty and Eliot J. B. McIntire

January 08 2015

Contents

Abstract: Implement a variety of simulation models, with a focus on spatially explicit raster models and agent based models. The core simulation components are built upon a discrete event simulation framework that facilitates modularity, and enables the user to include additional functionality by running user-built simulation modules. Included are numerous tools to visualize raster and other maps.

Website: https://github.com/achubaty/SpaDES

1 Introduction to SpaDES

1.1 Objectives and motivations

Building spatial simulation models often involves reusing various components, often having to reimplement similar fuctionality in multiple simulation frameworks (*i.e*, in different programming languages). When various

components of a simulaiton model become fragmented across multiple platforms, it becomes increasingly difficult to link these various components, and often solutions for this problem are idiosyncratic and specfic to the model being implemented.

`SpaDES` is a generic simulation platform that can be used to create new model components quickly. It also provides a framework to link with existing simulation models, so that an already well described and mature model, *i.e.*, Landis-II, can be used with *de novo* components. Alternatively one could use several *de novo* models and several existing models in combination. This approach requires a platform that allows for modular reuse of model components (herein called "modules") as hypotheses that can be evaluated and tested in various ways.

When beginning development of this package, we sought a general simulation platform at least the following characteristics:

1. Allow rapid building of models of a wide diversity of types (IBMs, raster models, differential equation models, etc.);
2. Run faster and more memory efficiently than current systems for doing similar things (NetLogo, SELES, Repast, etc.);
3. Use a platform that already has strong data analysis and manipulation capacities;
4. Be open source, but also make it as easy as possible for many people to easily contribute modules and code;
5. Be easy to use for a large number of scientists who aren't formally trained as computer programmers;
6. Should be built around modularity so that models can be seen as modules that are easily replaceable, not just "in theory" replaceable;
7. Allow tight coupling between data and model simulations so that calibration is not actually something that one has to redesign every time there is a new data set.

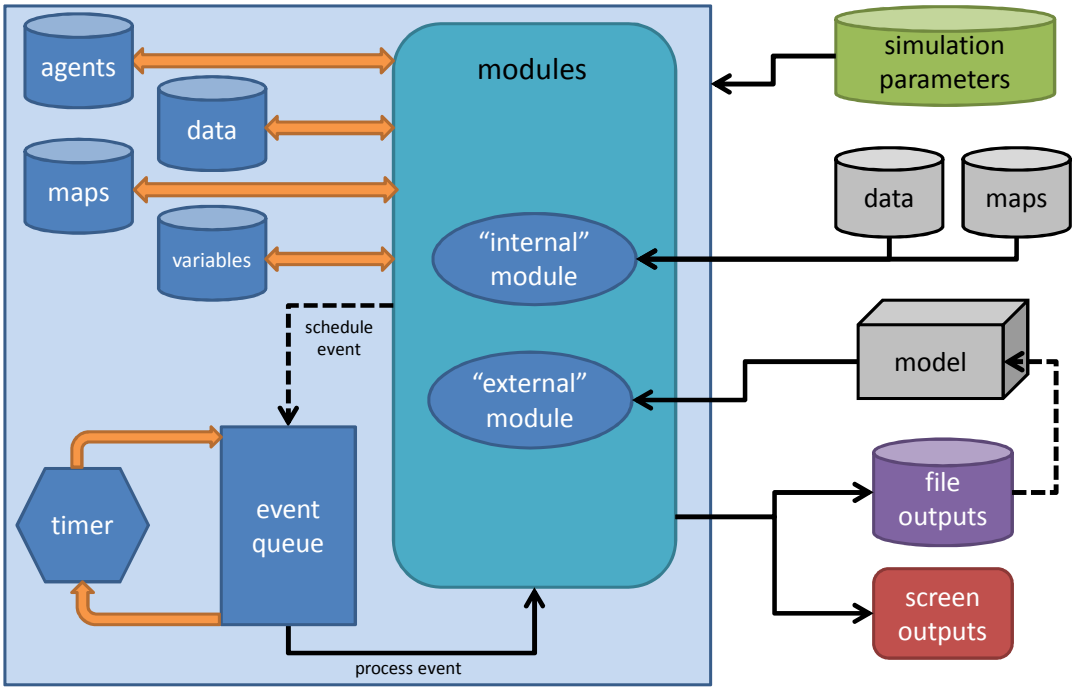
We selected `R` as the system within which to build `SpaDES`. `R` is currently the *lingua franca* for scientific data analysis. This means that anything developed in `SpaDES` is simply `R` code and can be easily shared with journals and the scientific community. We can likewise leverage `R`'s strengths as a data platform, its capabilities to run external code such as C and Python, call external software such as databases, its excellent visualization and graphics, and its abilities for high performance computing. We don't have to implement all of these from scratch ourselves!

1.2 Discrete event simulation and `SpaDES`

Discrete event simulation (DES) as implement here is "event driven", meaning that an activity changes the state of the system at particluar times (called events). This approach assumes that state of the system only changes due to events, therefore there is no change between events. A particular activity may have several events associated with it. Future events are scheduled in an event queue, and then processed in chronological order. Because the system state doesn't change between events, we do not need to 'run the clock' in fixed increments each timestep. Rather, time advances to the time of the next event in the queue.

'Time' is the core concept linking various simulation components via the event queue. Activities schedule events (which change the system according to their programmed rules) and do not need to know about each other. This allows for modularity of simulation components. Thus, complex simulations involving multiple processes (activities) can be built fairly easily, provided these processes are modelled using a common DES framework.

`SpaDES` provides such a framework, facilitating interaction between multiple processes (built as "modules") that don't interact with one another directly, but are scheduled in the event queue and carry out operations on shared data objects in the global simulation environment. This package provides tools for building modules natively in `R` that can be reused. Additionally, because of the flexibility`R` provides for interacting with other programming languages and external data sources, modules can also be built using external tools and integrated with `SpaDES`.



1.3 SpaDES demos and sample modules

The static nature of PDFs does not allow us to really show off the simulation visualization components of this package, so we invite you to check out the included demos, to run the sample simulation provided in this vignette, and to view the source code for the sample modules included in this package.

1.3.1 Demos

```
library("SpaDES")

# demo: randomLandscapes, fireSpread, caribouMovement
demo("spades-simulation", package="SpaDES")
```

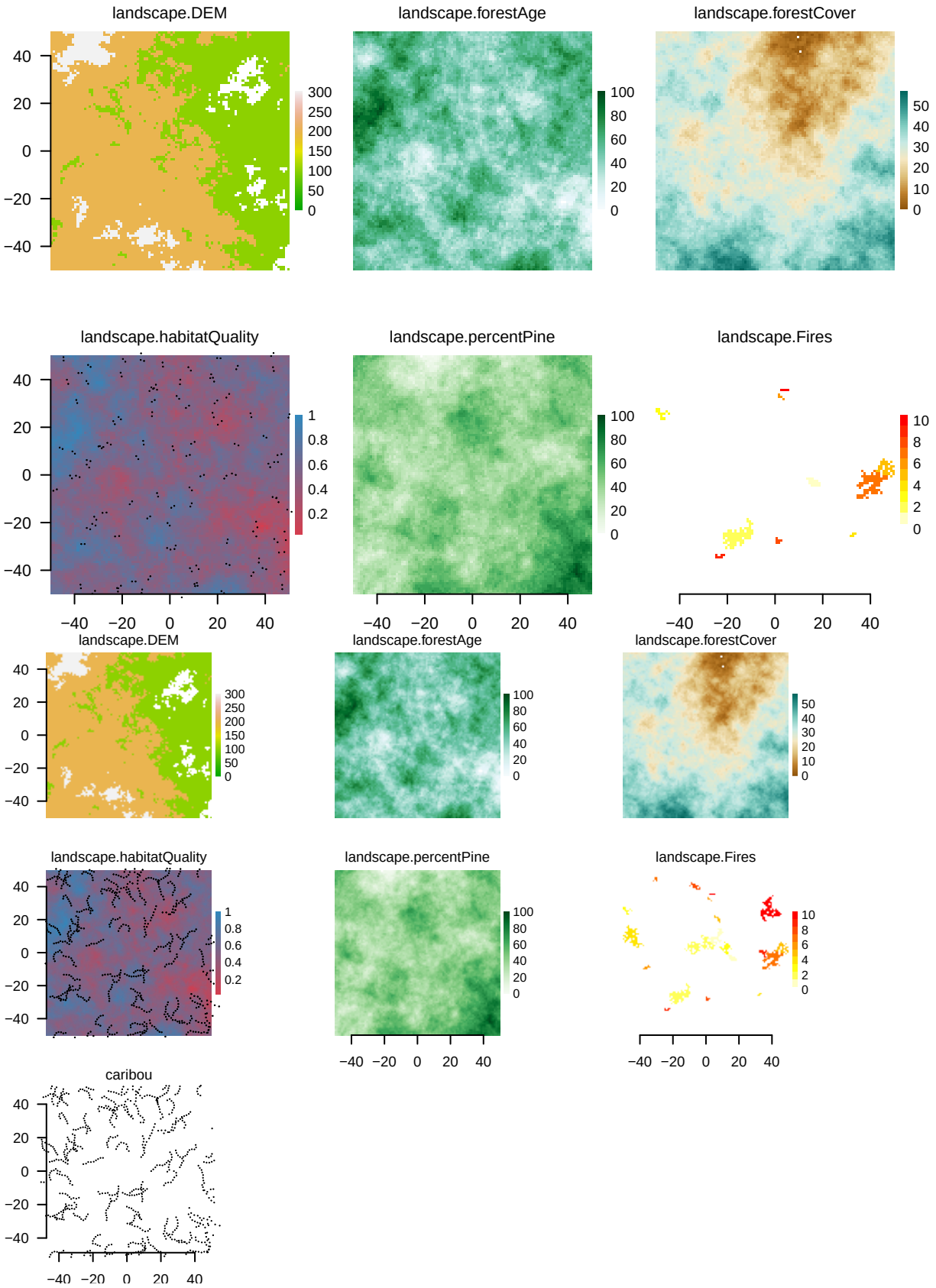
1.3.2 Sample model

```
library("SpaDES")

outputPath=file.path(tmpDir(), "simOutputs")
```

```
times <- list(start=0, stop=10.2)
parameters <- list(.globals=list(.stackName="landscape", .outputPath=outputPath,
                                 burnStats="nPixelsBurned"),
                   .progress=list(NA),
                   randomLandscapes=list(nx=1e2, ny=1e2, inRAM=TRUE),
                   fireSpread=list(nFires= 1e1, spreadprob=0.225, its=1e6,
                                   persistprob=0, returnInterval=10, startTime=0,
                                   .plotInitialTime=0.1, .plotInterval=10),
                   caribouMovement=list(N=1e2, moveInterval=1,
                                        .plotInitialTime=1.01, .plotInterval=1)
                   )
modules <- list("randomLandscapes", "fireSpread", "caribouMovement")
path <- system.file("sampleModules", package="SpaDES")

mySim <- simInit(times=times, params=parameters, modules=modules, path=path)
```

```
##
## Global parameters .outputPath are not used in any module.
```

```
#dev(4)
spades(mySim)
```


landscape.DEM
landscape.forestAge
landscape.forestCover
landscape.habitatQuality
landscape.percentPine
landscape.Fires
landscape.DEM
landscape.forestAge
landscape.forestCover
landscape.habitatQuality
landscape.percentPine
landscape.Fires
caribou

2 Using SpaDES to build simulations

Requirements: This packages makes heavy use of the `raster` and `sp` packages, so familiarity with these packages, their classes and methods is recommended. Plotting features are built using the `grid` package.

2.1 Setting up a simulation:

As you can see in the sample simulation code provided above, setting up and running a simulation in `SpaDES` is straightforward using existing modules. You need to specify somethings about the simulation environment including 1) all parameter values passed to the simulation, 2) which modules to use for the sim, and 3) any global data objects that should be used to store the simulation state. Each of these are passed as named lists to the simulation object upon initilization.

2.1.1 Initializing a simulation:

The details of each simulation are stored in an S4 `simList` object, including the simulation parameters and modules used, as well as storing the current state of the simulation and the future event queue. A list of all completed events is also stored, which can provide useful debugging information.

A new simulation is initialized using the `simInit` function, which does all the work of creating the `simList` object for your simulation, setting all the slot values appropriately. Furthermore, this function tries to provide additional feedback to the user regarding parameters that may be improperly specified.

You can inspect the contents of a `simList` object as you would any other `R` object (*e.g.*, `show(mySim)`).

2.1.2 Running a simulation:

Once a simulation is properly initialized it is executed using the `SpaDES` function. By default, a progress bar is displayed in the console (this can be customized), and any specified files are loaded (via including a fileList dataframe, see examples). Debugging mode (setting `spades(mySim, debug=TRUE)`) prints the contents of the `simList` object after the completion of every event during simulation.

2.2 SpaDES modules

`SpaDES` modules are event-driven, meaning that different actions are performed on data objects based on the order of scheduled events. A module describes the processes or activities that drive simulation state changes. Each activity consists of a collection of events which are scheduled depending on the rules of the simulation. Each event may evaluate or modify a simulation data object, or perform other operations such as saving and loading data objects or plotting.

The power of `SpaDES` is in modularity and the ease with which existing modules can be modified and new modules created. This vignette will highlight general use of the package and its features using the sample modules provided. Creating and customizing modules is a whole topic unto itself, and for that reason we have created a separate `modules` vignette with more details on module development.

3 Simulation and data

Historically, simulation models were built separately from the analysis of input data (*e.g.*, via regression) and outputs of data (*e.g.*, graphically, statistically). On the input data side, this effectively broke the linkage between data (*e.g.*, from field or satellites) and the simulation. This has the undesired effect of creating the appearance of reduced uncertainty in simulation model predictions, by breaking correlations between

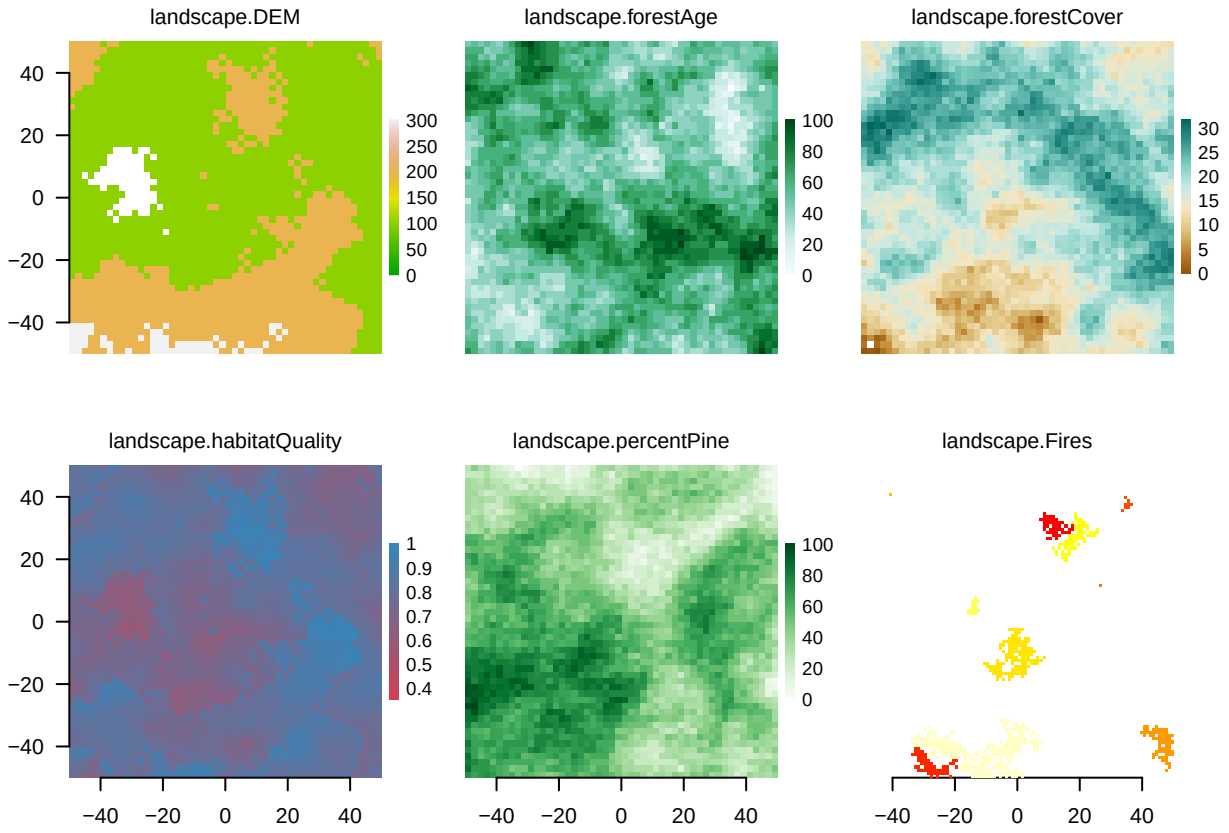
parameter estimates (that invariably occur in analyses of real data). Conversely, on the data output side, numerous tools, such as optimization (*e.g.*, pattern oriented modeling) or statistical analyses could not directly interact with the simulation model, unless a specific extension was built for that purpose. In R, those tools already exist and are robust. Thus, validation, calibration, and verification of simulation models can become rolled into the simulation model itself. This enhances transparency and reproducibility, both desired properties for scientific studies.

Bringing data into Ris easy, and can be done using any of the built in data import tools. To facilitate this, we have provided additional functionality to easily load maps or data from files via the load module. To automatically import a list of files, simply provide it as a parameter named `.loadFileList` when initializing the simulation. See `?loadFiles` and the modules vignette for more information on the load module.

```
### Example: loading habitat maps

# use all built-in maps from the SpaDES package
pathToMaps <- file.path(find.package("SpaDES", quiet=FALSE), "maps")
fileList <- data.frame(files=dir(pathToMaps, full.names=TRUE, pattern= "tif"),
                       functions="rasterToMemory", packages="SpaDES",
                       stringsAsFactors=FALSE)

# this list can be passed to simInit() as an entry in the parameter list
mySim <- simInit(times=list(start=0.0, stop=10),
                 params=list(
                   .loadFileList=fileList,
                   .progress=list(NA),
                   .globals=list(.stackName="landscape", burnStats="nPixelsBurned"),
                   #.globals=list(burnStats="nPixelsBurned"),
                   fireSpread=list(nFires=1e1, spreadprob=0.225, persistprob=0,
                                   its=1e6,returnInterval=10, startTime=0.1,
                                   .plotInitialTime = 0, .plotInterval=10)
                 ),
                 modules=list("fireSpread"),
                 path=system.file("sampleModules", package="SpaDES"))
```

```
## DEM read from /home/achubaty/R-dev/SpaDES/maps/DEM.tif using rasterToMemory
## forestAge read from /home/achubaty/R-dev/SpaDES/maps/forestAge.tif using rasterToMemory
## forestCover read from /home/achubaty/R-dev/SpaDES/maps/forestCover.tif using rasterToMemory
## habitatQuality read from /home/achubaty/R-dev/SpaDES/maps/habitatQuality.tif using rasterToMemory
## percentPine read from /home/achubaty/R-dev/SpaDES/maps/percentPine.tif using rasterToMemory
## individual files have been stacked into landscape and are not individual RasterLayers
```

```
spades(mySim)
```

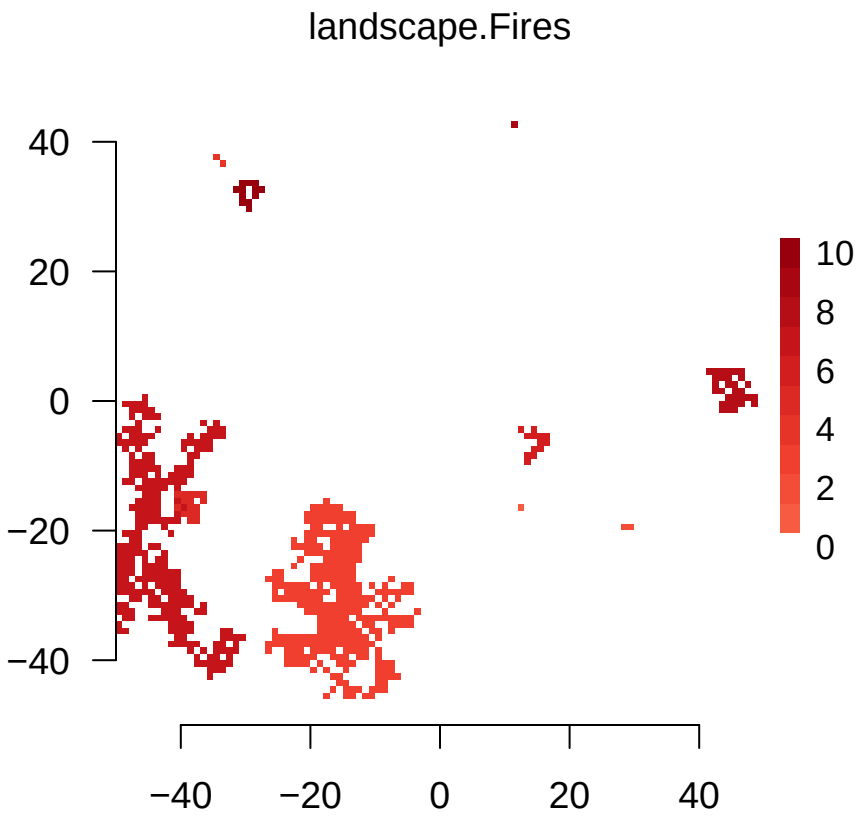


4 Modelling spread processes

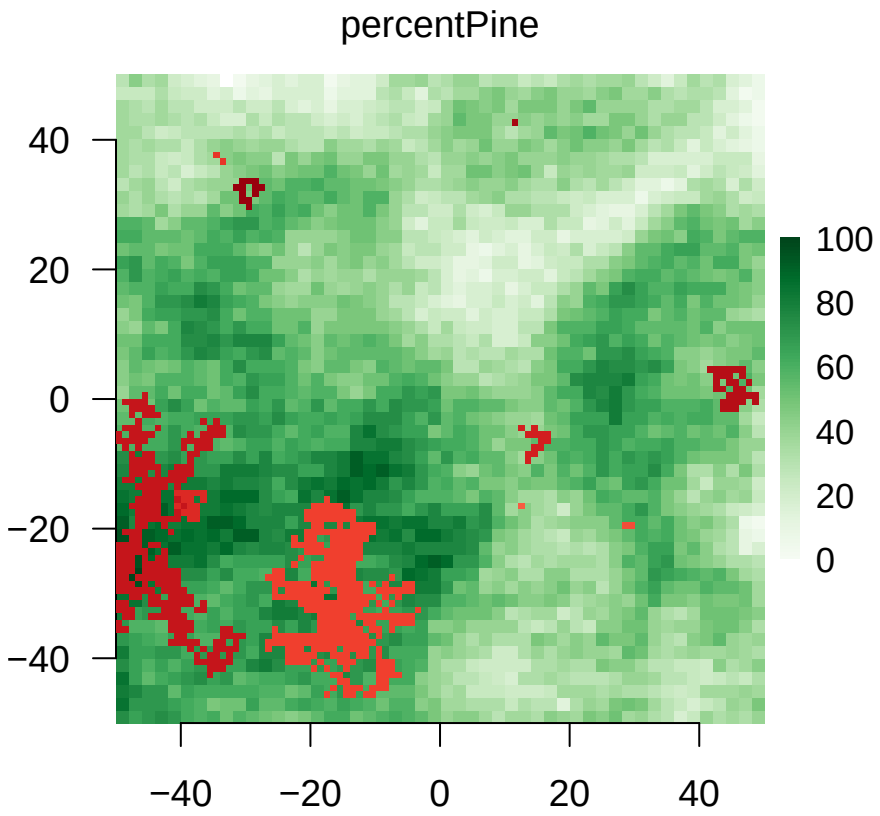
4.1 A simple fire model

Using the `spread` function, we can simulate fires, and subsequent changes to the various map layers. Here, `spreadProb` can be a single probability or a raster map where each pixel has a probability. In the example below, each cell's probability is taken from the Percent Pine map layer.

```
library(RColorBrewer)
nFires <- 10
landscape[["Fires"]] <-
  spread(landscape[["percentPine"]],
         loci=as.integer(sample(1:ncell(landscape), nFires)),
         spreadProb=landscape[["percentPine"]]/(maxValue(landscape[["percentPine"]])*5)+0.1,
         persistance=0,
         mask=NULL,
         maxSize=1e8,
         directions=8,
         iterations=1e6,
         plot.it=FALSE,
         mapID=TRUE)

setColors(landscape$Fires)<-brewer.pal(8,"Reds")[5:8]

Plot(landscape[["Fires"]], new=TRUE)
```




```
# Show the burning more strongly over abundant pine
percentPine<-landscape$percentPine
Plot(percentPine, new=TRUE)
#Plot(landscape[["Fires"]], new=FALSE)
Plot(landscape[["Fires"]], addTo="percentPine", legend=FALSE, title=FALSE)
```



We can see that the fires tend to be in the Pines because we made it that way, using an arbitrary weighting with pine abundance:

```
# Show the burning more strongly over abundant pine
fire <- reclassify(landscape[["Fires"]], rcl=cbind(0:1, c(0,ncell(landscape)), 0:1))
pine <- reclassify(landscape[["percentPine"]], rcl=cbind(0:9*10, 1:10*10, 0:9))
PineByFire <- crosstab(fire, pine, long=TRUE)
colnames(PineByFire) <- c("fire", "pine", "freq")
PineByFire$pine <- as.numeric(as.character(PineByFire$pine))
summary(glm(freq ~ fire*pine, data=PineByFire, family="poisson"))
```

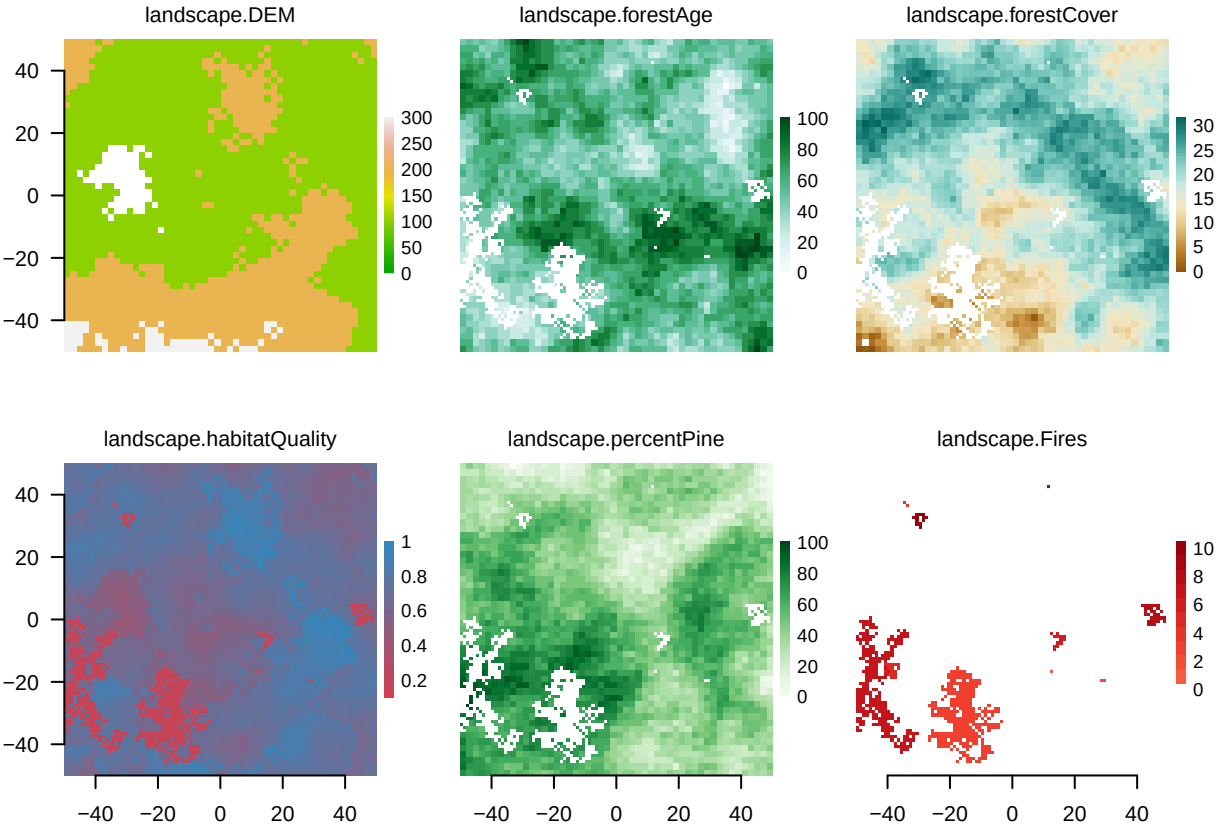
```
##
## Call:
## glm(formula = freq ~ fire * pine, family = "poisson", data = PineByFire)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -38.089  -11.784    1.318    7.592   28.863
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  6.915801   0.018810 367.664  < 2e-16 ***
## fire1       -3.680614   0.146052 -25.201  < 2e-16 ***
## pine        -0.017110   0.003604  -4.747 2.06e-06 ***
## fire1:pine   0.220772   0.020860  10.583  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 11802.4  on 16  degrees of freedom
## Residual deviance:  5433.5  on 13  degrees of freedom
## AIC: 5564.8
##
## Number of Fisher Scoring iterations: 5
```

Sure enough, there are more fires as the abundance of pine goes up, as seen by the positive interaction term (the negative `fire1` term means that there are more pixels without fires than with fires).

Impact some of the forest

```
landscape[["forestAge"]][landscape[["Fires"]]>0] <- 0
landscape[["forestCover"]][landscape[["Fires"]]>0] <- 0
landscape[["habitatQuality"]][landscape[["Fires"]]>0] <- 0.1
landscape[["percentPine"]][landscape[["Fires"]]>0] <- 0
Plot(landscape, new=TRUE)
```



5 Agent based modelling

A primary goal of developing `SpaDES` was to facilitate the development of agent-based models (ABMs), also known as individual-based models (IBMs).

5.1 Point agents

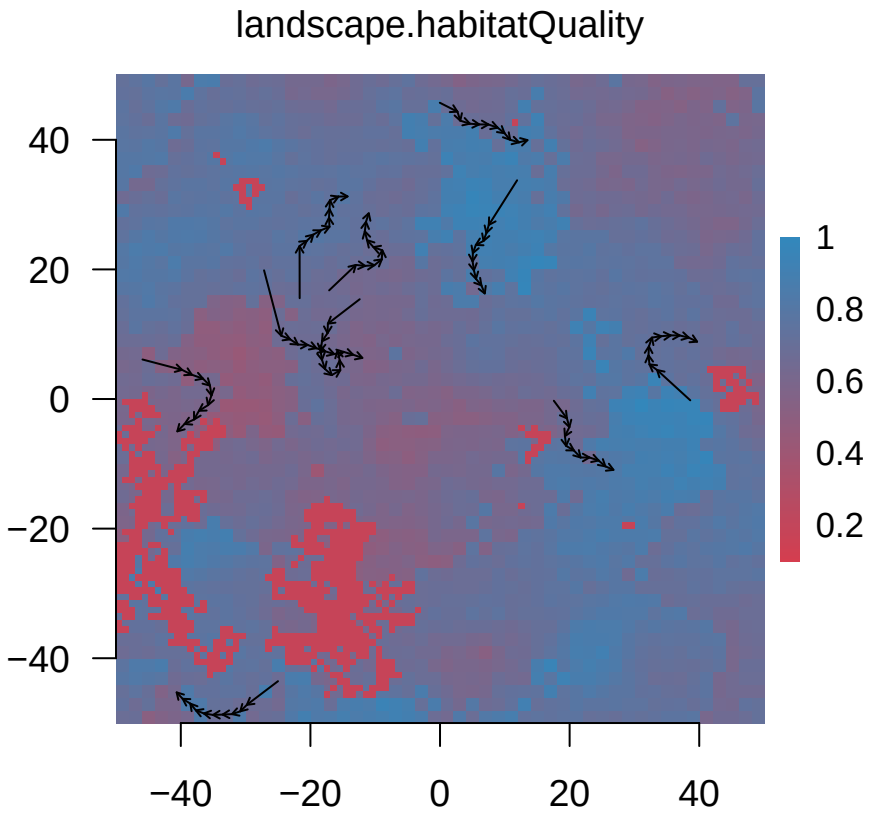
As ecologists, we are usually concerned with modelling individuals (agents) in time and space, and whose spatial location (position) can be represented as a single point on a map. These types of agents can easily be represented most simply by a single set of coordinates indicating their current position, and can simulated using a `SpatialPoints` object. Additionally, a `SpatialPointsDataFrame` can be used, which provides storage of additional information beyond agents' coordinates as needed.

To model mobile point agents, *e.g.*, animals (as opposed to non-mobile agents such as plants), use a `SpatialPointsDataFrame` containing additional columns for storing agents' previous `n` positions.

```
N <- 10 # number of agents

# caribou data vectors
IDs <- letters[1:N]
sex <- sample(c("female", "male"), N, replace=TRUE)
age <- round(rnorm(N, mean=8, sd=3))
x1 <- runif(N, -50, 50) # previous X location
y1 <- runif(N, -50, 50) # previous Y location

# caribou (current) coordinates
x0 <- rnorm(N, x1, 5)
y0 <- rnorm(N, y1, 5)

# create the caribou agent object
caribou <- data.frame(x1, y1, sex, age)
coordinates(caribou) <- cbind(x=x0, y=y0)
row.names(caribou) <- IDs
```

Using a simple landscape-dependent correlated random walk, we simulate the movement of caribou across a heterogeneous landscape. Because we had just had fires, and we assume that fires have a detrimental effect on animal movement, we can see the long steps taken in the new, low quality, post-burn sections of the landscape.

```
#dev(4)
Plot(landscape[["habitatQuality"]], new=TRUE)

for (t in 1:10) {
  #crop any caribou that went off maps
  caribou <<- crop(caribou,landscape)
  drawArrows(from=SpatialPoints(cbind(x=caribou$x1, y=caribou$y1)),
             to=caribou, length=0.04, addTo="landscape.habitatQuality")

  # find out what pixels the individuals are on now
  ex <- landscape[["habitatQuality"]][caribou]

  #step length is a function of current cell's landscape quality
  sl <- 0.25/ex

  ln <- rlnorm(length(ex), sl, 0.02) # log normal step length
  sd <- 30 # could be specified globally in params

  caribou <<- crw(caribou, stepLength=ln, stddev=sd, lonlat=FALSE)
}
```



5.2 Polygons agents

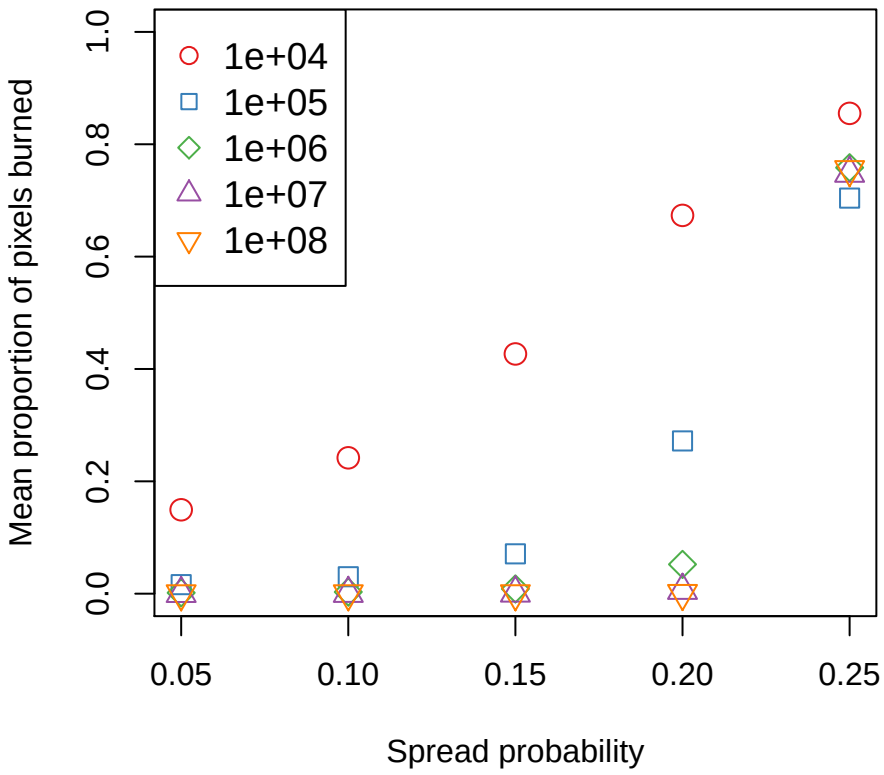
Analogously, it is possible to use `SpatialPolygons*`, but we haven't built `Plot` plotting methods for these yet.

6 Putting it all together

Running multiple simulations with different parameter values is a critical part of sensitivity analysis, simulation experiments, optimization, and pattern oriented modelling. Below is a greatly simplified example, using the sample `randomLandscapes` and `fireSpread` modules. *NB only two parameters are varied; no outputs are saved; and the analyses done here are kept simple for illustrative purposes. This will take a while to run!*

```
### WARNING this can take a while to run, especially for large mapSizes.

rasterOptions(maxmemory=1e9)

# list all parameter values to run sims with
parameters <- list(mapSize=round(sqrt(c(1e4, 1e5, 1e6, 1e7, 1e8))),
                   pSpread=seq(0.05, 0.25, 0.05))

# create data.frame with all parameter combinations
paramsdf <- expand.grid(parameters)

# outputs
nPixelsBurned <- numeric()
```

```
meanPixelsBurned <- cbind(paramsdf, pmean=NA, psd=NA)

set.seed(42)
for (i in 1:nrow(paramsdf)) {
  # initialize each simulation with a param combo from paramsdf
  mySim <- with(paramsdf,
                simInit(times=list(start=0.0, stop=20.0),
                        params=list(
                          .progress=list(.graphical=NA, .progressInterval=NA),
                          .globals=list(.stackName="landscape", burnStats="nPixelsBurned"),
                          randomLandscapes=list(nx=mapSize[i], ny=mapSize[i],
                                                inRAM=TRUE),
                          fireSpread=list(nFires=1000, spreadprob=pSpread[i],
                                          persistprob=0, its=1e6,
                                          returnInterval=10, startTime=0)
                          ),
                        modules=list("randomLandscapes", "fireSpread"),
                        path=system.file("sampleModules", package="SpaDES")))
  mySim <- spades(mySim)

  # collect stats for each run
  proportionBurned <- with(paramsdf, nPixelsBurned / (mapSize[i]^2))
  meanPixelsBurned[i, "pmean"] <- mean(proportionBurned)
  meanPixelsBurned[i, "psd"] <- sd(proportionBurned)

  # cleanup between runs
  rm(landscape, mySim, nPixelsBurned)
  for (j in 1:10) gc()
}

# overall statistics
pch <- c(21:25)
col <- brewer.pal(5, "Set1")

with(meanPixelsBurned, plot(pmean ~ pSpread, xlab="Spread probability",
                            ylab="Mean proportion of pixels burned",
                            ylim=c(0,1), pch=pch, cex=1.5, col=col))
with(parameters, legend("topleft", legend=formatC(mapSize^2, digits=0),
                              pch=pch, col=col, cex=1.2))
```



7 Additonal resources

7.1 `SpaDES` documentation and vignettes

Vignettes:

- `introduction`: Introduction to SpaDES [this vignette].
- `modules`: Building modules in `SpaDES`.
- `plotting`: Plotting with `SpaDES`.
- `debugging`: Debugging `SpaDES` simulations [Not yet written].

7.2 Reporting bugs

As with any software, there are likely to be issues. If you believe you have found a bug, please contact us via the package GitHub site: https://github.com/achubaty/SpaDES/issues. Please do not use the issue tracker for general help requests. We will soon be setting up a help mailing list soon for users who seek assistance using `SpaDES`.