The Schelling Model in R

Andrew S. Rosenberg

6/12/2017

The Code

Below, I include a function that replicates (ish) the Schelling (1971) model of segregation. The basic set-up is simple. I generate a matrix of 1's and 0's with given (square) dimensions. The 1's and 0's are generated initially with equal probability—I vary this parameter below. Then, in the spirit of the Schelling model, a random unit is selected, if fewer than half of its neighbors have the same value, then the unit switches. The function can be replicated many times to show how patterns of segregation emerege quickly with this simple rule.

Prior to presenting the main segregation function, I present two helper functions that are used within it. I created them separately because it is easier for me to visualize them as distinct parts of the function. They could be included, but I think it makes the final function too long and cluttered. This first block sets up a way to easily generate the row/column positions of a given unit's neighbors.

```
position_adjustments <- c(0, 1, -1)
positions <- gtools::permutations(3, 2, v = position_adjustments,
  repeats.allowed = TRUE)
positions <- positions[-5, ]
```

The next block is a simple function that sums the value of the neighbors for any given unit. I made sure to add an "if" statement that takes care of instances when the unit is on the border. There is more about the border annoyance below.

```
sum_of_neighbor_values <- function(mat = neighbors, dimens = dims)
{
  vals <- sapply(1:8, function(i)
   {
    row_idx <- mat[i, 1]
    col_idx <- mat[i, 2]
    if(row_idx %in% seq(1, dimens, 1) & col_idx %in% seq(1, dimens, 1))
    {
      world[row_idx, col_idx]
    }
   }
  )
  sum(unlist(vals), na.rm = TRUE)
}
```

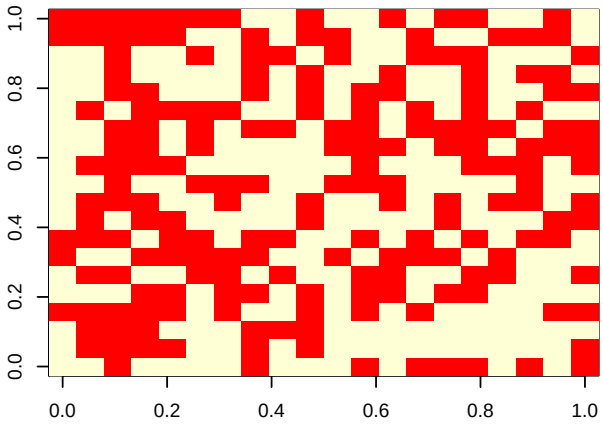
Finally, here is the main segregation function. It takes two arguments, the starting "world" or initial matrix of 1's and 0's, and the dimension of that matrix. As you can see, it isn't particularly elegant because it requires a bunch of "if else" statements to implement various rules governing corner and border units. Because corner solutions are a subset of border solutions, deal with them first. If a unit is on the corner, then it can only have a maximum of 3 neighbors. Therefore, the decision rule must be adjusted. The same logic applies for normal border units which only have 5 neighbors. The code is pretty ugly and I could have subsumed this logic into one statement (sum same <= 0.5 x number of neighbors, or something), but I ran out of time (and that is kind of annoying to program).

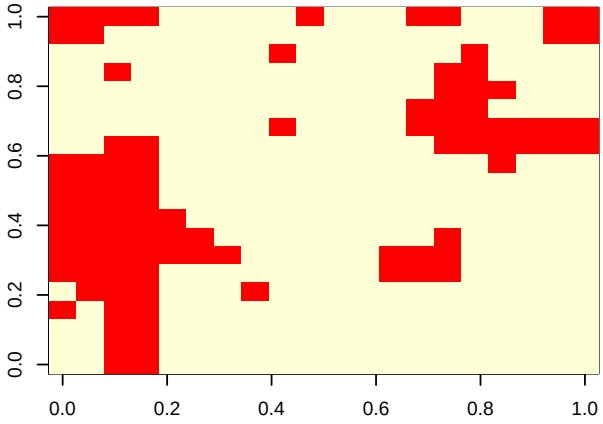
```
segregation_function <- function(world, dims)
{
```

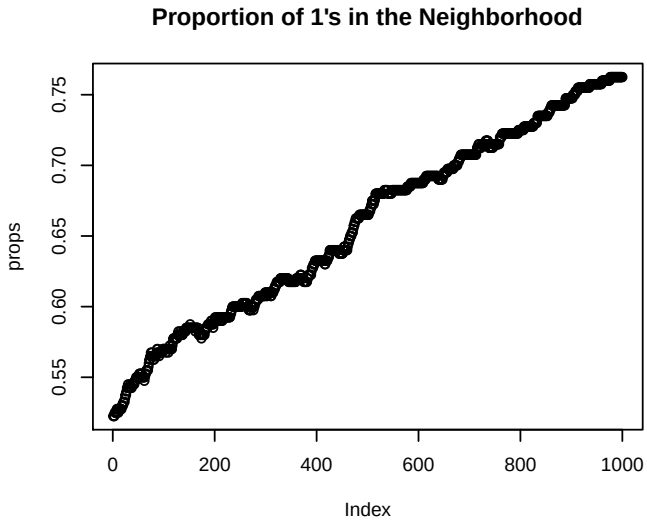
```
  world_matrix <- world
  row_pos <- sample(seq(1, dims, 1), 1, replace = TRUE)
  col_pos <- sample(seq(1, dims, 1), 1, replace = TRUE)
  selected_unit <- world_matrix[row_pos, col_pos]
  neighbors <- cbind(row_pos + positions[, 1], col_pos + positions[, 2])
  sum_of_neighbors <- sum_of_neighbor_values(mat = neighbors, dimens = dims)
  if(row_pos %in% c(1, dims) | col_pos %in% c(1, dims))
  {
    if(row_pos == col_pos | row_pos == 1 & col_pos == dims |
      row_pos == dims & col_pos == 1)
    {
      if(selected_unit == 1 & sum_of_neighbors < 2)
      {
        world_matrix[row_pos, col_pos] <- 0
      }
      if(selected_unit == 0 & sum_of_neighbors >= 2)
      {
        world_matrix[row_pos, col_pos] <- 1
      }
    }
    if(selected_unit == 1 & sum_of_neighbors < 3)
    {
      world_matrix[row_pos, col_pos] <- 0
    }
    if(selected_unit == 0 & sum_of_neighbors >= 3)
    {
      world_matrix[row_pos, col_pos] <- 1
    }
  }
  else{
    if(selected_unit == 1 & sum_of_neighbors < 4)
    {
      world_matrix[row_pos, col_pos] <- 0
    }
    if(selected_unit == 0 & sum_of_neighbors >= 4)
    {
      world_matrix[row_pos, col_pos] <- 1
    }
  }
  world <<- world_matrix
  sum(world) / (dims * dims)
}
```

Basic Results

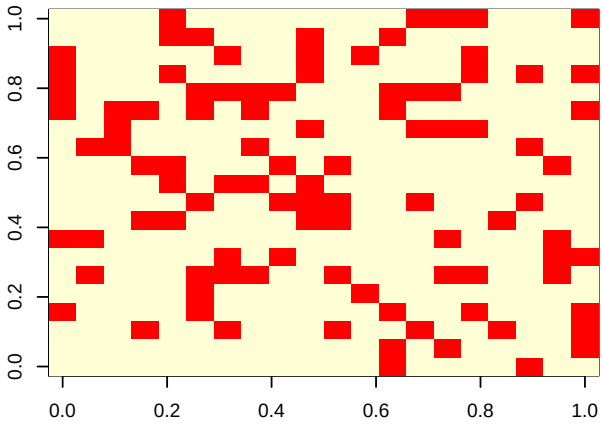
Here, I present three results. First, I run the function 1000 times with a 20x20 matrix. 1's and 0's have equal probability of occuring in the initial "world."

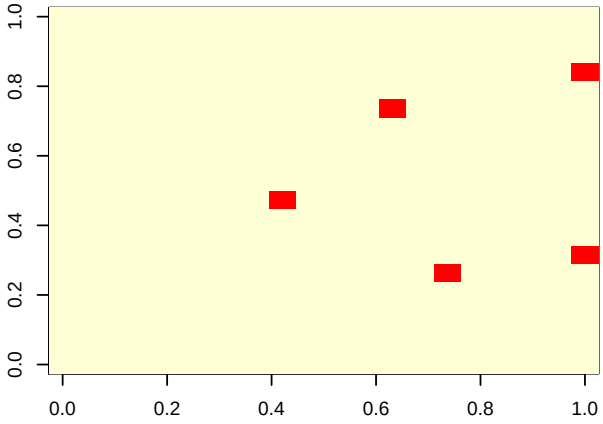

0.0
0.2
0.4
0.6
0.8
1.0

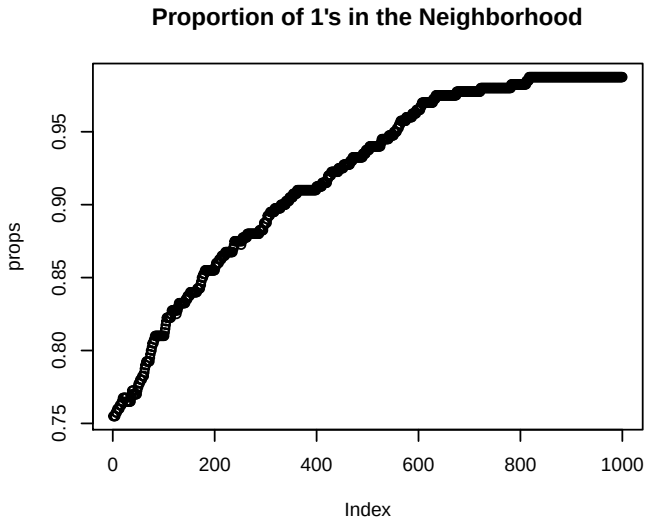

0.0
0.2
0.4
0.6
0.8
1.0


Proportion of 1's in the Neighborhood
props
0.55
0.60
0.65
0.70
0.75
0
200
400
600
800
1000
Index

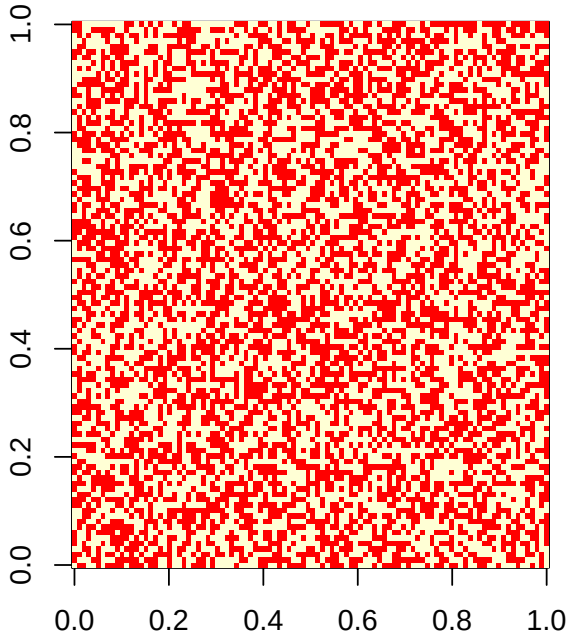
As one can see, the pattern of segregation emerges. In addition, I included a plot of how the proportion of 1's in the matrix changes over the course of the iterations. This plot presents an additional dimension for visualizing the pace of segregation. Next, using the same setup, I vary the probability of 1's and 0's. In this example, 1's have a 0.75 probability of occuring in the initial "world," while 0's have a 0.25 probability. Unsurprisingly, the 1's (white color) crowd out all of the 0's after 1000 iterations. NB: this result holds in a more general case: the greater the number of simulations, the more likely that one color crowds out another. However, I omit this result for brevity (it is obvious that the 50/50 case takes the longest for crowding out to occur). As an aside, I omit a discussion of how the demand for neighbors affects the onset of segregation. To test this proposition, one needs only to change the thresholds in the "if else"" statements in the function. I considered including this "demand"" as a parameter in the function to make it more general, but I ran out of time (just as I did above). Though, it shouldn't be too difficult.


0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0


Proportion of 1's in the Neighborhood
props
0.75
0.80
0.85
0.90
0.95
0
200
400
600
800
1000
Index

Finally, I expand the initial matrix to see whether these patterns emerge when the "neighborhood" is dramatically expanded. To do so, I use a 100x100 matrix. I run the 1000, 5000, and 10,000 simulations to show how segregation appears. In the panels below, the figure on the left always represents the initial random matrix. The figure on the right is how the initial matrix appears after the given number of iterations. While it may take an order of magnitude more iterations for segregation to appear, similar patters clearly emerge.



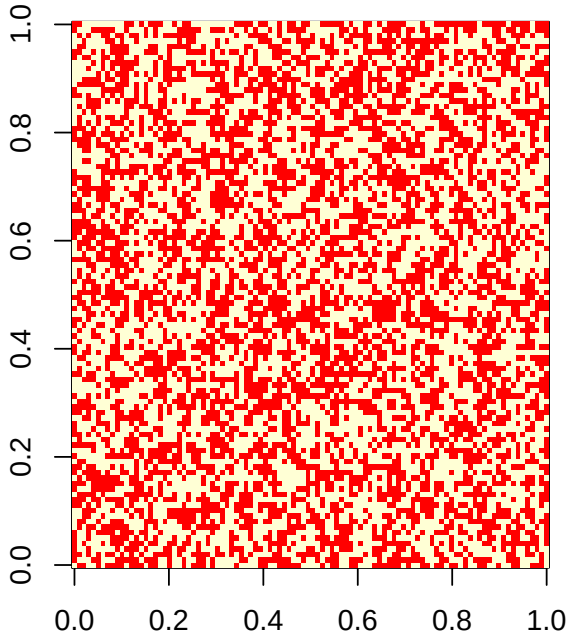
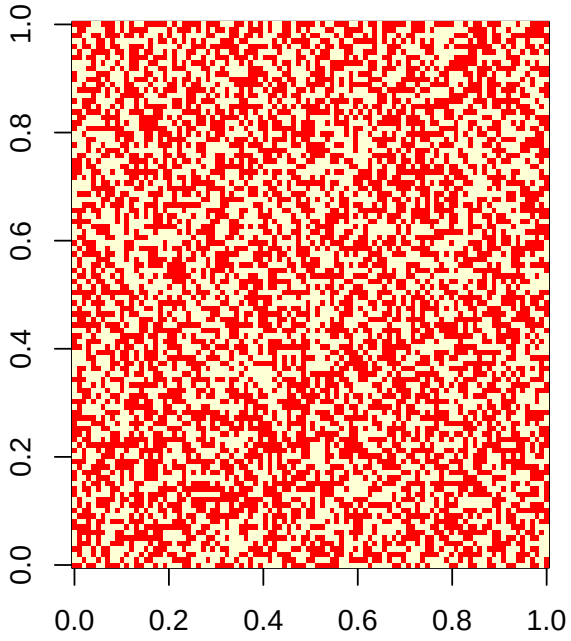


Figure 1: 1000 iterations



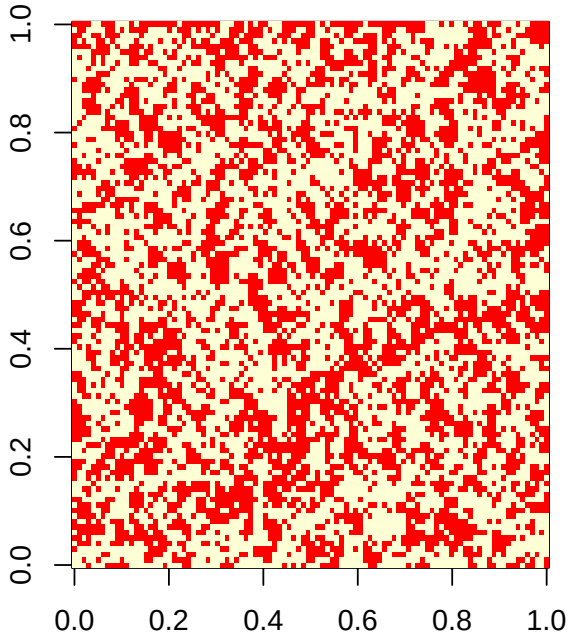


Figure 2: 5000 iterations

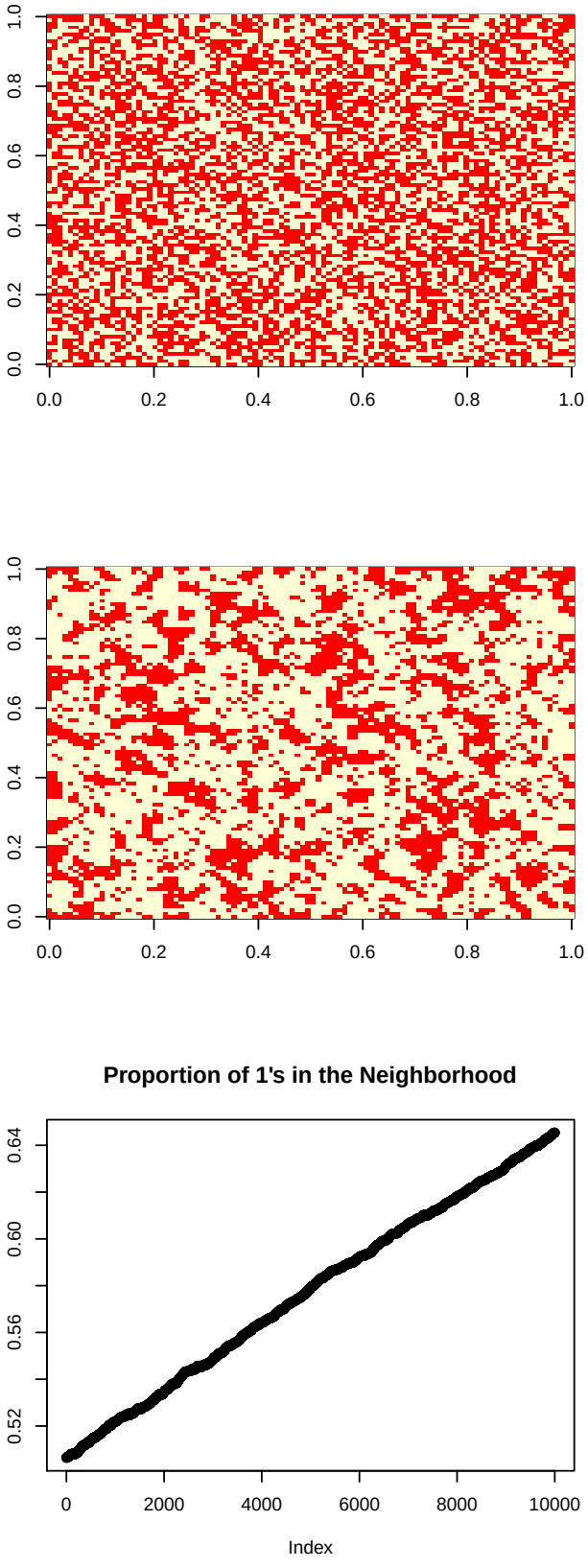


Figure 3: 10,000 iterations