Forest Fire Propagation: A Cellular Automata Approach

Assignment 1b | Probability & Statistics SA 2023-2024

Frova Davide

Abstract

The simulation that we are going to create will represent a forest made of different types of vegetation.

By spontaneous ignition or by a predifined starting point a forest fire born and spread, evolving in time in a Markov Chain manner.

Each cell will have a certain probability of ignition/burnout, which will be affected by it's previous state and the state of it's neighbours.

The probabilities will be affected by different factors:

- Specific cell type / state
- Near by cells types / states
- Natural events and characteristics
- Presents of puddles of water
- Presents of different kind of rivers
- Wind
- Altitude

Premise about code development

The development of some parts of the code that will follow were done with the help of the tool Github Copilot Chat, this was to fill in the missing syntax/functions knowledge about the R language. The tool was generating snippets of code to show examples on the usage of certain R functions, and it was mainly generating explanations about R functionalities, and as an help in finding not so easy to spot bugs and fixes to various problems.

Design and Implementation of the Forest Fire Model

Preparation-Step: Including the necessary external libraries

```
# Install and load the animation package
if (!require(animation)) install.packages("animation")
```

```
## Caricamento del pacchetto richiesto: animation
```

```
library(animation)
```

Defining the different types of cells

Here is the definition of the different cell types / states.

We will refer to *cell type* to indicate the type of the cell when we are looking at the forest model in a fixed time, and to *cell status/state* when we are in a on-going time frame, were the *type* of the cells is varying through time.

Pseudocode:

```
cell_states is a dictionary/set of states
  1. water
  2. dryGrass
  3. denseTrees
  4. burning
  5. normalForest
  6. burned
```

R code:

```r
cell_states <- c(
  "Water", # 1
  "DryGrass", # 2
  "DenseTrees", # 3
  "Burning", # 4
  "NormalForest", # 5
  "Burned" # 6
)
```

We preferred using the indexes to refer to a cell_state in the following code sections. The cell_states dictionary names will be used in the plot visualization, as a legend for the color coding.

Initialization of the forest grid

The forest will be represented by an n*m grid / matrix, each cell has it's type defined (index in the dictionary of types) and it's own probability of ignition/burnout.

In this function we will create the matrix and set the default value of the status and probability.

Pseudocode

```
initialize_grid is a function with parameters (n rows, m cols)
  grid is a matrix n by m
  foreach element of grid
    set the status field to normalForest
    set the probability to 0.0
  return the grid
```

R code:

```r
initialize_grid <- function(rows, cols) {
  # Create a list of matrices
  # status will store the status of the cells
  # prob   will store the probability of the cells
  grid <- list(
    "status" = matrix(nrow = rows, ncol = cols),
    "prob" = matrix(nrow = rows, ncol = cols)
  )

  # Foreach element set the status and probability
  for (y in 1:rows) {
    for (x in 1:cols) {
      grid$status[y, x] <- 5 # Normal Forest
```

```
      grid$prob[y, x] <- 0.0
    }
  }
  return(grid)
}
```

In the R code we have different ways of creating a matrix that for each cell contains 2 fields.

In this case we chose to create a matrix for the statuses and a matrix for the probabilities. Then we store these two matrices in a list called *grid*

Test of the function

```
initial_forest <- initialize_grid(3, 3)
initial_forest  # Original Values
```

```
## $status
##      [,1] [,2] [,3]
## [1,]    5    5    5
## [2,]    5    5    5
## [3,]    5    5    5
##
## $prob
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

```
# Set the status matrix with the names of the states
initial_forest$status <- matrix(
  sapply(
    initial_forest$status,
    function (val) cell_states[val]),
    nrow = nrow(initial_forest$status),
    ncol = ncol(initial_forest$status
  )
)

initial_forest  # Names instead of values
```

```
## $status
##      [,1]           [,2]           [,3]
## [1,] "NormalForest" "NormalForest" "NormalForest"
## [2,] "NormalForest" "NormalForest" "NormalForest"
## [3,] "NormalForest" "NormalForest" "NormalForest"
##
## $prob
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Filling the grid randomly

Now we generated a grid with default values inside. We could run a simulation based on it, but in our case we want more interesting characteristics to observe. That's why we will develop some functions to fill the

forest grid in random and semi-random ways with other values for the cell types.

Random values for the cell states

A basic first approach is to assign to each cell of the grid a random value from the cell_states dictionary

Pseudocode

```
random_fill is a function with parameters (forest_grid)
  foreach cell in forest_grid
    new_status is a random pick from cell_states
    set the cell "status" field to new_status
  return forest_grid
```

R code:

```
random_fill <- function(forest_grid){
  max_y <- dim(forest_grid$status)[1]
  max_x <- dim(forest_grid$status)[2]
  for (y in 1:max_y) {
    for (x in 1:max_x) {
      # Sample from the indexes of the cell_states
      status <- sample(1:length(cell_states), 1)
      forest_grid$status[y, x] <- status
    }
  }
  return (forest_grid)
}
```

Test of the function

```
initial_forest <- initialize_grid(3, 3)
filled_forest <- random_fill(initial_forest)
filled_forest
```

```
## $status
##      [,1] [,2] [,3]
## [1,]    3    2    3
## [2,]    5    4    1
## [3,]    2    3    3
##
## $prob
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

We will further develop more advanced ways of filling the grid, but for now this is enough to continue with the definition of other core functionalities.

Design of the plot grid function

Now that we have a way of randomly generate the forest grid, we can think of a way to visualize it.

Printing it with the matrix R representation is not the best as we saw before, so we are going to develop a function to plot our matrix in a color coded way.

Each color will represent a different type of cell.

Color palette

First we define a color palette to represent the different cells

Pseudocode

```
cell_states_colors is a dictionary containing colors
  1. Water color
  2. Dry grass color
  3. Dense trees color
  4. Buring/Fire color
  5. Normal forest color
  6. Burned color
```

R code

```r
cell_states_colors <- c(
  "#0000FF", # Blue / Water
  "#ADFF2F", # GreenYellow / DryGrass
  "#006400", # DarkGreen / DenseTrees
  "#FF0000", # Red / Burning
  "#008000", # Green / NormalForest
  "#201F1F"  # DarkGray / Burned
)
```

We represent the colors using the hexadecimal values, one for each of the cell types.

Plot grid function

Now we can define a function that will plot a representation of our matrix, that will create a rectangle/square for each cell, with the corresponding status color.

Pseudocode

```
plot_grid is a function with parameters (forest_grid, cell_states_colors)
  foreach row of forest_grid
    row_of_cells is an empty list
    foreach cell in the current row
      add a rectangle with the corresponding cell state color to the list
    plot the row_of_cells with the rectangles in line
    go to a new line in the plot
```

The R code implementation varied a lot during the development. The main changes where:

- Adding a title with various information to the plot
- Optimization in terms of speed of the plotting function
- Optimization of the quality of the produced plot

This was the first implementation

```r
plot_grid <- function(forest_grid) {
  # I flip&transpose the grid for better visualization
  # and consistency with the print of the values (0,0 top left)
  flipped_grid <- apply(forest_grid, 2, rev)
  transposed_again <- t(flipped_grid)

  # Get unique values to only take the colors needed
  # (image() will be angry otherwise)
  colors <- sort(unique(as.vector(forest_grid)))
```

```
  # Map each cell status to its color
  colors <- sapply(colors, function(val) cell_states_colors[val])

  # Print the image of the grid
  image(transposed_again, col = colors)
}
```

This is the final implementation with the above listed changes

R code

```
plot_grid <- function(forest_grid, plot_title = NULL) {
  # We will be working only on the status matrix
  forest_grid <- forest_grid$status

  # Increase the size of the grid matrix
  # This is to achieve a plot/graphic with higher quality
  # Each cell is now 10 cells
  # thanks to Github Copilot Chat
  enlarged_forest_grid <-
    kronecker(
      forest_grid,
      matrix(1, nrow = 10, ncol = 10)
    )

  # Get the number of rows and columns of the enlarged matrix
  nrow_enlarged <- dim(enlarged_forest_grid)[1]
  ncol_enlarged <- dim(enlarged_forest_grid)[2]

  # Create an empty plot
  plot(
    0, 0,
    type = "n",
    xlim = c(0, ncol_enlarged),
    ylim = c(0, nrow_enlarged),
    xlab = "", ylab = "", xaxt = "n", yaxt = "n",
    asp = 1
  )

  # Add the title to the plot
  if (!is.null(plot_title)) {
    title(main = plot_title)
  }

  # Create a color matrix
  # This will map each cell status to its cell_status_color
  color_matrix <- matrix(
    cell_states_colors[enlarged_forest_grid],
    nrow = nrow_enlarged,
    ncol = ncol_enlarged,
    byrow = TRUE
  )

  # Convert the color matrix to an image
  # Using raster for faster plotting
```

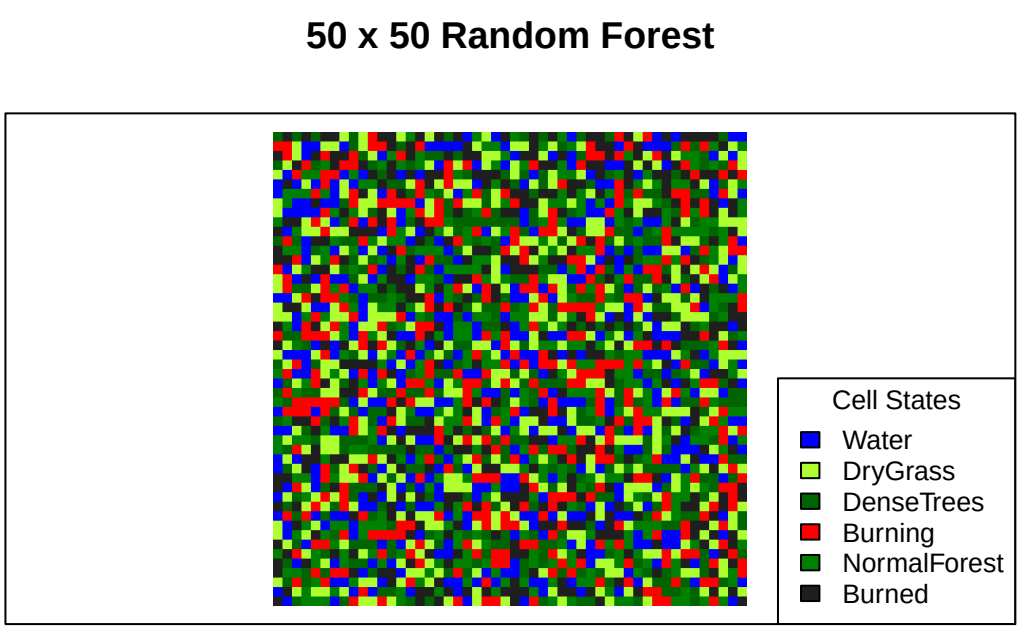
```
  img <- as.raster(color_matrix)

  # Draw the image
  rasterImage(img, 0, 0, ncol_enlarged, nrow_enlarged)

  # Add a legend
  legend(
    "bottomright",  # Position of the legend
    legend = cell_states,  # Labels for the legend items
    fill = cell_states_colors,  # Colors for the legend items
    cex = 0.8,  # Size of the legend items
    title = "Cell States"  # Title of the legend
  )
}
```

Test of the function

```
initial_forest <- initialize_grid(50, 50)
filled_forest <- random_fill(initial_forest)
plot_grid(filled_forest, "50 x 50 Random Forest")
```



Develop the neighbours function

A main key point of the evolution of the forest grid are the neighbours of a given cell.

This is because the probability of a cell to ignite, is the sum of the probabilities of it's neighbours to ignite it.

In other words, to determine if a cell becomes Burning, we need to check if its neighbours would spread fire to it or no.

Here is the pseudocode to detect the list of neighbours of a cell at position y, x

Pseudocode

```
neighbours is a function with parameters (forest_grid, y, x)
  neghs_pos is a list of possible neighbours positions given y,x
    (y - 1, x - 1),     # Top left
    (y, x - 1),         # Top center
    (y + 1, x - 1),     # Top right
    (y - 1, x),         # Center left
    (y + 1, x),         # Center right
    (y - 1, x + 1),     # Bottom left
    (y, x + 1),         # Bottom center
    (y + 1, x + 1)      # Bottom right

  valid_neghs is an empty list neighbours positions

  foreach neghs_pos
    if the position is valid (inside the grid)
      add the neighbour position to valid_neghs

  return valid_neghs
```

In the actual R code we did two main changes.

We define a neighbour as a more complex object, we do not store only the position but also the current state of the cell given by the corresponding *forest_grid$status* value.

The neighbour has the following structure:

```
neighbour:
  "state" : val
  "coords :
    "x" : val
    "y" : val
```

And secondly we do not take the y and x values as single parameters but inside a cell array of dim=2, which is composed as (y, x).

R code

```r
neighbours <- function(forest_grid, cell) {
  x <- cell[2]
  y <- cell[1]

  max_x <- dim(forest_grid$status)[2]
  max_y <- dim(forest_grid$status)[1]

  # Possible neighbours coordinates
  result <- list(
    c(y - 1, x - 1),     # Top left
    c(y, x - 1),         # Top center
    c(y + 1, x - 1),     # Top right
    c(y - 1, x),         # Center left
    c(y + 1, x),         # Center right
    c(y - 1, x + 1),     # Bottom left
```

```
    c(y, x + 1),          # Bottom center
    c(y + 1, x + 1)       # Bottom right
  )

  # Final list with the valid neighbours
  valid_result <- list()

  for (negh in result) {
    # Check if it is a valid coordinate
    if (negh[1] > 0 && negh[2] > 0 && negh[2] <= max_x && negh[1] <= max_y) {
      # Create the neighbour object
      negh_obj <- c(
        "state" = forest_grid$status[negh[1], negh[2]],
        "coords" = c(
          "x" = negh[2],
          "y" = negh[1]
        )
      )
      # Append to the valid_result
      valid_result <- append(valid_result, list(negh_obj))
    }
  }
  return(valid_result)
}
```

Test of the function

```
initial_forest <- initialize_grid(3, 3)
filled_forest <- random_fill(initial_forest)

filled_forest$status
```

```
##      [,1] [,2] [,3]
## [1,]    6    2    1
## [2,]    6    1    1
## [3,]    3    1    4
```

```
# Corner position gives us 3 neighbours
neighbours(filled_forest, c(1, 1))
```

```
## [[1]]
##    state coords.x coords.y
##        6        1        2
##
## [[2]]
##    state coords.x coords.y
##        2        2        1
##
## [[3]]
##    state coords.x coords.y
##        1        2        2
```

Construct the propagate function

Now it's time to define the probabilities of the forest grid cells to ignite eachother.

We are interested in getting the contribution probability of a specific cell to ingite each of its neighbours.

This probability is based on the following aspects:

- Its current state
- Neighboring cell states

Pseudocode

```
propagate is a function with parameters (current_cell, neighbours)
  if current_cell is not Burning
    return 0.0 foreach of the neighbours
  else
    foreach of the neighbours as neg
      if neg is Water set prob to 0
      if neg is DryGrass set prob to 0.125
      if neg is DenseTrees set prob to 0.05
      if neg is Burning set prob to 0
      if neg is NormalForest set prob to 0.0875
      if neg is Burned set prob to 0
    return neighoubrs probabilities
```

R code

```r
# Cell Structure to be given
# neighbour:
#   "state" : val
#   "coords :
#     "x" : val
#     "y" : val

propagate <- function(cell, neighbours) {
  if (cell[["state"]] != 4) {
    return(sapply(neighbours, function(val) 0.0))
  }

  neghs_probs <- sapply(neighbours, function(neg) {
    return(
      switch(neg[["state"]],
        "1" = 0.0,    # max = 0.0 | Water
        "2" = 0.125,  # max = 1.0 | DryGrass
        "3" = 0.05,   # max = 0.4 | DenseTrees
        "4" = 0.0,    # max = 0.0 | Burning
        "5" = 0.0875, # max = 0.7 | NormalForest
        "6" = 0.0     # max = 0.0 | Burned
      )
    )
  })
  return(neghs_probs)
}
```

Test of the function

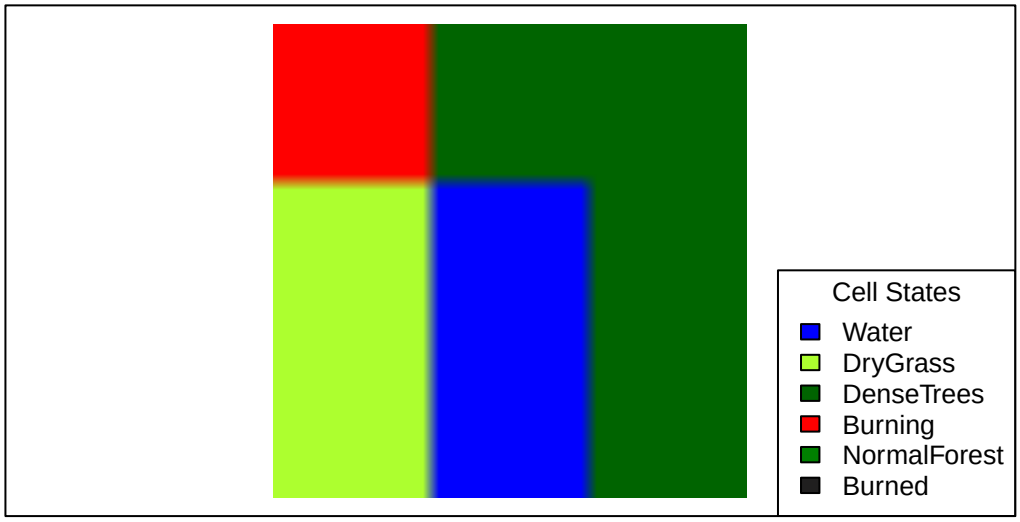
```r
initial_forest <- initialize_grid(3, 3)
filled_forest <- random_fill(initial_forest)
filled_forest$status[1, 1] <- 4
filled_forest$status[1, 2] <- 2
filled_forest$status[2, 2] <- 1
```

```r
filled_forest$status[2, 1] <- 3

neghs <- neighbours(filled_forest, c(1, 1))

current_cell <- c(
  "state" = filled_forest$status[1, 1],
  "coords" = c(
    "x" = 1,
    "y" = 1
  )
)

plot_grid(filled_forest)
```




```r
propagate(current_cell, neghs)
```

```
## [1] 0.050 0.125 0.000
```

The role of different types of vegetation

The probabilities of transfering the fire are based on a maximum value based on the type of the cell receiving the fire.

This is done in a way that if a cell is surrounded by fire cells its probability of ignition will add up to:

- 0.0 if it is Water
- 1.0 if it is DryGrass
- 0.4 if it is DenseTrees

- 0.0 if it is Burning
- 0.7 if it is NormalForest
- 0.0 if it is Burned

So, for example, if a cell of DryGrass is surrouned by Burning cells, it will for sure ignite in the next time step.

Design of the update grid function:

We have now the tools to design the function that will handle the various updates to be performed on the forest grid.

The updates and checks to be applied to each cell are:

- Check if the cell is going to the Burning state
- Check if the cell is going to the Burned state
- Compute the new probability of the cell to become Burning
- Increase the probability of the cell to become Burned

Pseudocode

```
update grid is a function with parameters (forest_grid)
  foreach of the cells in forest grid as cell
    if the cell is DryGrass, DenseTrees or NormalForest
      generate random value between 0.0 and 1.0
      if the value is less or equal to probability of cell
        set cell to Burning
        reset cell probability to 0
    else if cell is Burning
      generate random value between 0.0 and 1.0
      if the value is less or equal to probability of cell
        set cell to Burned
        reset cell probability to 0
      else
        increase probability of cell
    compute probability list of neighbours (propagate)
    foreach neighbour
      add the new computed probability
  return forest_grid
```

R code

```r
update_grid <- function(forest_grid) {
  for (y in 1:nrow(forest_grid$status)) {
    for (x in 1:ncol(forest_grid$status)) {
      # Get the current cell state
      cell_state <- forest_grid$status[y, x]
      cell_prob <- forest_grid$prob[y, x]

      # If the cell could burn, generate a random number
      # and check with its probability
      # if <= , set them to burning and reset their prob
      if (cell_state == 2 || cell_state == 3 || cell_state == 5) {
        rand_value <- runif(1)
        if (rand_value <= cell_prob) {
          forest_grid$status[y, x] <- 4
          forest_grid$prob[y, x] <- 0.0
```

```
        cell_state <- 4
        cell_prob <- 0.0
      }
    } else if (cell_state == 4) {
      # If the cell is burning, and the random value gets the prob
      # set it to burned and reset its prob
      # otherwise increase the prob of going out

      rand_value <- runif(1)
      if (rand_value <= cell_prob) {
        forest_grid$status[y, x] <- 6
        forest_grid$prob[y, x] <- 0.0

        cell_state <- 6
        cell_prob <- 0.0
      } else {
        # Increase the probability of going out
        forest_grid$prob[y, x] <- forest_grid$prob[y, x] + 0.3
        cell_prob <- cell_prob + 0.3
      }
    }

    # Get the neighbours of the current cell
    neghs <- neighbours(
      forest_grid = forest_grid["status"],
      cell = c(y, x)
    )

    # Create an "object" with the data needed from the propagate function
    propagate_cell <- c(
      "state" = cell_state,
      "coords" = c(
        "x" = x,
        "y" = y
      )
    )

    # Compute the probabilities of the propagate of the neighbours cells
    probs_propagate <- propagate(propagate_cell, neghs)

    # Foreach of the neighbours for which we have the probs. of ignition
    # we sum it to the current prob. values of the forest_grid
    # (Markov Chain) t_n = t_n-1 + new_computed_prob
    for (negIndex in 1:length(neghs)) {
      neg_y <- neghs[[negIndex]][["coords.y"]]
      neg_x <- neghs[[negIndex]][["coords.x"]]

      forest_grid$prob[neg_y, neg_x] <-
        forest_grid$prob[neg_y, neg_x] + probs_propagate[[negIndex]]
    }
  }
}
return(forest_grid)
```

```
}
```

Test of the function

```
initial_forest <- initialize_grid(5, 5)
filled_forest <- random_fill(initial_forest)

# Forest just initialized
filled_forest
```

```
## $status
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3    6    3    6    5
## [2,]    3    3    3    3    2
## [3,]    5    5    2    2    5
## [4,]    2    4    6    6    2
## [5,]    6    6    5    4    6
##
## $prob
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

```
filled_forest <- update_grid(filled_forest)

# Forest after one step of update_grid (t + 1)
filled_forest
```

```
## $status
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3    6    3    6    5
## [2,]    3    3    3    3    2
## [3,]    5    5    2    2    5
## [4,]    2    4    6    6    2
## [5,]    6    6    5    4    6
##
## $prob
##        [,1]   [,2]  [,3] [,4]  [,5]
## [1,] 0.0000 0.0000 0.000  0.0 0.000
## [2,] 0.0000 0.0000 0.000  0.0 0.000
## [3,] 0.0875 0.0875 0.125  0.0 0.000
## [4,] 0.1250 0.3000 0.000  0.0 0.125
## [5,] 0.0000 0.0000 0.175  0.3 0.000
```

Adanced grid filling functions

Previously we created a very basic function to fill our forest grid, in this section we will see how to make a more advanced version of it.

The goal is to be able to "draw" a forest based on a generic function.

To give an idea, we want to be able to draw a river (cell type water) with the shape of the sin function in a given span for x and y axis. Or any other "shape" with whichever cell type we want.

Generic fill grid function

Firs we will need a very simple function that will set each cell to a given cell_type, based on the result of a "check function" called pos_func, with a given probability given by a generic "prob_func"

Pseudocode

```
fill_grid is a function with parameters
(forest_grid, pos_func, prob_func, cell_type)
  foreach cell in forest_grid
    if pos_func(y, x) is true
      set the cell status to cell_type
      set the cell prob to the value of prob_func(y,x)
  return forest_grid
```

In the R code we added some customizations and other parameters to give more flexibility on the shape produced, like:

- Flip the orientation
- Thickness of the "lines" / Precision of the pos_func

R code

```r
fill_grid <- function(
    forest_grid, pos_func, prob_func,
    cell_type, thickness = 0.5, flip_orientation = FALSE
) {
  max_y <- dim(forest_grid$status)[1]
  max_x <- dim(forest_grid$status)[2]

  for (y in 1:max_y) {
    for (x in 1:max_x) {
      if (pos_func(y, x, max_y, thickness, flip_orientation)) {
        forest_grid$status[y, x] <- ifelse(
          cell_type == -1,
          forest_grid$status[y, x],
          cell_type
        )
        forest_grid$prob[y, x] <- prob_func(y, x)
      }
    }
  }
  return(forest_grid)
}
```

Generic function to grid coordinates

We cannot test the fill_grid function, unless we create some functions to pass to it.

For this we will create a generic function that will map a given function (like sin, cos, exp, ecc.) to our grid coordinates.

It will check if by overlapping the plot of the given function, the coordinates would end up near the values produced by the function.

Pseudocode

```
func_fill is a function with parameters (y, x, func_span, func, thickness)
  scale x to the func_span
  scale y to the func_span
```

```
  get y_func by calling func with x
  return if y_scaled is near y_func (error < thickness)
```

Values in R code

- y, x are the coordinates of the point to be tested,
- min_y, max_y, max_x will be used to handle the ratio/span of the function in the grid
- func is the function used that will produce the value to compare with the thickness
- thickness is the sensitivity/distance from the y,x coordinate and the actual function value
- Returns true or false if the coordinate is or not generated by the function

R Code

```r
func_fill <- function(y, x, min_y, max_y, max_x, func, thickness) {
  # Scale y to the range of the sine function.
  # Put the 0 of the sin in the middle of my matrix
  y_scaled <- ((y - min_y) / (max_y - min_y)) * 2 - 1

  # Scale x to the range of the sine function.
  x_scaled <- (x - max_x / 2) / (max_x / 2)

  # Calculate the y value of the sine wave at position x
  y_func <- func(x_scaled)

  # Check if the scaled y value is close to the y value of the sine wave
  return(abs(y_scaled - y_func) < thickness)
}
```

The func_fill will serve as an "interface" for all the position_functions that we may want to create.

Here are the 3 position_functions that we created to generate some various scenarios.

Position functions

Random variables for function span To produce each time random values/versions of the position_function we will need two random values, that will vary at each simulation.

These two random variables will handle the span that we are looking into for each of the position_functions.

R code

```r
rand_max_x <- pi * sample(10:50, 1)
rand_min_y <- sample(2:20, 1)
```

Here is the definitions of random_position functions that we created.

The way we create these is by "wrapping" the func_fill generic function with some predifined or randomly generated values, and with a specific func.

Random position using runif() function R Code

```r
# It will produce a random pattern
rand_pos <- function(y, x, max_y, thickness, flip_orientation = FALSE) {
  if (flip_orientation) {
    # Randomly change the function orientation
    temp <- y
    y <- x
    x <- temp
```

```
  }
  return(
    func_fill(
      y = y,
      x = x,
      min_y = 0,
      max_y = max_y,
      max_x = pi,
      func = function(val) runif(1, -max_y, max_y),
      thickness = thickness # 0.2
    )
  )
}
```

```
# It will produce a exp_lien function (straight river like)
exp_pos <- function(y, x, max_y, thickness, flip_orientation = FALSE) {
  if (flip_orientation) {
    # Randomly change the function orientation
    temp <- y
    y <- x
    x <- temp
  }
  return(
    func_fill(
      y = y,
      x = x,
      min_y = max_y * rand_min_y,
      max_y = max_y,
      max_x = rand_max_x,
      func = exp,
      thickness = thickness # 0.3
    )
  )
}
```

Random position using Exponential function

```
# It will produce a sin_wave function (wavy river like)
sin_pos <- function(y, x, max_y, thickness, flip_orientation = FALSE) {
  if (flip_orientation) {
    # Randomly change the function orientation
    temp <- y
    y <- x
    x <- temp
  }
  return(
    func_fill(
      y = y,
      x = x,
      min_y = 0,
      max_y = max_y,
```

```
      max_x = rand_max_x,
      func = sin,
      thickness = thickness # 0.3
    )
  )
}
```

Random position using Sin function

Simulate function

The last piece of our simulation machine is the functio that will orchestrate every functionality that we developed until now.

We are talking about the simulate function.

This function will generate a certain amount of simulations (forest_grids), for each of them it will step through the simulation a specific amount of times.

In the end we will have run multiple simulations, based on randomly generated scenarios (forest_grids)

Pseudocode

```
simulate is a function with parameters (t_max, n_sims, grid_rows, grid_cols)
  for n_sims times
    intialize a new forest_grid with dims grid_rows * grid_cols
    randomly fill the forest_grid using random position_functions
    set a random starting Burning cell
    for t_max times
      update the grid state
```

In the R code implementation there are some aspects that needed more attention than the pseudocode.

- Randomize the position_functions values
- Choose random scenarios and their values
- Handle the production of a .gif or .mp4 file as result

R code

```
simulate <- function(t_max, N, grid_rows, grid_cols, debug = FALSE) {
  # N simulations, until t_max iterations

  for (i in 1:N) {
    if(debug) {paste("Simulation", i)}

    # Initialize the grid full of normal forest
    forest_grid <- initialize_grid(grid_rows, grid_cols)

    # Add random dry grass with some small self ignition probability
    forest_grid <- fill_grid(
      forest_grid = forest_grid,
      pos_func = rand_pos,
      prob_func = function(y, x) 0.0002, # self-ignition prob
      cell_type = 2, # Dry grass
      thickness = runif(1, 0, 20) # Higher value = Higher population
    )

    # Add random dense trees with some small self ignition probability
```

```r
forest_grid <- fill_grid(
  forest_grid = forest_grid,
  pos_func = rand_pos,
  prob_func = function(y, x) 0.00002, # self-ignition prob
  cell_type = 3, # Dense trees
  thickness = runif(1, 0, 20) # Higher value = Higher population
)

#  Randomize the values of the exp_pos function
rand_max_x <- pi * sample(10:50, 1)
rand_min_y <- sample(2:20, 1)

# Adding a river with a random function plotted (exp or sin)
# I kept them separated and not param only
# The func_pos to be able to modify them separately
rand_value_river <- runif(1)
water_type <- 0
if (rand_value_river < 0.33) {
  # Exp function
  forest_grid <- fill_grid(
    forest_grid,
    exp_pos,
    function(y, x) 0.0,
    1,
    runif(1, 0.05, 0.3),
    ifelse(runif(1) < 0.5, TRUE, FALSE)
  )
  water_type <- 0
} else if (rand_value_river >= 0.33 && rand_value_river < 0.66) {
  # Random sin river
  forest_grid <- fill_grid(
    forest_grid,
    sin_pos,
    function(y, x) 0.0,
    1,
    runif(1, 0.05, 0.3),
    ifelse(runif(1) < 0.5, TRUE, FALSE)
  )
  water_type <- 1
} else {
  # Random rain puddles
  forest_grid <- fill_grid(
    forest_grid,
    rand_pos,
    function(y, x) 0.0,
    1,
    runif(1, 0.05, 10),
    ifelse(runif(1) < 0.5, TRUE, FALSE)
  )
  water_type <- 2
}

# Random starting point burning
```

```
    forest_grid$status[sample(1:grid_rows, 1), sample(1:grid_cols, 1)] <- 4

    # Define the animation function

    for (t in 1:t_max) {
      # Print the current iteration
      # paste("Iteration", t)
      # Update the grid
      forest_grid <- update_grid(forest_grid)

      # Plot the grid
      plot_grid(
        forest_grid,
        paste(
          "Simulation n. ", i, " / ", N, "\nStep n. ", t, " / ", t_max,
          "\n Water type: ",
          ifelse(
            water_type == 0, "Exp River",
            ifelse(water_type == 1, "Sin River", "Rain puddles")
          )
        )
      )
      if(debug) {print(paste("T: ", t, " / ", t_max))}
    }
    if(debug) {print(paste(i, "| Simulation Completed"))}
  }
}
```

Run the simulation

Produce a gif

Change to {R} code to execute

```
# GIF
saveGIF(
 simulate(
   t_max = 50,
   N = 3,
   grid_rows = 100,
   grid_cols = 100,
   debug = TRUE
 ),
 movie.name = paste("forest_simulation.gif"),
 interval = 0.2
)
```

Produce a mp4

Save the result in a MP4

Change to {R} code to execute

```
# MP4
saveVideo(
 simulate(
```

```
   t_max = 30,
   N = 3,
   grid_rows = 50,
   grid_cols = 50,
   debug = TRUE
 ),
 video.name = paste("forest_simulation.mp4"),
 interval = 0.2
)
```

Analysis of update_grid function

Random variables

In the implementation of the update_grid function we used 2 random variables:

The first one is a discrete random variable, and it is responsible of the change of state of a cell to a Burning state.

It is a Bernoulli random variable which will define the next state of the cell.

Let S be a Bernoulli random variable with parameter p_i, where p_i is the probability of ignition for cell i.

Then the probability mass function of S is given by:

$$P(S = k) = \begin{cases} p_i & \text{if } k = 1, \text{ i.e., the cell changes to a Burning state,} \\ 1 - p_i & \text{if } k = 0, \text{ i.e., the cell does not change to a Burning state.} \end{cases}$$

During the execution of the *update_grid* function, for each cell i, we generate a random value and compare it with p_i to determine the next state of the cell.

Each cells probability of ignition is assigned at grid initialization and updated during the *update_grid* execution.

The second one is the same as the previous one, but it handles the change to Burned state of a Burning cell.

In the first case, the *to Burning* step, would not have any effects on the probabilities if the result was k=0.

In the *to Burned* process a result of k=0 would result in the increase of the probability of the cell by a predefined amount, 0.3 in our implementation.

Random number generators

The values to be compared to the probabilities in the previous section need to be somehow generated at runtime.

In our implementation we use the R integrated function *runif()* to generate a random continuous value from a uniform distribution between 0 and 1, or between our desired range of values.

In other parts of the implementation, like in the simulation function, we used the R integrated function *sample()* to produce a random discrete value from a choosen set of values.

Stochastic process

Just by adding these 2 random variables, and by generating the initial grid also randomly, we are able to add stochasticity to our model.

In this case the stochastic part is the random evolution of the cells to the Burning or Burned state.

If we change the thresholds or how we generate this probabilities, we are able to achieve different scenarios.

In our general case we have that the fire spreads at a certain "speed", and it will not that easily burn out before spreading to the whole forest grid. But if we would increase the *to Burned* transition probability we might get a simulation were the fire burns out immediately.

This is all to say that with the addition of those random variables, and the probabilities chosen we will always have a different result for each simulation run.

Markov Chain characteristics

We already cited before in the implementation that our model structure can be seen as a Markov Chain.

In this case each of the grid cells has its own Markov Chain development, this means that we have m*n markov chains if m and n are the rows and cols of the grid.

The states of the Markov Chain are the types that the cell can assume, but they do not directly correspond to our definition of cell states, but to a subset of those.

In particular the states of the Markov Chains are:

- Not Burning
- Burning
- Burned

In which the Not Burning corresponds to any of starting types. Each of the cells can then, during the update_grid execution, change their state from *Not Burning* to *Burning* to *Burned*.

We need to specify that these transformations are one-way, that means that after you "evolve" to the next state, you have no way of coming back.

At least, this is how we decided to implement the model, a possibility would be to make a *Burned* cell have a probability of *Reignition* i.e. to go back to the *Burning* state.

We can also see an interesting aspect, in this type of simulation, the different Markov Chain interact with each other, as the state of one Markov Chain will affect the neighbours Markov Chains.