

Spreading of Fire

File: *Fire.mw*

Based on "Contagion in Excitable Media" from *Modeling Nature: Cellular Automata Simulations with Mathematica* by Richard J. Gaylord and Kazume Nishidate, Copyright 1996 TELOS/Springer-Verlag, pp. 155-171.

[-] Development of Fire Program

[-] Lattice (matrix, grid)

Grid-site values:

EMPTY (0) - empty
TREE (1) - non-burning tree
BURNING (2) - burning tree

probTree = probability of grid site occupied by tree (value 1); i.e., tree density
probBurning = probability that a tree is burning (value 2); i.e., fraction of burning trees
probImmune = probability of immunity from catching fire - global variable
probLightning = probability of lightning - global variable

```
> # constants
EMPTY := 0:
TREE := 1:
BURNING := 2:
```

[-] Update rules: Rules to compute next site value

At next time step tree grows in empty cell

```
> undefine(spread):
define(spread,
  spread(EMPTY, N::integer, E::integer,
    S::integer, W::integer) =
  EMPTY
);
```

Burning tree results in empty cell at next time step

```
> definemore(spread,
  spread(BURNING, N::integer, E::integer,
    S::integer, W::integer) =
  EMPTY
);
```

Perhaps next time step tree with burning neighbor(s) burns itself.

Function *rand0to1()* and variable *probImmune* must be undefined for rule definitions but must be given values before rules are used.

```
> unassign(rand0to1):
probImmune := 'probImmune':

definemore(spread,
  spread(TREE, BURNING, E::integer, S::integer,
```

```

        W::integer) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING),

    spread(TREE, N::integer, BURNING, S::integer,
        W::integer) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING),

    spread(TREE, N::integer, E::integer, BURNING,
        W::integer) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING),

    spread(TREE, N::integer, E::integer,
        S::integer, BURNING) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING)
);

```

Perhaps tree is hit by lightning and burns next time step.

Function *rand0to1()* and variables *probImmune* and *probLightning* must be undefined for rule definitions but must be given values before rule is used.

```

> unassign(rand0to1):
    probImmune := 'probImmune':
    probLightning := 'probLightning':

    definemore(spread,
        spread(TREE, N::integer, E::integer,
            S::integer, W::integer) =
            `if`(rand0to1() < probLightning *
                (1 - probImmune),
                BURNING, TREE)
    );

```

▣ Boundaries

Periodic boundary conditions in this case

Infinite system

Extend boundary by 1 for sniff and by 2 for walk

Function to return an $(n + 2)$ -by- $(n + 2)$ matrix for periodic boundaries *mat*, *n*-by-*n* matrix

```

> unassign(extendLat):
    extendLat := proc(mat)
        local matNS, trans, transEW;

        # glue on wrap of N & S rows
        matNS := [mat[-1], op(mat), mat[1]]:

        # glue on wrap of E & W columns assuming
        # original matrix is square
        trans := ListTools[Transpose](matNS):
        transEW := [trans[-1], op(trans), trans[1]]:

```

```
ListTools[Transpose](transEW);
end proc;
```

[-] Function to apply a function parameter to extended matrix

Function to apply a function parameter to a matrix extended by 1 cell in each direction

Function returns a matrix of applications of function *fnc*, called as *fnc(site, N, E, S, W)*, to each element (*site*) of extended matrix *matExt* except for the first and last rows and columns

```
> applyExtended := proc(fnc, matExt)
  local n;
  n := nops(matExt) - 2;
  [seq(
    [seq( fnc(matExt[i, j], matExt[i - 1, j],
             matExt[i, j + 1], matExt[i + 1, j],
             matExt[i, j - 1]), j = 2..(n + 1) )
    ],
    i = 2..(n + 1) )
];
end proc;
```

[-] Fire simulation

Function *rand0to1()* returns a random floating point number between 0 and 1. Function *fire* and *spread* rules use this function.

Function *fire* drives the simulation and returns a list of grid "matrices".

Grid-site values:

EMPTY (0) - empty

TREE (1) - non-burning tree

BURNING (2) - burning tree

Variables *probLightning* and *probImmune* must global for *spread* rules.

n = number of cells in each direction

probTree = probability of grid site occupied by tree (value 1); i.e., tree density

probBurning = probability that a tree is burning (value 2); i.e., fraction of burning trees

chanceImmune (globally *probImmune*) = probability of immunity from catching fire

chanceLightning (globally *probLightning*) = probability of lightning

t = number of time steps in simulation

```
> unassign(fire, rand0to1):
  rand0to1 := proc()
    stats[random, uniform]();
  end proc;

  fire := proc(n, probTree, probBurning,
              chanceLightning, chanceImmune, t)
    local forest, forestExtended, grids, i, j;
    global probLightning, probImmune;

    probLightning := chanceLightning;
    probImmune := chanceImmune;
```

```

### Initialize grid ###
forest := [seq([seq(
  `if`(rand0tol() < probbTree,
    `if`(rand0tol() < probbBurning,
      BURNING, TREE),
    EMPTY),
  j = 1..n)], i = 1..n)];

### Perform simulation ###
grids := [forest]:
for i from 1 to t do
  # Extend matrix
  forestExtended := extendLat(forest):

  # Apply spread of fire function to each
  # grid point
  forest := applyExtended(spread,
    forestExtended):

  # Save new matrix
  grids := [op(grids), forest];
end do;

grids;
end proc:

```

[-] Animation of simulation

Function to display a list of grids, *graphList*, with cell values colored as follows: *EMPTY* --> yellow, *TREE* --> green, *BURNING* --> burnt orange

Note that the sequence of squares for a grid are grouped into a display with *plots[display](plotGrid)* to be appended to the end of the list of grid plots, *plotList*.

```

> unassign(matchColor):
# Function to associate color with cell's value
matchColor := (mat, i, j)->eval(mat[i, j],
  [ EMPTY = yellow,
    TREE = COLOR(RGB, 0.1, 0.75, 0.2),
    BURNING = COLOR(RGB, 0.6, 0.2, 0.1)
  ]):

unassign(showGraphs):
# Function to display a list of grids starting
# at 1 through entire list of grids
# Empty (0) shows yellow; tree (1) shows green;
# burning tree (2) shows burnt orange.
showGraphs := proc(graphList)
  local k, g, aSquare, plotGrid, plotList, n;

  plotList := []:
  aSquare := [[0, 0], [0, 1],[1, 1],[1, 0]]:
  for k from 1 to nops(graphList) do
    g := graphList[k]: # grid at k-th time step
    n := nops(g):

```

```

    plotGrid := seq(seq(
      plottools[ translate](
        plots[polygonplot](aSquare,
          axes = none,
          scaling = CONSTRAINED,
          color = matchColor(g, i, j)),
        j, i),
      j = 1..n), i = 1..n):
    plotList := [op(plotList),
      plots[display](plotGrid)]:
  end do;

  if (nops(plotList) > 0) then
    plots[display](plotList,
      insequence = true);
  else
    plots[display](plotList);
  end if;
end proc:

```

[-] Run Program

```
[fire(n, probTree, probBurning, chanceLightning, chanceImmune, t)
```

```
[Initialization with n = 20, probTree = 0.5, probBurning = 0.5
```

```
[> fireList := fire(20, 0.5, 0.5, 0.5, 0.5, 3):
```

```
[> showGraphs(fireList);
```

```
[chanceLightning = 0
```

```
[chanceImmune = 0
```

```
[> fireList := fire(20, 0.3, 0.001, 0, 0, 15):
```

```
[> showGraphs(fireList);
```

```
[chanceLightning small
```

```
[chanceImmune = 0
```

```
[> fireList := fire(20, 0.3, 0, 0.00025, 0, 20):
```

```
[> showGraphs(fireList);
```

```
[chanceLightning = 0
```

```
[chanceImmune = 0.52
```

```
[> fireList := fire(20, 0.3, 0, 0, 0.52, 25):
```

```
[> showGraphs(fireList);
```

[-] Fire Functions in One Cell

```
> unassign(rand0to1, extendLat, applyExtended, fire, matchColor,
showGraphs):
```

```
probImmune := 'probImmune':
```

```
probLightning := 'probLightning':
```

```
probImmune := 'probImmune':
```

```

# constants
EMPTY := 0:
TREE := 1:
BURNING := 2:

undefine(spread):
# At next time step tree grows in empty cell
define(spread,
    spread(EMPTY, N::integer, E::integer,
        S::integer, W::integer) =
        EMPTY
    );

# Burning tree results in empty cell at next time
# step
definemore(spread,
    spread(BURNING, N::integer, E::integer,
        S::integer, W::integer) =
        EMPTY
    );

# Perhaps next time step tree with burning
# neighbor(s) burns itself.
# Function rand0to1() and variable probImmune must
# be undefined for rule definitions but must be
# given values before rules are used.

unassign(rand0to1):
probImmune := 'probImmune':

definemore(spread,

    spread(TREE, BURNING, E::integer, S::integer,
        W::integer) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING),

    spread(TREE, N::integer, BURNING, S::integer,
        W::integer) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING),

    spread(TREE, N::integer, E::integer, BURNING,
        W::integer) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING),

    spread(TREE, N::integer, E::integer,
        S::integer, BURNING) =
        `if`(rand0to1() < probImmune,
            TREE, BURNING)
    );

# Perhaps tree is hit by lightning and burns next

```

```

# time step.
# Function rand0tol() and variables probImmune and
# probLightning must be undefined for rule
# definitions but must be given values before rule
# is used.

```

```

unassign(rand0tol):
probImmune := 'probImmune':
probLightning := 'probLightning':

```

```

definemore(spread,
  spread(TREE, N::integer, E::integer,
    S::integer, W::integer) =
  `if`(rand0tol() < probLightning *
    (1 - probImmune),
    BURNING, TREE)
);

```

```

# Function to return an (n + 2)-by-(n + 2) matrix
# for periodic boundaries mat, n-by-n matrix

```

```

unassign(extendLat):
extendLat := proc(mat)
  local matNS, trans, transEW;

  # glue on wrap of N & S rows
  matNS := [mat[-1], op(mat), mat[1]];

  # glue on wrap of E & W columns assuming
  # original matrix is square
  trans := ListTools[Transpose](matNS):
  transEW := [trans[-1], op(trans), trans[1]]:
  ListTools[Transpose](transEW);
end proc:

```

```

# Function to apply a function parameter to a
# matrix extended by 1 cell in each direction.
# Function returns a matrix of applications of
# function fnc, called as fnc(site, N, E, S, W),
# to each element (site) of extended matrix matExt
# except for the first and last rows and columns

```

```

applyExtended := proc(fnc, matExt)
  local n;
  n := nops(matExt) - 2:
  [seq(
    [seq( fnc(matExt[i, j], matExt[i - 1, j],
      matExt[i, j + 1], matExt[i + 1, j],
      matExt[i, j - 1]), j = 2..(n + 1) )
    ],
    i = 2..(n + 1) )
];
end proc:

```



```

        # Save new matrix
        grids := [op(grids), forest];
    end do;

    grids;
end proc:

# Function to associate color with cell's value
matchColor := (mat, i, j)->eval(mat[i, j],
    [ EMPTY = yellow,
      TREE = COLOR(RGB, 0.1, 0.75, 0.2),
      BURNING = COLOR(RGB, 0.6, 0.2, 0.1)
    ]):

# Function to display a list of grids starting
# at 1 through entire list of grids
# Empty (0) shows yellow; tree (1) shows green;
# burning tree (2) shows burnt orange.
showGraphs := proc(graphList)
    local k, g, aSquare, plotGrid, plotList, n;

    plotList := [];
    aSquare := [[0, 0], [0, 1],[1, 1],[1, 0]];
    for k from 1 to nops(graphList) do
        g := graphList[k]: # grid at k-th time step
        n := nops(g):
        plotGrid := seq(seq(
            plottools[ translate](
                plots[polygonplot](aSquare,
                    axes = none,
                    scaling = CONSTRAINED,
                    color = matchColor(g, i, j)),
                j, i),
            j = 1..n), i = 1..n):
        plotList := [op(plotList),
            plots[display](plotGrid)];
    end do;

    plots[display](plotList, insequence = true);
end proc:

```

>

