VG101 Recitation Class - Week 3

Zhang Yichi

UM-SJTU Joint Institute

May 30, 2018

A Reminder

Please see the following announcement about extending your Matlab and follow the instructions.

https://umjicanvas.com/courses/547/discussion_topics/10797

Function

What is function?

A function is a sequence of scripts that can run with several input arguments and can return some output arguments (also called return values).

The main differences between Matlab scripts and Matlab functions are

- Functions accept input arguments and return values (but not mandatory) while scripts do not.
- The variables created in functions will not be stored in the workspace as those in a script do.

Why do we need functions?

- The first reason is that using function can simplify the programming process. In a program some certain sequences of commands might appear for many times. Using a function will prevent you from typing the same codes again and again. Also it will be more convenient when you want to change this part of codes.
- Another situation is that when you are implementing a program for some other users, you should think carefully about **what** some codes do and **how** they do it. But in many cases the user only need to know the **what** part. Then using a function will be more friendly for the user. This is also called abstraction in programming.

How to implement a function?

In Matlab, the function can be put in two places.

- A .m file with the same name as the function (then all the other programs can use this function)
- A .m file which is for other scripts or functions (only those scripts/functions in this file can use this function and it is also called a subfunction).

The basic syntax of defining a function is like:

```
function [out1,out2,...]=Functionname(in1,in2,...)
    ..
    out1 = ..;
    out2 = ..;
    ..
end
```

Comments on the functions

In order to let the user have a better understanding of your function, you can have some comments on the head of it.
Two basic pieces of information is the **effect of the function and the requirement for the input argument**. (In some cases, you should also include information about whether the value of any input argument is modified in the function, but it does not appear much in Matlab)

```
function M = mymagicdoubleeven(n)
% The function will return a magic matrix of order
% n should be a multiple of 4
    M = (1:n:n*n)'+(0:n-1);
    K = (mod((1:n)'+(1:n),4)==1)|...
        (mod(abs((1:n)'-(1:n)),4)==0);
    M(K) = n*n+1-M(K);
end
```

Common Matlab Functions

First, if you want to search some common functions in Matlab documentation, you can use *help elfun, help specfun and help elmat*.

And as Manuel has already included some functions in the lectures, I will not spend time talking about them and I will only add some extra points.

```
%% Rounding functions
round(n) % round n to the nearest integer
 fix(n) % round n towards 0
 ceil(n) % round n towards infinity
 floor(n) % round n towards minus infinity

%% Remainder Function
mod(x,y)
rem(x,y)
% When sign(x)~=sign(y),sign(mod(x,y))==sign(y)
% but sign(rem(x,y))==sign(x)
```

```
%% Matrix Generation
zeros(m,n) % generate a m*n 0 matrix
ones(m,n) % generate a m*n 1 matrix
eye(n) % generate a n*n identity matrix
linspace(a,b,n) % generate linear spaced sequence
%% Matrix Properties
size(A) % the size of A
numel(A) % the number of elements of A
isempty(A) % judge whether A is empty
isequal(A,B) % judge whether A and B are equal
%% Matrix Operations
repmat(A,m,n) % replicate A in the m*n form
reshape(A,m,n) % reshape A into a m*n matrix
```

sortnames.m

```
function sortnames(finput, foutput)
  fd1=fopen(finput,'r');
  i=1;
  line=fgetl(fd1);
  while line ~= -1
    a=find(isspace(line),2);
    info{i}=sprintf('%s %s %s\n', line(a(1)+1:a(2)-1), ...
      line(1:a(1)-1), line(a(2)+1:end));
    i=i+1; line=fgetl(fd1);
  end
  fclose(fd1);

  fd2=fopen(foutput,'w');
  for j=randperm(i-1)
    fprintf(fd2,info{j});
  end
  fclose(fd2);
```

A trick of outputting a matrix using fprintf()

When we want to output a matrix into a file, we need to set the alignment clear enough. And here is a trick using *fprintf()* to make your output process more efficient.

When we write the function like the following:

```
fprintf(fout,'%5d%5d%5d%5d\n',A);
```

It seems that the function only output 4 elements, but actually it is not. This command will repeat the format and output A(1)-A(4), A(5)-A(8),... until the end.

However, the result output looks different from the original matrix. It is because that the counting process of elements in a matrix is first along the column, then along the row. In other words, A(2) is in fact the element on the second row, first column.

To solve this problem, just let the function to output the transpose of the matrix.

```
fprintf(fout,'%5d%5d%5d%5d\n',A');
```

But this is still not enough, because it can only output 4 elements in a row. What if the column number of a matrix is not 4?

It is also easy to solve. Just find the column numbers(n) of A and make a string with n consecutive '%5d's and a '\n' at the end.

```
n = size(A,2);
form = [repmat('%5d',1,n),'\n']
fprintf(fout,form,A')
```

Here I will leave another thing to you to think about: what if there are elements larger than 10000 in A? How can the separation between numbers be dynamically set based on the matrix?

Recursion

Definition

Recursion is a technique in programming letting a function to call itself.

The most common algorithm that using recursion is called divide and conquer, which means split a complex problem into similar but smaller problems. The splitting process will repeat until an initial condition, where the answer can be obviously given, is reached. Then the process goes back to solve the original problem layer by layer.

Use Recursion to solve mathematical problems

- For example, when we try to find the factorial of n

```
function f = fact(n)
    if n==0 % initial condition
        f = 1;
    else
        f = n*fact(n-1);
    end
end
```

When you are solving the mathematical problems using recursion, the most important part is finding the **recursive expression** of the answer.

Exercise: Josephus Problem

A group of n people are standing in a circle, and are numbered from 1 to n in the clockwise order. Now a count-off by m process begins from the No.1 person and goes in the clockwise order. Every time a person is counted as m, he should go out of the circle. The process is repeated until there is only one person left. So what is the number of the last man?

Please implement a function josephus(n,m) which returns the number of the last person.

The problem is easy to solve using recursion.

- When $n = 1$, it is obvious that the only man in the circle is the only man left, so in this case the number of the last man is 1
- Suppose that $josephus(n-1, m) = k$, then consider the situation that there are n people. Obviously the first one to get out is No.m (or $mod(m, n)$ if $m > n$)
- So there are only $n-1$ men left now, and it can be seen that the last man should be the kth man counting from No.$m+1$ (or $mod(m+1, n)$), which should be No.$mod((m+k), n)$
- The mod function has a limitation that it will return 0. However what we want is No.n instead of No.0, so a if statement should be written.

```matlab
function y = josephus(n,m)
    if n==1
        y = 1;
    else
        y = mod(josephus(n-1,m)+m,n);
        if y==0
            y = n;
        end
    end
end
```

Using recursion to solve more complex problems

Sometimes a question is not so pure-mathematical and might be a little bit complex. And I think the most crucial part of solving this kind of problems is properly setting the **input arguments** and **return values** of a function.

- As for the input arguments, they should be complete enough to describe a partial procedure in solving a large problem.
- Then what should you do is splitting a whole procedure into several partial procedures.
- Then is to find the initial conditions. Note that in many cases, there might be multiple initial conditions.

Example: Hanoi Tower

A Tower of Hanoi consists of three rods and a number of disks of different sizes, which may slide onto any rod. The system starts with disks on one rod, with disks in ascending order of size, the smallest on the top. You job is to move every all to another rod, under such conditions:

- Only one disk can be moved at a time.
- A disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Display all the moves.

https://www.mathsisfun.com/games/towerofhanoi.html

- The whole process is to move n disks from rod 1 to rod 3. So the definition of the partial procedure should also be move some disks from a certain rod to another rod.
- So I define the subfunction $hanoimove(n, a, b)$, which is the function to print the process of moving n disks from rod a to rod b.
- It is obvious that the initial condition is when $n = 1$, then we just move it from a to b.
- In the case that $n > 1$, it can be seen the disk that should be settled on rod b is the last disk which is the largest. But before that, we should remove all the disks above it, and they should go to the third rod c.

- When all the disks are moved to rod c, then we just move the bottom disk to rod b.
- At last, we should move the disks that has been moved to rod c back to rod b. Then the whole process is over.

So $hanoimove(n, a, b)$ includes the three partial procedures:
$hanoimove(n-1, a, c)$, $hanoimove(1, a, b)$ and $hanoimove(n-1, c, b)$

```matlab
function hanoi(n)
    hanoimove(n,1,3);
end

function hanoimove(n,a,b)
    if n==1
        fprintf('%d --> %d\n',a,b);
    else
        c = 6-a-b;
        hanoimove(n-1,a,c);
        hanoimove(1,a,b);
        hanoimove(n-1,c,b);
    end
end
```

Why using recursion?

It is because that in many cases a recursion is more "obvious", and is easier to implement and more clear for the reader to understand. But it is not a method to save time and memory, and actually improper recursion has risk of memories.