11.2 Spreading of Fire

Maple Quick Review Questions

Introduction to Computational Science: Modeling and Simulation for the Sciences
Angela B. Shiflet and George W. Shiflet
Wofford College

This file contains system-dependent Quick Review Questions and answers in *Maple* for Module 11.2 on "Spreading of Fire." Complete all code development in *Maple*.

Initializing the System

Quick Review Question 1 Suppose the *fire* function begins as follows:

```
fire := proc(n, probTree, probBurning,
             chanceLightning, chanceImmune, t)
```

Part a assigns values to global variables, and Parts b and c initialize *forest* to be an *n*-by-*n* matrix and *grids* to be a list containing *forest*. Quick Review Question 9 completes the implementation of the function *fire*.

a. Each site is assigned a value as described in the algorithm for Cell Initialization. Complete the code to make the variables on the first line local, the variables on the second line global, and to assign values to the latter.

```
_____________ forest, forestExtended, grids, i, j;
_____________ probLightning, probImmune;

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

b. Complete the code to initialize *forest* to be an *n*-by-*n* matrix (list of *n* lists of *n* elements each). Each cell is assigned a value as described in the algorithm for Cell Initialization. Assume the function call *rand0to1*() returns a random floating point number between 0 and 1.

```
forest := [_____________ ([_____________ (
      `if`(_____________ _____________ probTree,
          `if`(rand0to1() < probBurning,
              _________________, _________________),
          _________________),
      j = 1..n)], i = 1..n)];
```

c. Give command to establish *grids* as a list containing *forest*. At the end of the simulation, *grids* holds all the simulation grids.

Updating Rules

Quick Review Question 2 The following questions develop the rule for *spread*(*site*, *N*, *E*, *S*, *W*) that applies to the situation where a site does not contain a tree at this or any time step:

- **a.** Select the value of *site: EMPTY*, *TREE*, *BURNING*, none of these
- **b.** The values at the neighboring sites–*N*, *E*, *S*, and *W*–are irrelevant but must have types. Give the *N* parameter along with its type declaration.
- **c.** Select the return value: *EMPTY*, *TREE*, *BURNING*, none of these
- **d.** Give the *Maple* function that generates a definition.
- **e.** Write the entire rule.

Quick Review Question 3 The following questions develop the rule for *spread*(*site*, *N*, *E*, *S*, *W*) that applies to the situation where a site contains a burning tree:

- **a.** Select the value of *site: EMPTY*, *TREE*, *BURNING*, none of these
- **b.** The values at the neighboring sites–*N*, *E*, *S*, and *W*–are irrelevant but must have types. Give the *S* parameter along with its type declaration.
- **c.** Because a burning tree always burns down, give the return value of the *spread* function for this situation.
- **d.** Give the *Maple* function that generates an additional definition.
- **e.** Write the entire rule.

Quick Review Question 4 The questions below develop the rule for *spread*(*site*, *N*, *E*, *S*, *W*) that applies to the situation where a site contains a non-burning tree that may catch fire because a neighboring site contains a burning tree. This and several other rules use the procedure call *rand0to1*(), which is to return a random floating point number between 0 and 1. We do not employ *stats*[*random*, *uniform*]() in the *spread* rule definition, because if we did, *Maple* would immediately call the uniform random number generator and the rule would use one particular generated number throughout execution. Thus, before the *spread* definitions, we unassign *rand0to1*, which we define later as *stats*[*random*, *uniform*]().

- **a.** Select the value of *site: EMPTY*, *TREE*, *BURNING*, none of these
- **b.** Select the meaning of the following call to *if*.

```
`if`(rand0to1() < probImmune, TREE, BURNING)
```

  A. If a random number is less than the probability of immunity, then the tree catches fire; else it does not.
  B. If a random number is less than the probability of immunity, then the tree does not catch fire; else it does.
  C. If a random number is less than the probability of immunity, then the tree stays immune; else it does not.
  D. If a random number is less than the probability of immunity, then the tree does not stay immune; else it does.

- **c.** For the tree to have a chance of burning due to fire at a neighboring site, give the value that at least one of *N*, *E*, *S*, *W* must have.

d. Give the most number of rules necessary to test if one of the parameters *N*, *E*, *S*, *W* is *BURNING*.
e. Give the rule where *N* is *BURNING*.

Quick Review Question 5 Complete the rule for *spread*(*site*, *N*, *E*, *S*, *W*) that applies to the situation where a site contains a non-burning tree that may be hit by lightning and burn. Assume that the function call *rand0to1*() returns a random floating point number between 0 and 1.

```
definemore(spread,
    spread(TREE, N::integer, E::integer,
           S::integer, W::integer)
     ________(_______ < probLightning * (1 - probImmune), _____, ______)
);
```

The following segment contains all the updating rules for the function *spread:*

```
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 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)
   );
```

Periodic Boundary Conditions

Quick Review Question 6 This question extends a matrix as in Figure 11.2.6 by attaching the last row to the beginning and the first row to the end of the original matrix.

- **a.** Write an expression for the last row of matrix *mat*.
- **b.** Write an expression for the first row of matrix *mat*.
- **c.** Complete the following statement to make *matNS* an extended matrix of *mat* as described in this question.

```
matNS := [_________, __________, __________]:
```

Quick Review Question 7 This question extends a matrix as in Figure 11.2.7.

- **a.** Write the long form (including the package designation) of a statement to make *trans* a matrix with the rows and columns of matrix *matNS* switched. Do not display *trans*.
- **b.** Make *matEW* a matrix containing the concatenation of the last row of *trans*, *trans*, and the first row of *trans*. Do not display *matEW*.
- **c.** Give the long form (including the package designation) of a call to a function to return a matrix with the rows and columns of *matEW* switched. Have the statement display the result.
- **d.** If the original matrix *mat* is of size 7-by-7, after extending the matrix as in this and the previous Quick Review Question, give the size of the extended matrix.

Applying a Function to Each Grid Point

Quick Review Question 8 This question develops the function *applyExtended*.

a. Complete the code to start the definition of *applyExtended*, which is to have parameters *fnc* and *matExt* and local variables *n*, *site*, *N*, *E*, *S*, and *W*.

```
applyExtended := _____________(fnc, matExt)
    ___________ n, site, N, E, S, W;…
```

b. Write the statement ending in a colon to assign to *n* the length of the internal (non-extended) matrix.

c. We can use the function *seq* and lists to generate the return matrix. Suppose *i* represents the row index and *j* the column index. To apply the function *fnc* to each internal cell of *matExt*, we let *i* (and *j*) vary between two values. Give the initial value of *i* (or *j*).

d. Give the final value of *i* (or *j*).

e. Select the symbol that separates the body of *seq* from a range of indices, such as $j = 2..(n + 1)$ or $i = 2..(n + 1)$.

A.	,	B.	.	C.	;	D.	:
E.	_	F.	/.	G.	/;	H.	/:

f. Figure 11.2.9 gives the coordinates of a site and its neighbors. Give the code for the (*i*, *j*)-element of matrix *matExt*.

g. Give the code corresponding to the neighbor to the north.

h. Give the code corresponding to the neighbor to the east.

i. Give the complete definition of *applyExtended*.

Simulation Program

Quick Review Question 9 Implement the loop in the *fire* function, assuming *grids* is a list containing the initial forest Quick Review Question 1 develops.

Display Simulation

Quick Review Question 10 This question develops the function *showGraphs* that produces a graphic corresponding to each simulation matrix in a list (*graphList*).

a. Complete the definition of a function *matchColor* to associate a color with a cell of matrix *mat*, as follows: yellow for *EMPTY*, forest green for *TREE*, and burnt orange for *BURNING*.

```
matchColor := (mat, i, j)->_________________ (mat[i, j],
   _____ EMPTY = ___________,
     TREE = _____________ (___________, 0.1, 0.75, 0.2),
     BURNING = _____________ (___________, 0.6, 0.2, 0.1)
   _____):
```

b. Give the function call to return the number of elements in *graphList*.

c. Give the statement to assign to local variable *g* the *k*-th element in *graphList*.

d. Complete the segment that assigns to *plotGrid* a sequence of colored squares, translated into their proper positions. The segment processes every cell of a matrix (list of *n* lists, each of *n* elements), *g*, of values a row at a time. The function *matchColor* returns the color for a square. Use the long form of the functions to translate and plot a square.

```
aSquare :=[[0, 0], [0, 1],[1, 1],[1, 0]]:
plotGrid := ___________ (__________ (
            plottools[translate](
               plots[polygonplot](aSquare,
                  axes = none,
                  scaling = CONSTRAINED,
                  _________ = matchColor(g, i, j)),
               j, i),
         j = 1..n), i = 1..n):
```

e. Complete the statement to plot the squares of *plotGrid* together and to append this display onto the end of list *plotList*.

```
plotList := [__________ (plotList), plots[___________](plotGrid)]:
```

f. Complete the statement to generate an animation of the graphis in *plotList*.

```
plots[_____________](plotList, _________________ = _________________);
```

g. Give the entire definition of *showGraphs*.

Answers to Quick Review Question

1. **a.**
```
local forest, forestExtended, grids, i, j;
global probLightning, probImmune;

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

b.
```
forest := [seq([seq(
   `if`(rand0to1() < probTree,
       `if`(rand0to1() < probBurning,
           BURNING, TREE),
        EMPTY),
    j = 1..n)], i = 1..n)];
```

c.
```
grids := [forest]:
```

2. **a.** *EMPTY*

b. *N::integer*

c. *EMPTY*

d. *define*

e.
```
define(spread,
   spread(EMPTY, N::integer, E::integer,
          S::integer, W::integer) =
      EMPTY
   );
```

3. **a.** *BURNING*
b. *S::integer*
c. *EMPTY*, which indicates an empty cell
d. *definemore*
e.
```
definemore(spread,
   spread(BURNING, N::integer, E::integer,
          S::integer, W::integer) =
      EMPTY
   );
```

4. **a.** *TREE*
b. B. If a random number is less than the probability of immunity, then the tree does not catch fire; else it does.
c. *BURNING* (2)
d. 4
e.
```
definemore(spread,
   spread(TREE, BURNING, E::integer, S::integer,
          W::integer) =
      `if`(rand0to1() < probImmune,
          TREE, BURNING)
);
```

5.
```
definemore(spread,
     spread(TREE, N::integer, E::integer,
               S::integer, W::integer) =
          `if`(rand0to1() < probLightning *
                                (1 - probImmune), BURNING, TREE)
        );
```

6. **a.** `mat[-1]`
b. `mat[1]`
c.
```
matNS := [mat[-1], op(mat), mat[1]]:
```

7. **a.**
```
trans := ListTools[Transpose](matNS):
```
b.
```
transEW := [trans[-1], op(trans), trans[1]]:
```
c.
```
ListTools[Transpose](transEW);
```
d. 9-by-9

8. **a.**
```
applyExtended := proc(fnc, matExt)
     local n, site, N, E, S, W;…
```
b.
```
n := nops(matExt) - 2:
```
c. 2
d. $n + 1$
e. A. ,
f. `matExt[i][j]` or `matExt[i, j]`
g. `matExt[i - 1][j]` or `matExt[i - 1, j]`
h. `matExt[i][j + 1]` or `matExt[i, j + 1]`

i.
```
applyExtended := proc(fnc, matExt)
    local n, site, N, E, S, W;
       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:
```

9.
```
for i from 1 to t do
    forestExtended := extendMat(forest):
    forest := applyExtended(spread, forestExtended):
    grids := [op(grids), forest];
end do;
```

10. **a.**
```
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)
    ]):
```

b.
```
nops(graphList)
```

c.
```
g := graphList[k]:
```

d.
```
aSquare :=[[0, 0], [0, 1], [1, 1], [1, 0]]:
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):
```

e.
```
plotList := [op(plotList), plots[display](plotGrid)]:
```

f.
```
plots[display](plotList, insequence = true);
```

g.
```
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;
```