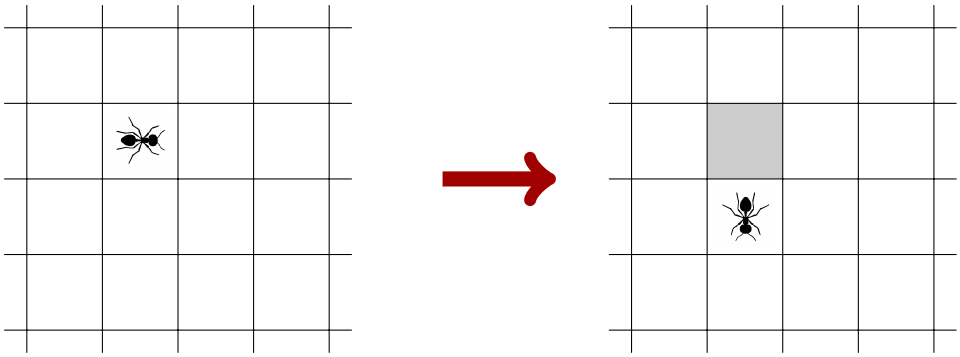
1 Background

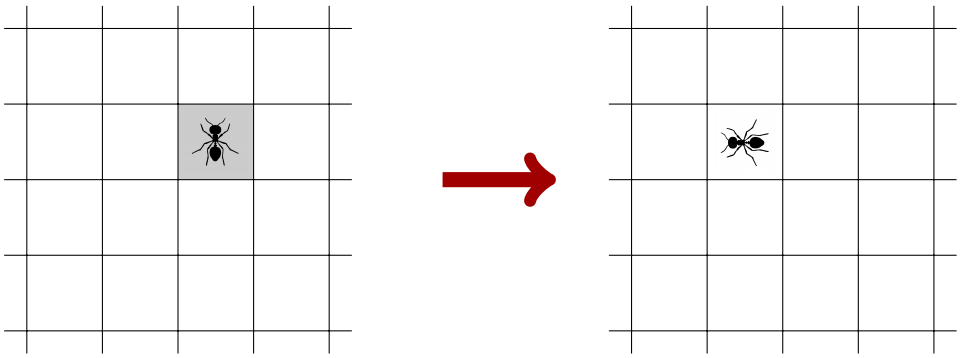
We will continue with walks in this section, however, this time we will do it in a deterministic fashion, that is, the direction and step size will not be driven by random processes, but for well defined deterministic rules. For this we will move to a type of discrete models known as *cellular automata*. In our study we will apply the name cellular automaton (CA) to the result of applying very simple rules to a rectangular grid, where each cell can only have two states, namely, on/off.

During this session we will focus on the CA known as the Langton's ant. Consider an ant walking in a direction on a grid, each grid cell is colored with two different colors, say, black and white. Let us assume that at a given time the ant walks into a cell, these are the only two options:

(O1) The cell is white: in this case the ant turns to the right and move forward. When it leaves the cell, it changes to black.



(O2) The cell is black: in this case the ant turns to the left and move forward. When it leaves the cell, it changes to white.



These two simple rules will describe the evolution of the ant, regardless of how many black/white cells are at the beginning of the experiment.

2 Problems

(P1) Create a square grid of size $n \times n$, $n = 100$. Use the function `numpy.zeros()` to do that. Note that initially all cells are going to have the value 0, which we will interpret as having white color

```
w = numpy.zeros([n, n], dtype = numpy.int8)
```

(P2) We will make use of the following notation for the directions (0 : east), (1: south), (2: west) and (3: north). Design a function that given a direction `d` and the color `c` of the cell in which the ant is standing, decides the next direction along which the ant will move

```
def nextdirection (d, c)
```

(P3) Write a function that given a that given a direction of motion `d` and a position on the grid `(i, j)`, returns the next position to which the ant will move

```
def moveforward (i, j, d):
```

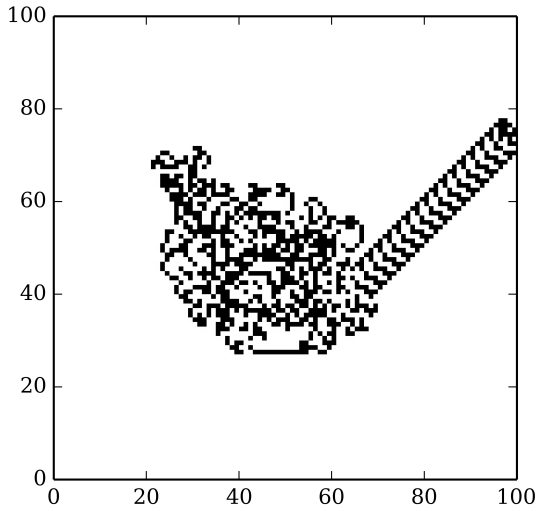
(P4) Design a function that takes care of the boundary problem. That is, if the next position to which the ant will move is not a valid index, the function will return 1 (0, otherwise).

```
def validindex (i, j, n):
```

(P5) Starting with `i = n/2`, `j = n/2` and `d = 0`, simulate 20000 steps of the ant, or until it leaves the grid.

(P6) Plot the final configuration of the grid

```
plt.imshow(w, interpolation='none', cmap=plt.cm.Greys, extent=(0, n, 0, n))
```



3 Solutions

(P1)
```
# initializing grid
n = 100
w = np.zeros([n, n], dtype = np.int8)
```

(P2)
```
# finds the next direction
def nextdirection(d, c):

    if c == 0:
        dnext = (d + 1) % 4
    else:
        dnext = (d - 1) % 4

    return dnext
```

(P3)
```
# steps forwards in the indicated direction
def stepforward (i, j, d):

    inext = i
    jnext = j

    if d == 0:
        inext = i + 1
    elif d == 1:
        jnext = j - 1
    elif d == 2:
        inext = i - 1
    else:
        jnext = j + 1


    return inext, jnext
```

(P4)
```
# decides if the index is out of boundary
def validindex (i, j, n):
    e = 0
    if (i < 0 or i >= n) or (j < 0 or j >= n):
        print 'index out of bounds'
        e = 1

    return e
```

(P5)
```
# initializing ant
i = n / 2
j = n / 2
d = 0

nsteps = 20000

for k in range(nsteps):
```

```
        c = w[i, j]

        # next direction
        d = nextdirection(d, c)

        # steps forward
        w[i, j] = 1 - c
        i, j = stepforward(i, j, d)

        # valid index?
        e = validindex(i, j, n)
        if e == 1:
            break
```

(P6) `plt.imshow(w, interpolation='none', cmap=plt.cm.Greys, extent=(0, n, 0, n))`