Spatial Models

Therese Donovan

2020-05-06

Introduction

We'll now build an R version of the spreadsheet fire model we built in class.

We need to start with a landscape grid of a certain dimension, and we further need to populate the landscape grid with random numbers betweeen 0 and 1. These random numbers will represent the "flammability" index associated with each cell. The `matrix()` function can be used to create our landscape, and the `runif()` function (pronounced "R - unif") can be used to populate the grid cells with a random number between 0 and 1.

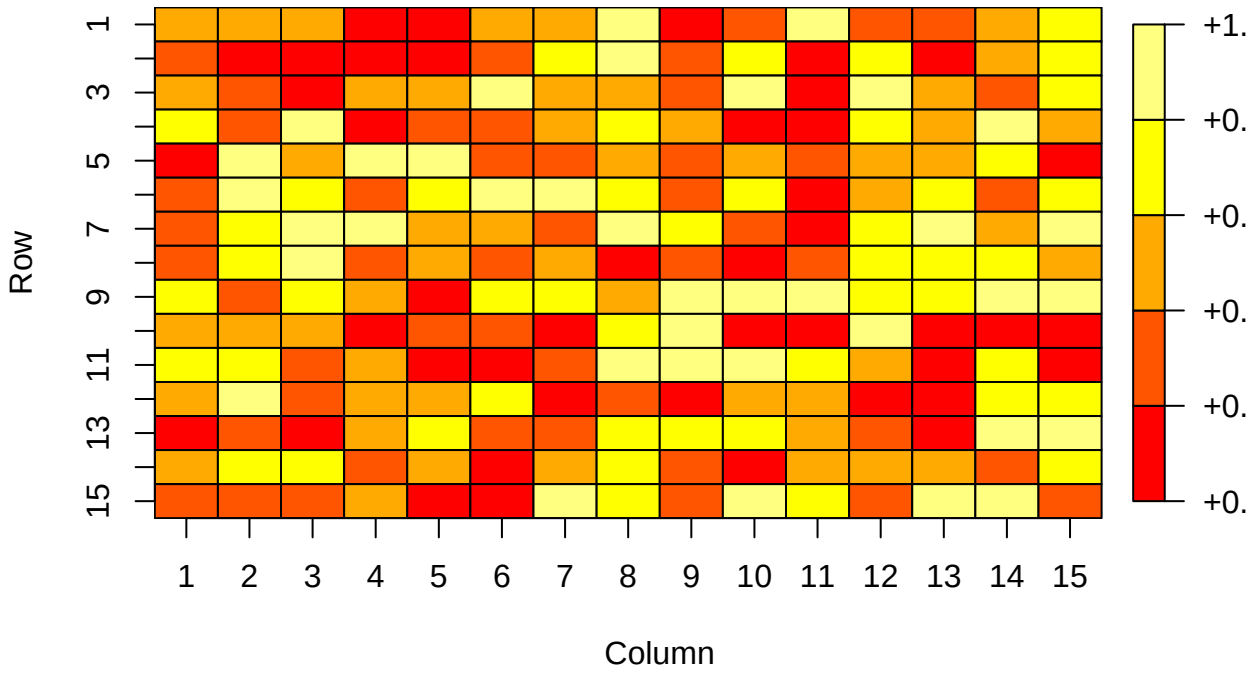
```
# set the landscape number of rows
ls.rows <- 15

# set the landscape number of columns
ls.cols <- 15

# compute the total number of cells (pixels) in the landscape
ls.cells <- ls.rows * ls.cols

# set a random number seed for reproducibility
set.seed(10)

# create the landscape matrix
ls <- matrix(data = runif(n = ls.cells, min = 0, max = 1), nrow = ls.rows, ncol = ls.cols)
str(ls)
```

```
##  num [1:15, 1:15] 0.5075 0.3068 0.4269 0.6931 0.0851 ...
```

Look in your Global Environment for this object. Note that it is a matrix, as identified by the [rows,columns] notation.

Let's plot this matrix the help of the plot.matrix package. This is a great little package for visualing R objects that are matrices. The vignette can be found at: https://cran.r-project.org/web/packages/plot.matrix/vignettes/plot.matrix.html

```
# add formatting to the matrix
library('plot.matrix')
plot(ls)
```



Now that we have a landscape, the next step is to set a burn threshold. Cells with random values above this threshold will have enough fuel to catch fire. It doesn't mean they *will* catch fire – it just means they *can* catch fire.

```
burn.th <- 0.6
```

The `plot()` function allows one to pass in several arguments to format the matrix colors and identify breaks. Next, we identify some "cut" intervals – those cells above the threshold and those below it – and their associated colors. Green will represent cells above the threshold, while gray will represent cells below the burn threshold.

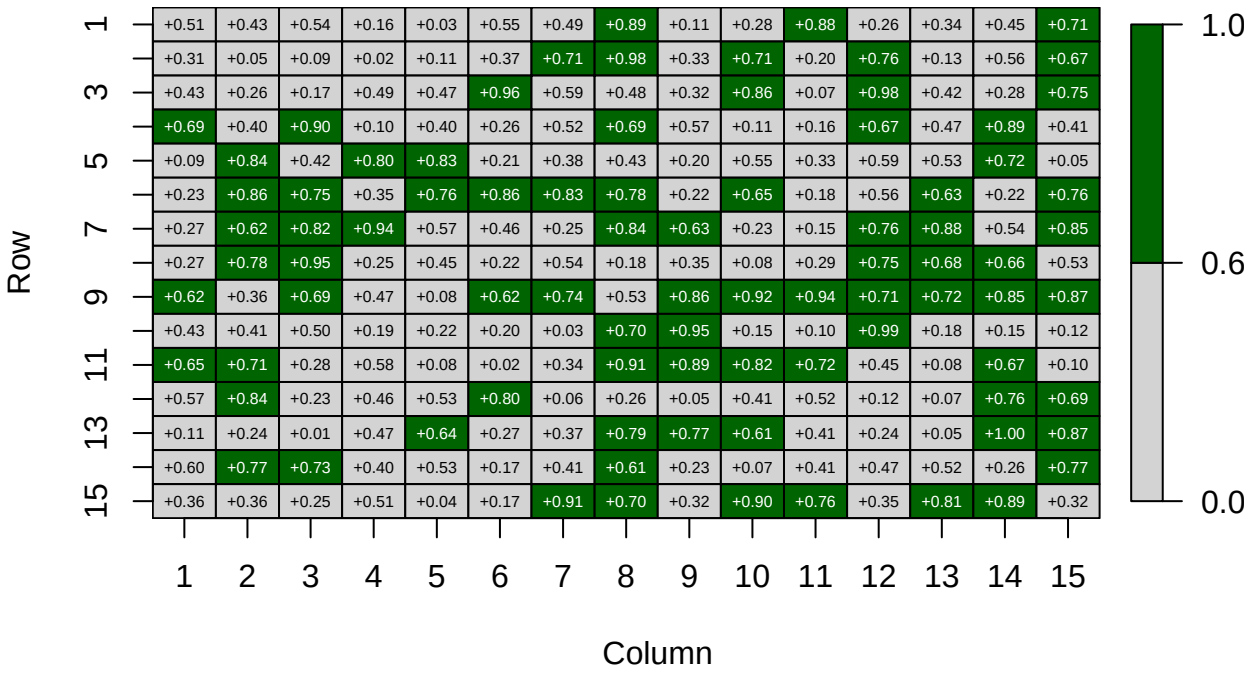
```
# identify the cut intervals; this will create two distinct classes
cuts <- c(0, burn.th, 1)

# for each cut class, assign a color; green will be flammable.
col = c('lightgray', 'darkgreen')
```

These objects can be used as arguments to the `plot()` method defined by the package, **plot.matrix**. To see the helpfiles for this package, you should specify not only the function name, but also the package name:

```
help(topic = "plot", package = 'plot.matrix')
```

Look for the arguments called "breaks" and "col". And while you are at it, look for the arguments called "text.cell", "digits", and "fmt.key".

```
# plot the ls matrix with specified formatting
plot(ls,
     breaks = cuts,
     col = col,
     fmt.key = "%.3f",
     digits = 2,
     text.cell = list(cex = 0.5))
```



Now we have our landscape, and are ready to implement a spatial model. But instead of working with matrices, we will turn to the **raster** package for spatial modeling.

The raster package

The raster package in R is an absolutely critical package for working with spatial objects. You can find the an overview of working with rasters in R at https://rspatial.org/raster/index.html.

Assuming you've installed the package, let's load it.

```
library(raster)
```

```
## Loading required package: sp
```

And of course, if you haven't used a package before, the very first thing to do is to look at the package's help index page.

```
help(package = raster)
```

Read the description file, and also scan the list of functions that are available. We can use the `raster()` function to turn a matrix to a raster. Let's first look at this function's helpfile:

```
help("raster")
```

Let's turn our matrix to a raster using the default values of the `raster()` function:

```
# turn our landscape matrix into a raster
ls.raster <- raster(ls)

# "show" the object
ls.raster
```

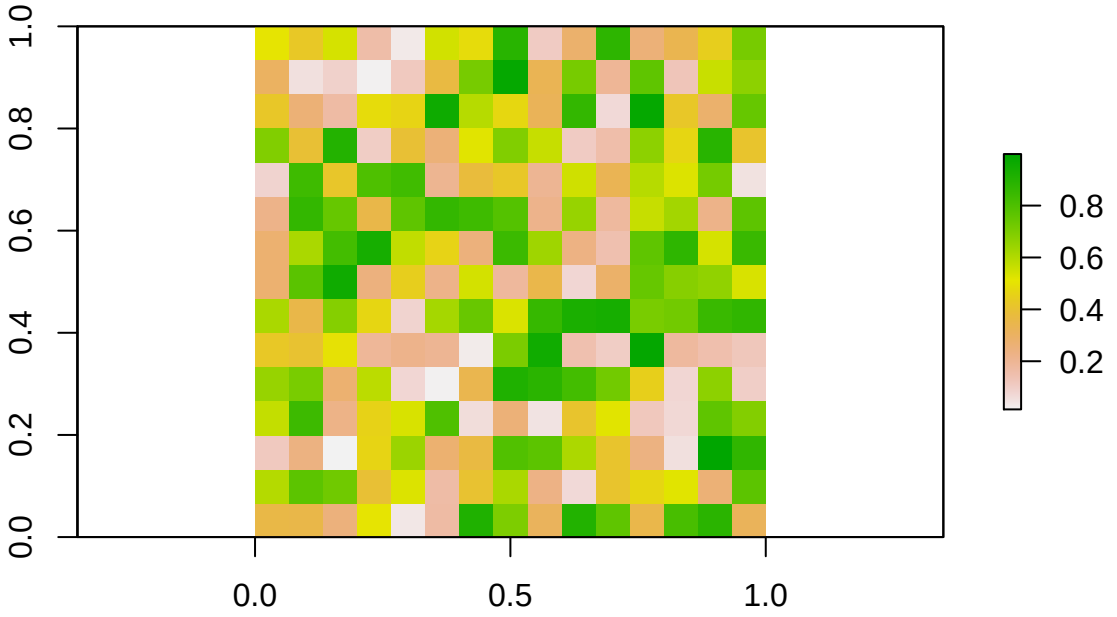
```
## class      : RasterLayer
```

```
## dimensions : 15, 15, 225  (nrow, ncol, ncell)
## resolution : 0.06666667, 0.06666667  (x, y)
## extent     : 0, 1, 0, 1  (xmin, xmax, ymin, ymax)
## crs        : NA
## source     : memory
## names      : layer
## values     : 0.01443391, 0.9986345  (min, max)
```

Note that our object, **ls.raster**, is of class "RasterLayer". It has 15 rows and 15 columns, with a total of 225 cells. Note that the raster goes from 0 to 1 in both directions (its extent), which means that each cell has a resolution of 0.667 units per side. You can change this if you want by setting the "xmn", "xmx", "ymn", and "ymx" arguments, or by setting the extent with the "ext" argument directly.

Notice also that this raster has a "crs" value of NA. CRS stands for "coordinate reference system". This is an unprojected raster map. We don't know what xy units are, nor do we know where on Earth this raster is located. This is fine for our little fire model, but in real world applications, the crs is essential.

```
# plot the raster
plot(ls.raster)
```



Our raster has random values between 0 and 1. For our R fire-model, we can simplify things by turning this raster into a binary raster, with

- 1 = burnable (at risk)
- 0 = unburnable (no risk, below the burn threshold)

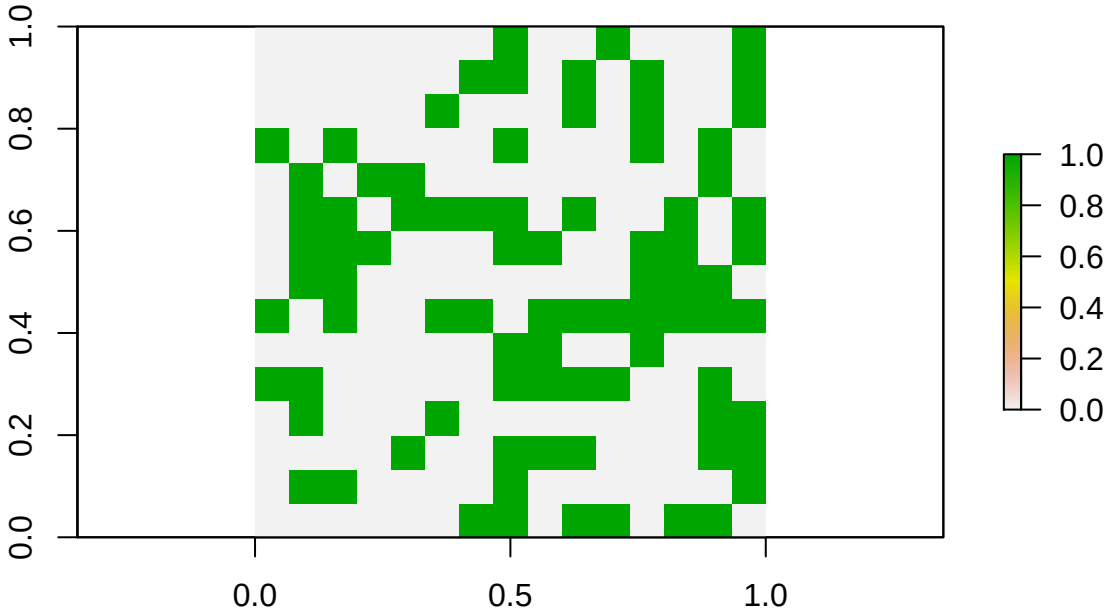
This can be easily done with a "test". Next, we ask "is each cell in the **ls.raster** > **burn.th**? R will return a raster of 1's (true) and 0's (false), which will be called **fuel.r**. This object will be central to our fire model.

```
# create a binary fuel raster that indicates cells at risk (1 = at risk, else 0)
fuel.r <- ls.raster > burn.th

# show the raster
fuel.r
```

```
## class      : RasterLayer
## dimensions : 15, 15, 225  (nrow, ncol, ncell)
## resolution : 0.06666667, 0.06666667  (x, y)
```

```
## extent     : 0, 1, 0, 1  (xmin, xmax, ymin, ymax)
## crs        : NA
## source     : memory
## names      : layer
## values     : 0, 1  (min, max)
```

```
# plot the raster
plot(fuel.r)
```



Our map now contains just two values: 1's are cells that are flammable and have the potential of burning, or become burned, while 0's cannot burn.

Ok! Let's start a fire! We need to make sure that our fire starts in a pixel that has enough fuel. Eyeballing the map, it looks like the cell in row 9, column 6 is has a value of 1. Let's confirm this:

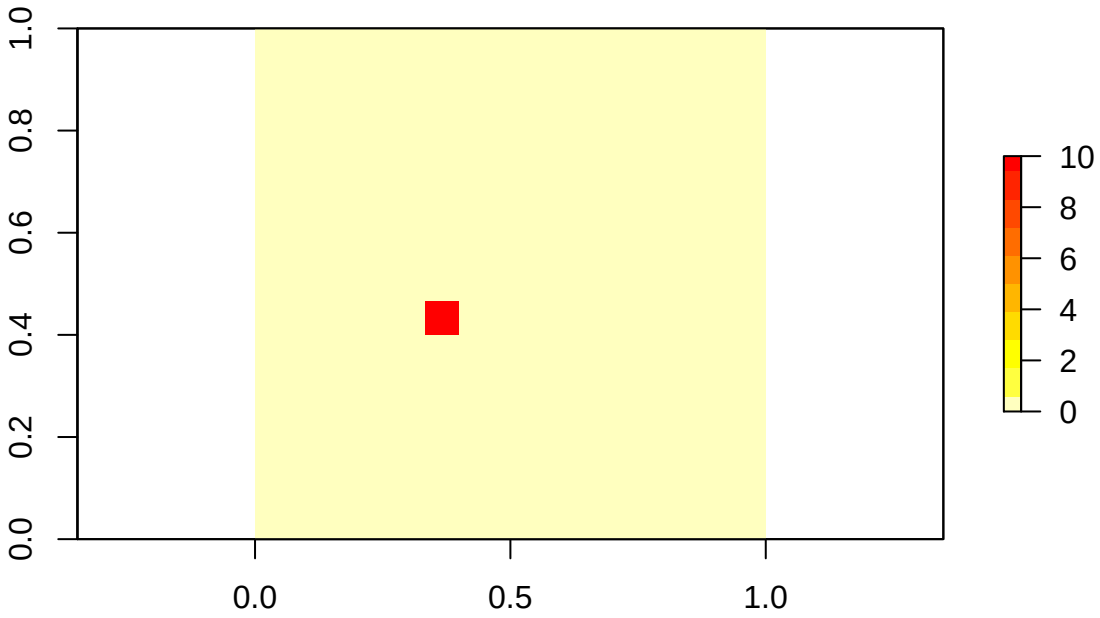
```
# start a fire at a flammable location
fuel.r[9,6]
```

```
## [1] 1
```

Yup! It's flammable.

Next, let's create a new raster called **burn.t**. This will be a raster that contains cells that are burned (currently burning or were previously burned). This will be another map with just two values: 0 (unburned) or 10 (burned). You'll understand why I selected this value shortly.

```
# create a matrix of 0's
burn.t <- matrix(0, nrow = ls.rows, ncol = ls.cols)

# set cell [9,6] to 10, which means it has burned
burn.t[9,6] <- 10

# turn the matrix to a raster
burn.t <- raster(burn.t)

# plot the raster
plot(burn.t, col = heat.colors(n = 10, rev = T))
```



Here, the red indicates a cell with a value of 10. Yellow indicates non-burned cells. These colors have been set by specifying values in the "col" argument – the exact values are passed in from the `heat.colors()` function, with a reverse order. You can play with this function in your console if your like.

```
heat.colors(n = 10)
```

```
##  [1] "#FF0000" "#FF2400" "#FF4900" "#FF6D00" "#FF9200" "#FFB600" "#FFDB00"
##  [8] "#FFFF00" "#FFFF40" "#FFFFBF"
```

```
heat.colors(n = 10, rev = TRUE)
```

```
##  [1] "#FFFFBF" "#FFFF40" "#FFFF00" "#FFDB00" "#FFB600" "#FF9200" "#FF6D00"
##  [8] "#FF4900" "#FF2400" "#FF0000"
```

Each of these values is a "color-hex" code for a given color. See https://www.color-hex.com/ if you would like to explore this more.

OK, now we have started a fire. The next step is to see how it will spread. In our spreadsheet, we used a Queen rule to examine each pixel's surrounding cells. We asked if a neighboring cell was burning, AND if it had enough fuel to catch fire. Here, we will do this in two steps.

First, we can use the `focal()` function (from package raster) to do a moving window analysis. Let's first look at this function's helpfile.

```
help(focal)
```

The description reads: Calculate focal ("moving window") values for the neighborhood of focal cells using a matrix of weights, perhaps in combination with a function. We need to send in a raster layer that we wish to run our focal function on. And, importantly, we need to set the moving window as a matrix of weights. We are interested in a 3*3 moving window to set the Queen's rule:

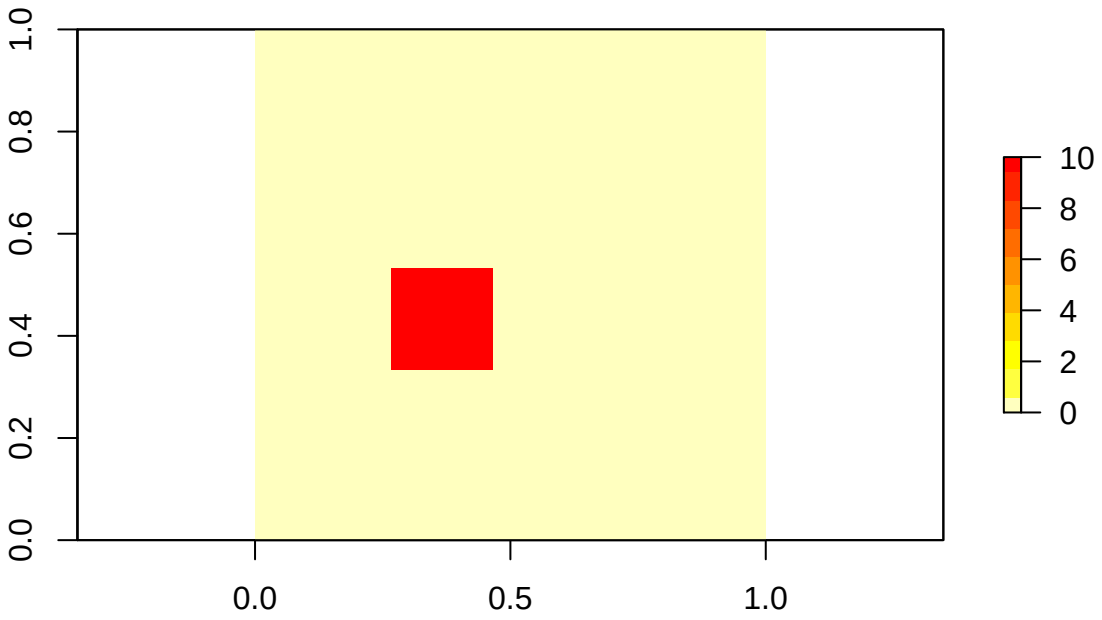
```
# set the Queen rule matrix
w <- matrix(1, nrow = 3, ncol = 3)

# show the matrix
w
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
```

```
## [2,]    1    1    1
## [3,]    1    1    1
```

The focal function has other arguments that we need to pay attention to. First, you've probably noticed that our raster has edges, and if we place this weight matrix over the upper left cell, the matrix will spill off the raster. This is the external buffer, and we can proactively deal with how to handle the buffer by setting the "pad" argument to TRUE and the "padValue" argument to 0. This provides a buffer of 0 around our whole raster. Finally, the "fun" argument allows you specify exactly what should be returned for each focal pixel. Here, we will use the "max" function: we want to place the weight matrix over the top of each and every cell (a 3*3 matrix centered on each pixel), and return the maximum of those 9 values. Let's try it:

```
focal.max <- focal(burn.t, w = w, pad = T, padValue = 0, fun = max)

plot(focal.max, col = heat.colors(n = 10, rev = T))
```



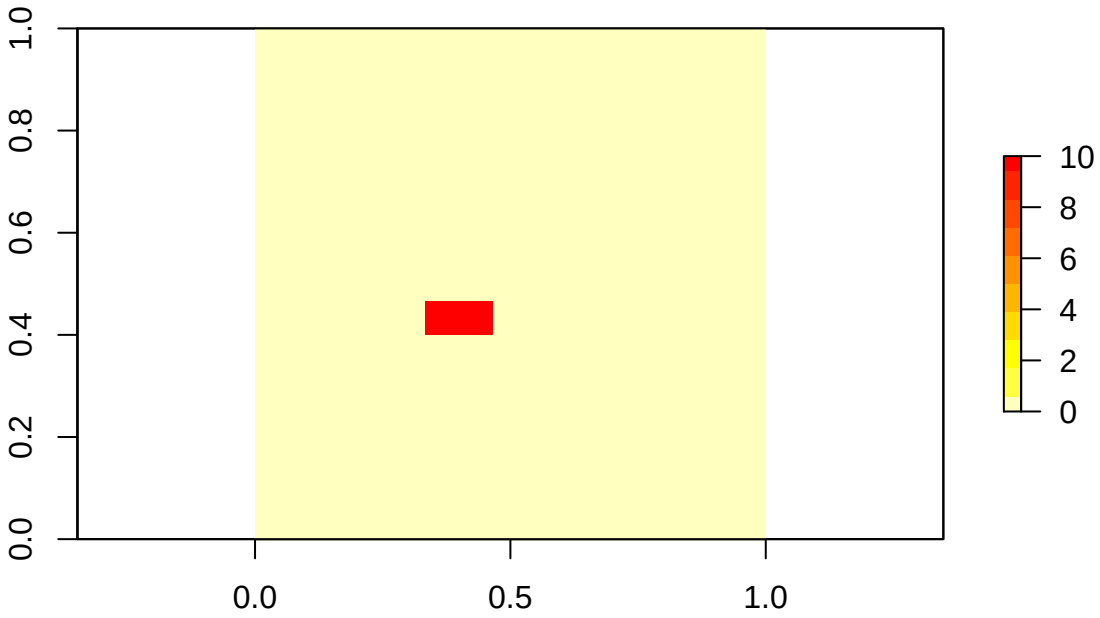
First, the focal function returns a raster, where each cell gives the result of the focal function call. You should see that the number 10 is returned for all cells that are next to our initial spark (and including the initial spark).

It is important to keep in mind that the **focal.max** raster gives us the *potential* fire spread - not the actual fire spread. It shows the cells in which the fire in this time step *can* spread to, and including the cells that have already burned. But in order for it to actually spread to these potential cells, they must have enough flammability. That is, their fuel level must exceed the threshold.

Now we have two rasters that can be used to predict the fire spread in the next time step: **fuel.r** is a binary raster that indicates if a cell has enough fuel to burn, while **focal.max** indicates which cells are in the line of fire (pun intended!).

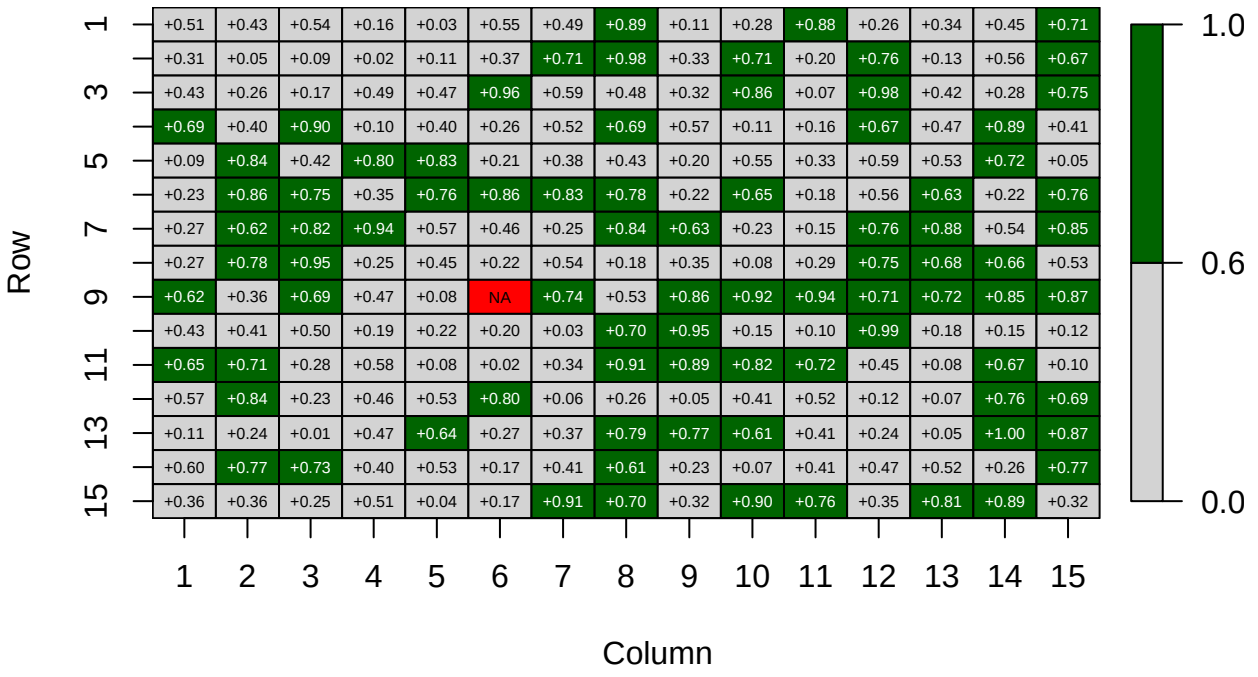
We can use both of these rasters to create the burn raster at time t + 1. We will simply multiply the **focal.max** raster by the **fuel.r** raster – only those cells that are in the line of fire *and* also burnable will be returned, and the resulting raster will once again contain only 0's or 10's.

```
# create the burn map at the next time step
burn.t1 <- fuel.r * focal.max

# plot the new landscape
plot(burn.t1, col = heat.colors(n = 10, rev = T))
```



Let's see if we can make sense of this result by matching it up to our original landscape matrix, which is called **ls**. First, let's set our initial fire start to a value of NA so that we can easily identify it when we plot it.

```
# set the initial fire cell value to NA
ls[9,6] <- NA

# plot the matrix; note the argument na.col (na color) is set to red
plot(ls,
     breaks = cuts,
     col = col,
     fmt.key = "%.3f",
     digits = 2,
     text.cell = list(cex = 0.5),
      na.col = "red"
     )
```



Notice that our "model" indicates that the cell to right of our spark cell has burned in the next time step, which makes sense.

What do you think will happen in the next 4 time steps?

To model several time steps, we will want to set up a loop and loop through the time steps. First, we'll create an empty list to hold our results. We'll populate the first element of this list with our **burn.t** raster. Then we'll start our loop. This loop will look at the previous time step's raster, and run the focal function to get the **focal.max** raster – remember, these are cells in the line of fire (burnable or already burned). Then, we'll once again multiply this raster by the **fuel.r** raster to give us our map in the next time step, which we'll call **burn.t1**. Finally, we'll use the `append()` function to append this raster to our results list, and the loop can continue.

Here we go:

```
 # set up results list to hold results
 results <- list()

 # initialize the starting landscape
 results[[1]] <- burn.t

 # loop through the landscape for 4 more time steps
for (i in 2:5) {

  focal.max <- focal(results[[i - 1]], w = w, pad = T, padValue = 0, fun = max)
  burn.t1  <- fuel.r * focal.max
  results <- append(results, values = burn.t1)

}
```

Have a look at the **results** list in your Global Environment. Click on the magnifying glass to get a better look, or use the `str()` function instead.

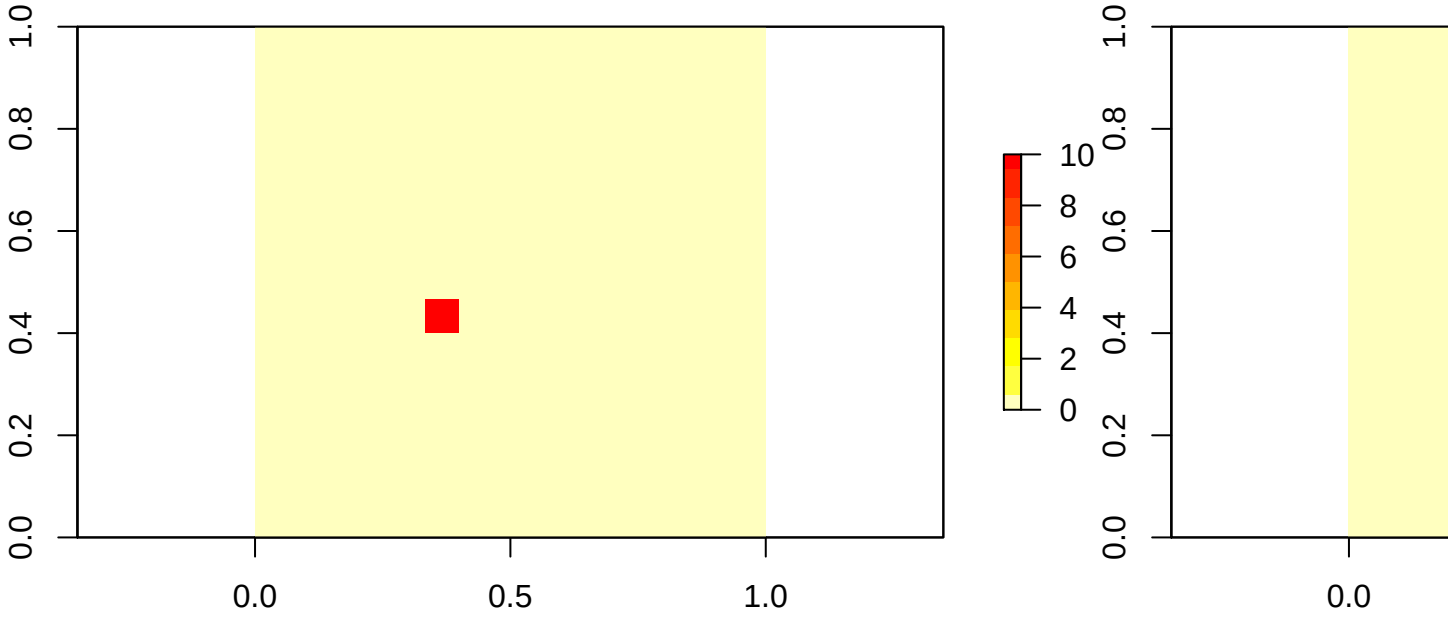
```
str(results, max.level = 1)
```

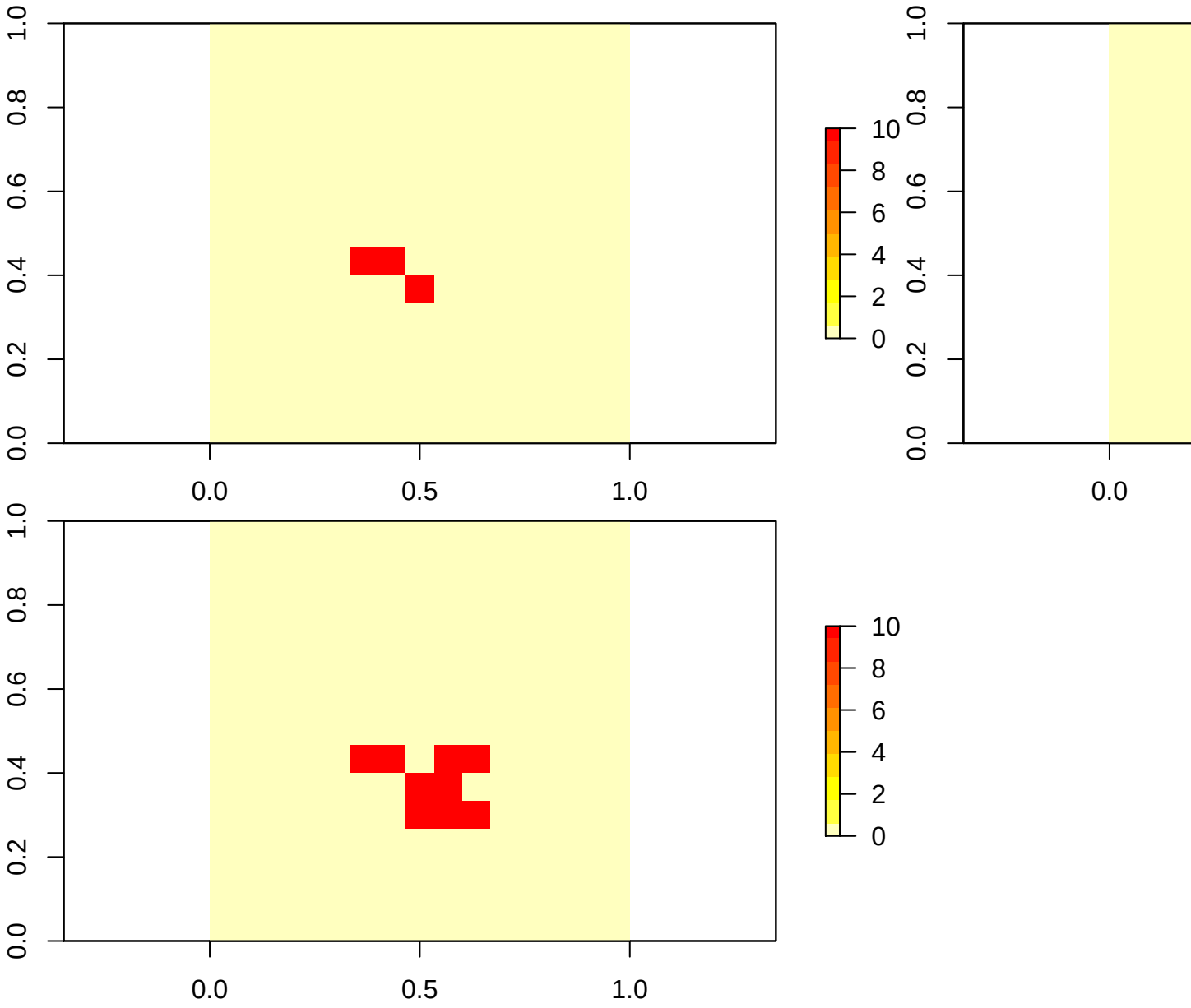
```
## List of 5
##  $ :Formal class 'RasterLayer' [package "raster"] with 12 slots
##  $ :Formal class 'RasterLayer' [package "raster"] with 12 slots
##  $ :Formal class 'RasterLayer' [package "raster"] with 12 slots
##  $ :Formal class 'RasterLayer' [package "raster"] with 12 slots
##  $ :Formal class 'RasterLayer' [package "raster"] with 12 slots
```

As you can see, this list contains 5 elements, and each element is an object of class "RasterLayer", depicting the change in landscape through time.

We can use the `lapply()` function to show all of the raster plots. This function is used when you wish to apply a particular function across each element of a list (lapply stands for list-apply). We are going to march our way through a list (named **results**), and the function we will apply to each list element is the `plot()` function.

```
# show the maps
invisible(lapply(results, FUN = plot, col = heat.colors(n = 10, rev = T)))
```





Technically, the model we have built is a cellular automata model. Tony mentioned that these models have the following properties:

1. The map at time t+1 depends only on the map state at time t
2. You an calculate the state of any cell at time t+1 from the states of the corresponding cell at time t and its immediate neighbors.

You can use other functions in `lapply()`. For example, let's tally the number of 0 and 10 values in each time step:

```
# show the maps
lapply(results, FUN = freq)
```

```
## [[1]]
##      value count
## [1,]     0   224
## [2,]    10     1
```

```
##
## [[2]]
##      value count
## [1,]     0   223
## [2,]    10     2
##
## [[3]]
##      value count
## [1,]     0   222
## [2,]    10     3
##
## [[4]]
##      value count
## [1,]     0   218
## [2,]    10     7
##
## [[5]]
##      value count
## [1,]     0   216
## [2,]    10     9
```

Adding the stochastic coin flip

Tony mentioned in his lecture that we don't need to force each cell to catch fire if it has enough flammability and touches a neighboring cell that is burning. He mentioned we can let this be stochastic or probabilistic instead. How would we do that?

As usual, there are many possibilities! It depends on your modeling goals, of course. One easy, easy thing to do is to multiply the initial random matrix (which gives our fuel levels) by another set of random numbers, which gives the probability of actually burning given a neighboring fire. If **both** the fuel level *and* the random number multiplied together exceed some threshold, the fire will catch and the model is essentially the same as what we have already used.

```
# create a random matrix
randoms <- matrix(data = runif(n = ls.rows*ls.cols), nrow = ls.rows, ncol = ls.cols)

# turn this into a random binary, depending on what you want the probability of burn to be.
random.bin <- ifelse(test = randoms <= 0.4, yes = 1, no = 0)

# turn the matrix into a raster
random.bin <- raster(random.bin)

# multiply the original bin.m by the random binary matrix
bin.m <- fuel.r  * random.bin
```

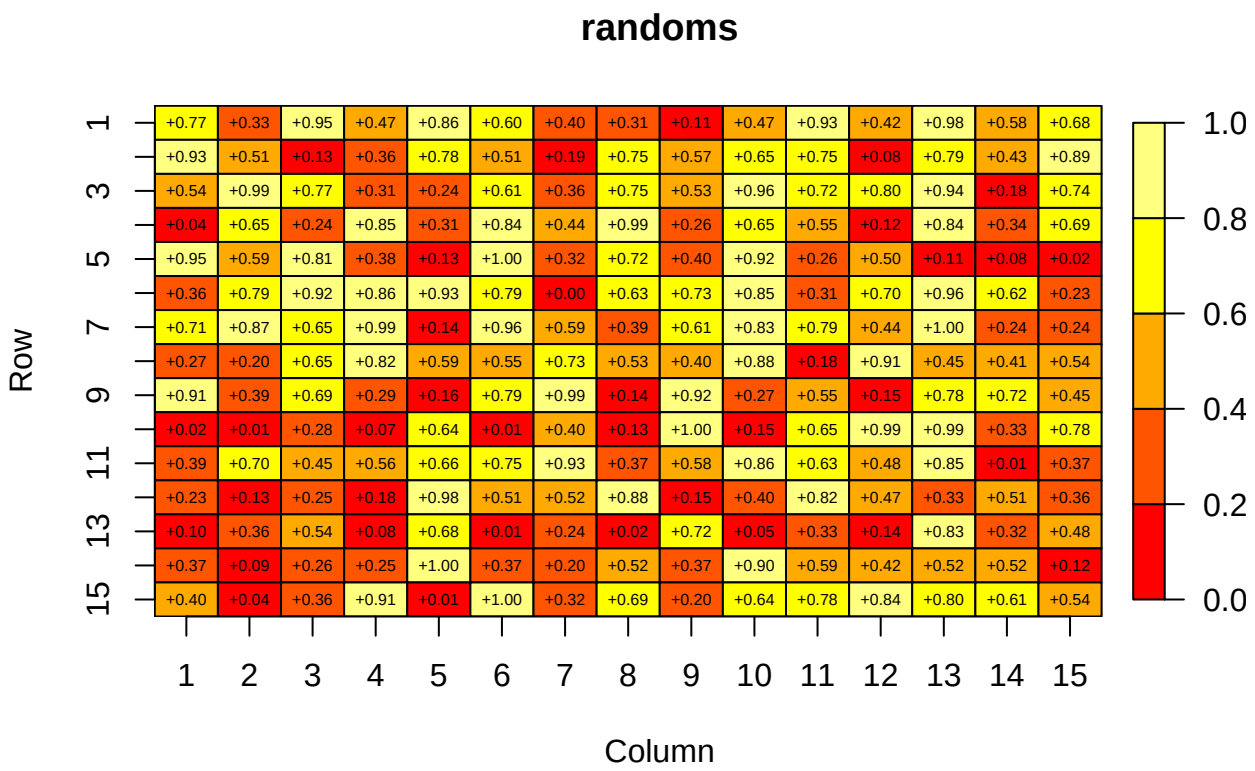
Of course there are many ways to skin a cat. Can you think of an alternative?

Adding wind

Tony also mentioned that wind direction may play a role in flammability. For example, if the wind blows left to right across our landscape, we might increase flammability of cells ahead of wind and decrease behind the wind. How might this done?

Originally, our landscape was populated by a set of random numbers ranging from 0 to 1.

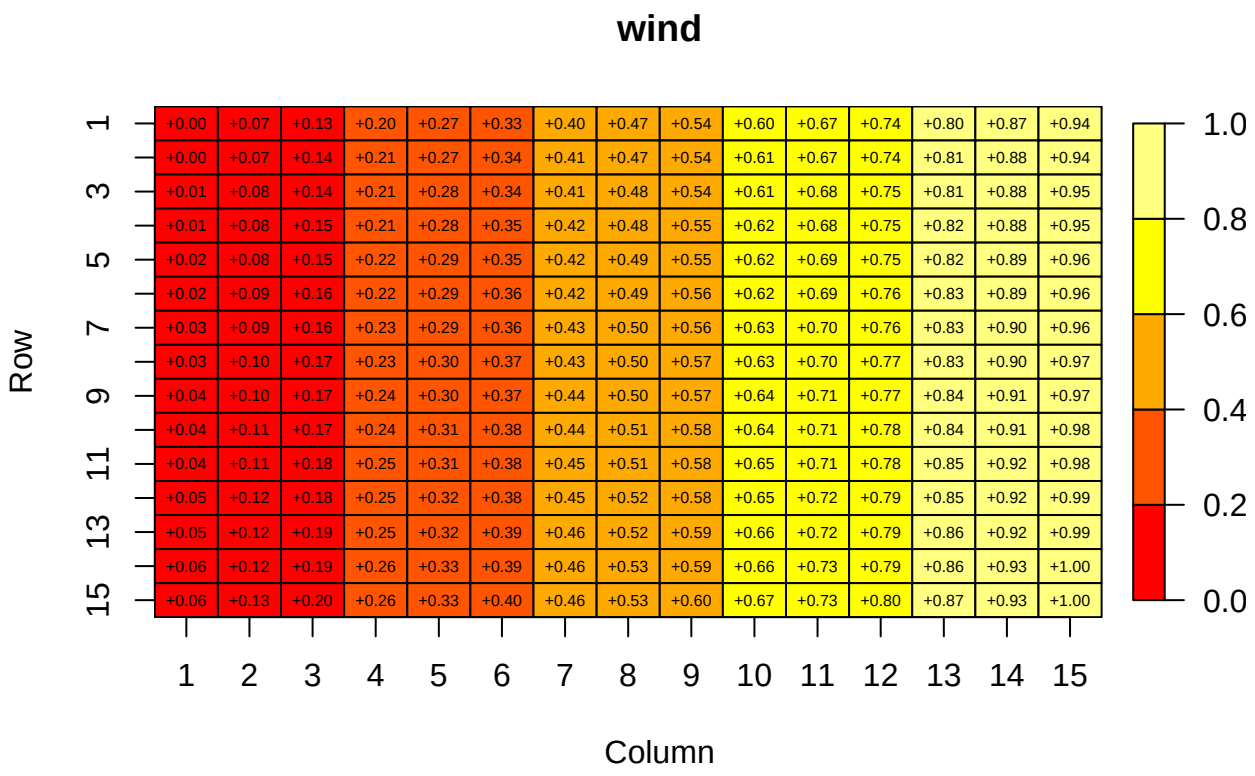
```
# create a random matrix
randoms <- matrix(data = runif(n = ls.rows*ls.cols), nrow = ls.rows, ncol = ls.cols)

# plot the matrix
plot(randoms,
     fmt.key = "%.3f",
     digits = 2,
     text.cell = list(cex = 0.5)
     )
```



As expected, there is no real pattern to the cell values. Now let's create a flammability matrix that has pattern to it.

First, let's set the wind information. We want a sequence of 225 numbers that start at 0 and extend to 1. Here, our data for the matrix will be ordered, set by the `seq()` function.

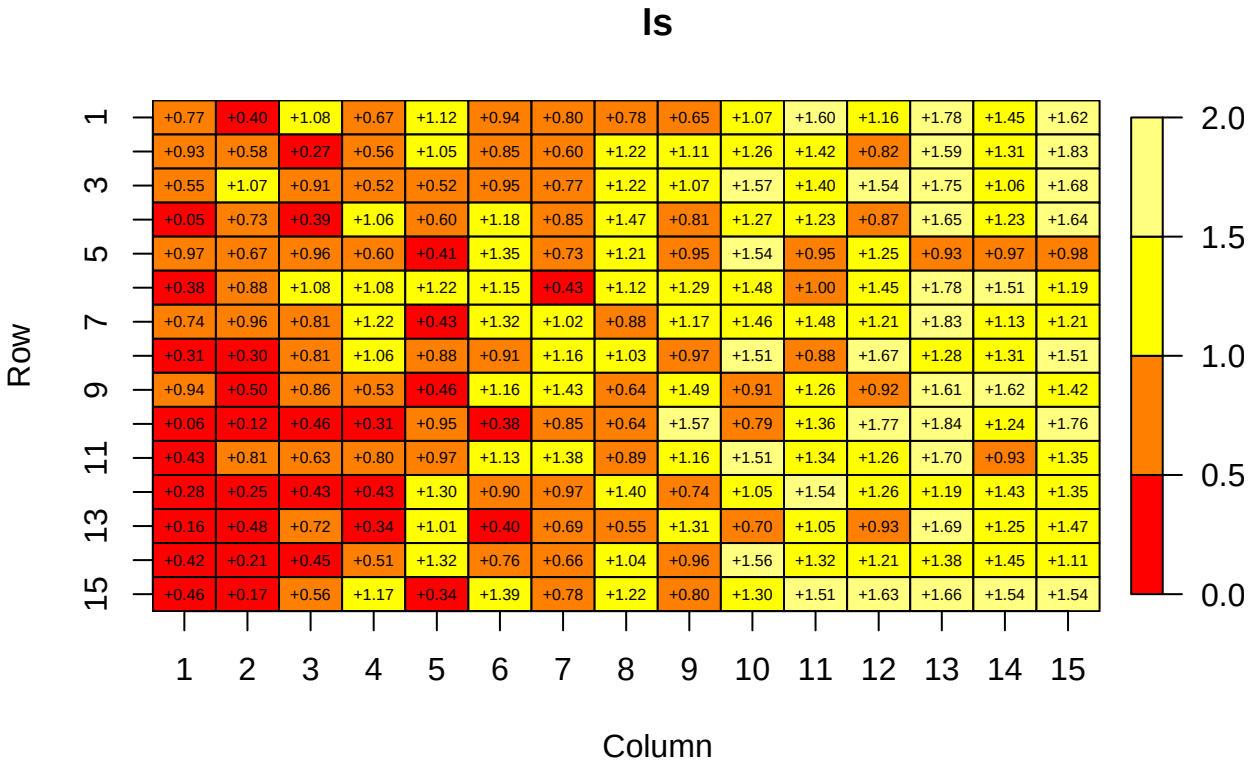
```
# create the data for our landscape matrix with the seq function
data <- seq(from = 0, to = 1, length.out = ls.rows*ls.cols)

# create the matrix
wind <- matrix(data = data, nrow = ls.rows, ncol = ls.cols, byrow = FALSE)

# plot the matrix
plot(wind,
     fmt.key = "%.3f",
     digits = 2,
     text.cell = list(cex = 0.5)
     )
```



As you can see, the wind scores now have a pattern with increasing values from left to right.

Now, let's add some flammability on top.

```
ls <- randoms + wind

# plot the matrix
plot(ls,
     fmt.key = "%.3f",
     digits = 2,
     text.cell = list(cex = 0.5)
     )
```



Conclusions

Tony concludes the spatial models module by emphasizing that the type of spatial model one can use is highly dependent on the scale of the pixels *and* the time step. And never forget: any model should be purpose driven; i.e., your objectives should drive the model, rather than vica versa. As always, rapid prototyping should rule the day!

Now that you know a bit more about spatial analysis in R, here's a few resources you might find useful:

- https://www.rspatial.org/
- https://cran.r-project.org/web/views/Spatial.html
- https://geocompr.robinlovelace.net/
- https://www.neonscience.org/raster-data-r