Brief Explanation Of The Code In ”2016/hw8/maxproduct.m”

George Papadopoulos

pgeorgios8@gmail.com

Abstract

In response to an email asking clarifications for the code in ”2016/hw8/maxproduct.m”, in this small informal article we explore the solution of the fifth problem and the algorithmic process that leads to its solution more thoroughly. It is the author’s view, meaning ”my opinion” that this is the most difficult problem of the eighth module and the most time consuming to solve throughout the course.

This article is divided into two sections, describing the two steps of this solution and the abstract thinking behind the first step that led to the second.

The problem we are asked to solve is to write a program with two inputs, a matrix A and a **positive** integer n (radius) that gives as output the indexes of the maximum product of n consecutive elements of the matrix in all directions, meaning row, column, diagonal and cross diagonal directions, firstly in ascending row wise order and secondly if all the elements belong to the same row in ascending column wise order. If no such product exists, the output must return the empty array and in the case of multiple maximum products the output must return the first one found.[Problem 11 - Euler Project]

In the two following sections for simplicity reasons [1] we will use the 2×2 matrix

$$A = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix}$$

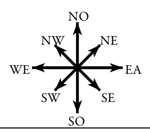
for which is obvious that the output we should expect is $[1, 2]$ and $[2, 1]$, since $A(1, 2) \cdot A(2, 1) = 12$ which is the maximum of all the possible products of two consecutive elements in this matrix.

1 String Assignment.

The problem we should first consider is that some elements miss some directions that lead to an error [2], since **matlab** expects only valid operations to be conducted and expects from the end user to respect that principle, for example we cannot get the third element of a two dimensional array. The first information we are given is that we should search in eight directions diagonal, cross diagonal, horizontal and vertical, the second information is that we should compute in the valid directions (the ones that exist) the product of n consecutive elements and store it ”somewhere” along with the indexes, then when all the elements of the matrix are scanned, go back to find where the maximum product was found in the matrix.
Suppose we are ”somewhere” on a dangerous terrain and the only tool we have in our hands is a compass with eight directions starting from north west in a clockwise direction each denoted by two letters NW, NO, NE, EA, SE, SO, SW, WE [3] and a piece of paper with some letters and numbers next to them like the following

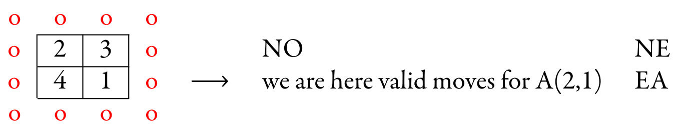
Direction	Steps
NE	3
NO	2
WE	2

that give us the safe path we can walk on in order to avoid the pits and find the exit. We can imagine that at each point we will have to stop, we are standing on an element of a matrix around which the ”x” denotes a point of the path and the ”o” a pit we should avoid,

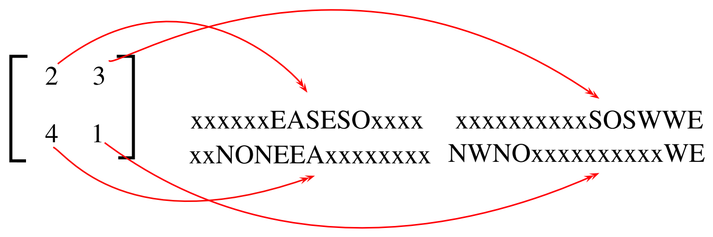
o	o	x	x	exit	compass
o	o	x	o	o	
o	o	x	o	o	
o	x	o	o	o	
x	o	o	o	o	

so what if we followed the same logic in our problem? Suppose we are standing at position $(2, 1)$ of matrix A, with the same logic described above we can imagine an image like the one below

[1] TEX typesetting might be difficult some times believe me.
[2] Index exceeds matrix dimensions.
[3] north-west, north, north-east, east, south-east, south, south-west, west



so we can write the *valid* directions for $A(2,1)$ a a sting sequence like 'NONEEA' which is translated into north, north-east, east. What if we used a fixed string with all the directions starting from NW in the clockwise direction? So let directions $=$ *”NWNONEEASESOSWWE”*, which is a 16-bit string. For every element of a matrix we can check each direction by getting two bits at a time of the string ”directions” and replace them if there is no path of length n to the specified direction with an *”xx”* 2-bit string. So if we found a way understood by the computer to do the checking for $A(1,1)$ for $n = 2$ we would get the string *”xxxxxxEASESOxxxx”* $\dagger$, meaning that the valid paths are east, south-east and south. An easy way to do it with matlab would be to create a cell of the same size of matrix A and assign to its empty positions the outputs of the form $\dagger$. For our case a schema showing the above description would be



and the code in matlab which illustrates what is described here is the following.

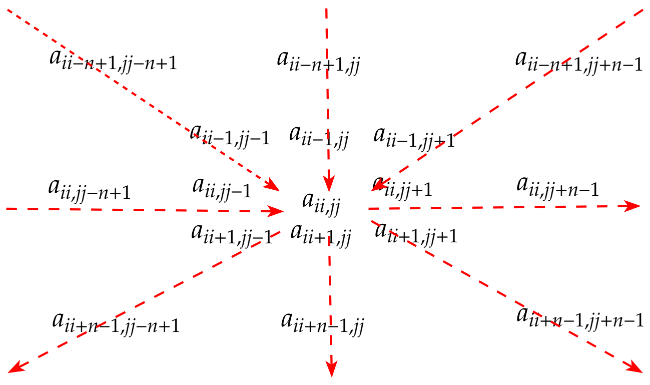
```
function v = maxproduct(A,n)
assert(isscalar(n) && isnumeric(n) && n >= 0 && ceil(n) == n,'error radius must be greater
than zero!');
c = cell(size(A,1),size(A,2));
for ii = 1 : size(A,1)
    for jj = 1 : size(A,2)
        directions = 'NWNONEEASESOSWWE';
        try %NW
            A(ii-n+1:ii,jj-n+1:jj);
        catch
            directions(1:2) = 'xx';
        end
        try %NO
            A(ii-n+1:ii,jj);
        catch
            directions(3:4) = 'xx';
        end
        try %NE
            A(ii-n+1:ii,jj:jj+n-1);
        catch
            directions(5:6) = 'xx';
        end
        try %EA
            A(ii,jj:jj+n-1);
        catch
            directions(7:8) = 'xx';
        end
        try %SE
            A(ii:ii+n-1,jj:jj+n-1);
        catch
            directions(9:10) = 'xx';
        end
        try %SO
            A(ii:ii+n-1,jj);
        catch
            directions(11:12) = 'xx';
        end
        try %SW
            A(ii:ii+n-1,jj-n+1:jj);
        catch
            directions(13:14) = 'xx';
        end
        try %WE
            A(ii,jj-n+1:jj);
        catch
            directions(15:16) = 'xx';
        end
        c{ii,jj} = directions;
        end
    end
    v = c;
end
```

We can test this function for various matrices with various $n > 0$ values and this concludes section 1 having solved the problem of the invalid directions for each matrix element.
Examples:

```
>>maxproduct([2 3;4 1],2)

 ans =
      'xxxxxxEASESOxxxx'      'xxxxxxxxxxSOSWWE'
      'xxNONEEAxxxxxxxx'      'NWNOxxxxxxxxxxWE'

 >>class(ans)

 ans =

      cell

>>maxproduct([1 2 3 ; 4 5 6;7 8 9],3)

 ans =

      'xxxxxxEASESOxxxx'   'xxxxxxxxxxSOxxxx'     'xxxxxxxxxxSOSWWE'
      'xxxxxxEAxxxxxxxx'   'xxxxxxxxxxxxxxxx'     'xxxxxxxxxxxxxxWE'
      'xxNONEEAxxxxxxxx'   'xxNOxxxxxxxxxxxx'     'NWNOxxxxxxxxxxWE'

>>class(ans)

 ans =

      cell
```

2 Maximum Product And Indexes.

For the second part of the program, we should be aware that the output must sort the indexes in a row wise ascending order and secondly in a column wise ascending order. Here is a figure to give us an impression of how we should sort the indexes, the arrows' tails show to us the first term's indexes in the product we compute (if there exists an element in that direction and the direction is not marked as "xx") and the arrow's head the element which is scanned at each time (imagine a planet with its satellites).



The question we should next answer is how we should compute the product in each direction and how we should store the indexes, for which we got a glimpse of how they should be sorted correctly. Let us examine each direction separately:
NW.
In this case the product is the product of the elements of the diagonal of the north-western $n \times n$ matrix with its lowest right element being $a_{ii,jj}$ and this is as written in matlab as $A(ii - n + 1 : ii, jj - n + 1 : jj)$.

```
product = prod(diag(A(ii-n+1:ii,jj-n+1:jj)))
indexes = [(ii-n+1:ii)',(jj-n+1:jj)']
```

NO.
In this case the product is the product of the elements of the array in the northern direction $n \times 1$ with its last element being $a_{ii,jj}$ and this is written in matlab as $A(ii-n+1:ii,jj)$.

```
product = prod(A(ii-n+1:ii,jj)))
indexes = [(ii-n+1:ii)',repmat(jj,n,1)]
```

NE.
In this case the product is the product of the elements of the reverse diagonal of the north-eastern $n \times n$ matrix with its lowest left element being $a_{ii,jj}$ and this is written in matlab as $A(ii-n+1:ii,jj:jj+n-1)$. In order to compute the product we will have to flip the matrix upside down and take the indexes from right to left in a column wise direction.

```
product = prod(diag(flipud(A(ii-n+1:ii,jj:jj+n-1))))
indexes = [(ii-n+1:ii)',(jj+n-1:-1:jj)']
```

EA.
In this case the product is the product of the elements of the array of length n in the eastern direction with its first left element being $a_{ii,jj}$ and this is as written in matlab as $A(ii,jj:jj+n-1)$.

```
product = prod(A(ii,jj:jj+n-1))
indexes = [repmat(ii,n,1),(jj:jj+n-1)']
```

SE.
In this case the product is the product of the elements of the diagonal of the south-eastern $n \times n$ matrix with its upper left element being $a_{ii,jj}$ and this is as written in matlab $A(ii:ii+n-1,jj:jj+n-1)$.

```
product = prod(diag(A(ii:ii+n-1:-1,jj:jj+n-1:-1)))
indexes = [(ii:ii+n-1)',(jj:jj+n-1)']
```

SO.
In this case the product is the product of the elements of the array in the southern direction $n \times 1$ with its first element being $a_{ii,jj}$ and this is written in matlab as $A(ii:ii+n-1,jj)$.

```
product = prod(A(ii:ii+n-1,jj)))
indexes = [(ii:ii+n-1)',repmat(jj,n,1)]
```

SW.
In this case the product is the product of the elements of the reverse diagonal of the south-western $n \times n$ matrix with its upper right element being $a_{ii,jj}$ and this is written in matlab as $A(ii:ii+n-1,jj-n+1:jj)$. In order to compute the product we will have to flip the matrix upside down and take the indexes from right to left in a column wise direction.

```
product = prod(diag(flipud(A(ii:ii+n-1,jj-n+1:jj))))
indexes = [(ii:ii+n-1)',(jj:-1:jj-n+1)']
```

WE.
In this case the product is the product of the elements of the array of length n in the western direction with its last right element being $a_{ii,jj}$ and this is as written in matlab as $A(ii,jj-n+1:jj)$.

```
product = prod(A(ii,jj-n+1:jj))
indexes = [repmat(ii,n,1),(jj-n+1:jj)']
```

and now that we know what to do with the indexes and the products in each direction we are ready for the final step. We can assign an initial value $p = 0$ to the product and an empty initial array of indexes $v = []$, such that if no product greater than zero is found, then the output of the program shall return the empty array. We must iterate through all the elements of cell c created in the first section, get the string of each position, split the string into 8 parts beginning from left to right by step two, see which match a valid direction meaning not equal to "xx", then compute the product and finally if that product is greater than p set v equal to the indexes of that combination of terms. In this case the "switch-case" function turns to be useful, this gives us the code presented the next page and our program is ready!

```
function v = maxproduct(A,n)
assert(isscalar(n) && isnumeric(n) && n ⩾ 0 && ceil(n) == n,'error radius must be greater
than zero!');
c = cell(size(A,1),size(A,2));
for ii = 1 : size(A,1)
    for jj = 1 : size(A,2)
        directions = 'NWNONEEASESOSWWE';
        try %NW
            A(ii-n+1:ii,jj-n+1:jj);
        catch
            directions(1:2) = 'xx';
        end
        try %NO
            A(ii-n+1:ii,jj);
        catch
            directions(3:4) = 'xx';
        end
        try %NE
            A(ii-n+1:ii,jj:jj+n-1);
        catch
            directions(5:6) = 'xx';
        end
        try %EA
            A(ii,jj:jj+n-1);
        catch
            directions(7:8) = 'xx';
        end
        try %SE
            A(ii:ii+n-1,jj:jj+n-1);
        catch
            directions(9:10) = 'xx';
        end
        try %SO
            A(ii:ii+n-1,jj);
        catch
            directions(11:12) = 'xx';
        end
        try %SW
            A(ii:ii+n-1,jj-n+1:jj);
        catch
            directions(13:14) = 'xx';
        end
        try %WE
            A(ii,jj-n+1:jj);
        catch
            directions(15:16) = 'xx';
        end
        c{ii,jj} = directions;
    end
end
p = 0;v = [];
for ii = 1 : size(c,1)
    for jj = 1 : size(c,2)
        el = c{ii,jj};
        for kk = 1 : 8
            input = el(2*kk-1:2*kk);
            switch input
                case 'NW'
                    pr    = prod(diag(A(ii-n+1:ii,jj-n+1:jj)));
                    ind  = [(ii-n+1:ii)',(jj-n+1:jj)'];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
                case 'NO'
                    pr    = prod(A(ii-n+1:ii,jj));
                    ind  = [(ii-n+1:ii)',repmat(jj,n,1)];
                    if pr > p;
                        v = ind;
                        p = pr;
                    end
                case 'NE'
                    pr    = prod(diag(flipud(A(ii-n+1:ii,jj:jj+n-1))));
                    ind  = [(ii-n+1:ii)',(jj+n-1:-1:jj)'];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
                case 'EA'
```

```
                    pr    = prod(A(ii,jj:jj+n-1));
                    ind   = [repmat(ii,n,1),(jj:jj+n-1)'];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
                case 'SE'
                    pr    = prod(diag(A(ii:ii+n-1,jj:jj+n-1)));
                    ind   = [(ii:ii+n-1)',(jj:jj+n-1)'];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
                case 'SO'
                    pr    = prod(A(ii:ii+n-1,jj));
                    ind   = [(ii:ii+n-1)',repmat(jj,n,1)];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
                case 'SW'
                    pr    = prod(diag(flipud(A(ii:ii+n-1,jj-n+1:jj))));
                    ind   = [(ii:ii+n-1)',(jj:-1:jj-n+1)'];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
                case 'WE'
                    pr    = prod(A(ii,jj-n+1:jj));
                    ind   = [repmat(ii,n,1),(jj-n+1:jj)'];
                    if pr > p
                        v = ind;
                        p = pr;
                    end
            end
        end
    end
end
end
```