Introduction to Programming in MATLAB

Abhinav Kshitij

Arizona State University

akshitij@asu.edu

April 24, 2020

Overview

Program Control

Loops and conditionals control program flow. While conditionals evaluate logical conditions to select the right branch of a program, loops (FOR-loop, WHILE-loop) repeat a set of statements contained within their own block. Loops may contain conditionals and more loops, depending on the application at hand although it comes at a cost - slower execution speeds.

Program Control

Numerous predefined functions are available in Matlab that are optimized to perform tasks involving loops and conditionals, e.g. `sort()`, `find()`, `sum()`, `prod()`. However advanced engineering courses require students to write their own set of loops and conditionals: Use of predefined functions may not be permitted in HWs and projects. Regardless of whether such functions are available or not, designing program control lies at the heart of programming as an essential skill.

FOR-Loop

FOR-loops run an index variable through a series of integers (typically an arithmetic series) defined by the colon operator. An arithmetic series is a set of integers with a start, an increment value and an end value. For example, the colon notation `3:2:7` generates the series - `3, 5, 7`.

```
for i = 3:2:7
    fprintf("i = %d\n", i);
end
```

The loop executes as many times as the number of elements present in the series (in this case three) with the index variable (`i`) taking on the three values at each iteration.

FOR-Loop: Arrays

FOR-loops work closely with arrays. The series generated by the colon operator allows a FOR-loop to visit each of its nth array element. The loop variable (i) is used for accessing the third, fifth and seventh array element (a(i)) by taking on values `3,5,7`, generated by `3:2:7`.

```
a = [8 7 4 3 6 2 1 9 3];
for i = 3:2:7
    fprintf("a(%d) = %d\n", i, a(i));
end
```

FOR-Loop: Arrays

Typically a FOR-loop is used to accessing all array elements between two limits, called an array section. The increment is dropped out, setting its value to one. In this example the FOR-loop prints the all of the elements of array `a`. Here the `length()` function determines the end value in the colon specifier.

```
a = [8 7 4 3 6 2 1 9 3];
for i = 1:length(a)
    fprintf("a(%d) = %d\n", i, a(i));
end
```

This is the most common form of the FOR-loop. It finds another important application in printing column entires for each row of a tabular output, CSV files, Excel sheets and images.

FOR-Loop: File Output

Loop indices and the `fprintf()` function are used in printing tables and generating output files most often in CSV format that can be opened as a regular Excel sheet. The following example prints every third element of arrays `a` and `b` to the file `output.csv`.

```
a = [8 7 4 3 6 2 1 9 3];
b = [6 8 3 2 6 3 0 2 1];
fID = fopen('output.csv','w');
fprintf("i, a, b \n"); % column header
for i = 1:3:length(a)
    fprintf(fID, "%d, %d, %d \n", i, a(i), b(i));
end
fclose(fID);
```

FOR-Loop: Index-based operations

Equivalently array elements can be referenced using the colon notation. The third, fifth, and the seventh element of array a can be directly accessed with `a(3:2:7)` line. This method is often used while slicing out subsections of arrays for data processing. Q: Why use a FOR-loop when a single line of code is enough for accessing array elements?

Index-based operations

There are two important situations when no other alternative to a FOR-loop exists. One of them is printing out rows in a tabular form. But the more application comes in the form of performing computations that involves two or more array elements.

Example: Compute local (moving) averages

To determine the local average we visit each of the "interior" elements and compute the average of the element at that location (`a(i)`), its left neighbor (`a(i-1)`), and its right neighbor (`a(i+1)`). Since there is no left or right neighbor available to compute the local average for the first and the last array element, we can only run the FOR-loop within the "interior" elements of `a`, i.e. from `a(2)` to `a(8)`.

```
a = [8 7 4 3 6 2 1 9 3];
avg = zeros(length(a), 1); % contains 9 elements
for i = 2:length(a)-1
    avg(i) = (a(i-1) + a(i) + a(i+1))/3;
    fprintf("avg(%d) = %4.2f\n", i, avg(i));
end
```

Arrays, Conditionals and Loops

.

The first major use of FOR-loop comes in the form of using conditionals on one or several array elements. Programming languages come with a set of inbuilt functions that are tested and optimized for speed and efficiency. Nearly all major inbuilt functions make use of conditionals and loops under the hood. Common array operations like `search` and `sort` apply IF-statements to one or more array elements within FOR-loops. While other inbuilt functions use conditionals with WHILE loops.

Example: Max, Min value in a 1D array


```
a = [8 7 4 3 6 2 1 9 3];
max = a(1); % Assign max to the first array element
min = a(1); % Assign min to the first array element
for i = 1:length(a)
    % Check for max value
    if (a(i) >= max)
        max = a(i);
    end
    % Check for min value
    if (a(i) <= min)
        min = a(i);
    end
end
fprintf("Max = %d, Min = %d \n", max, min);
```

Example: Compute local (moving) averages - II

Earlier we left out the first and the ninth element of array while determining the moving averages. Now here we recognize that moving averages at "boundary" elements, i.e. a(1) and a(9) can be still determined by taking a two-point average of their immediate available neighboring points. Therefore to compute `avg(1)`, we take the average of `a(1)` and its immediate right neighbor; to compute `avg(9)`, we take the average of `a(9)` and its immediate left neighbor. We therefore have different rules for three cases, and they can be enforced by conditionals (IF-ELSEIF statements).

Example: Compute local (moving) averages - II


```
a = [8 7 4 3 6 2 1 9 3];
avg = zeros(length(a), 1);
for i = 1:length(a)
     if (i == 1)
         avg(1) = (a(i) + a(i+1)) / 2;
     elseif (i == length(a))
         avg(i) = (a(i) + a(i-1)) / 2;
     else
         avg(i) = (a(i-1) + a(i) + a(i+1))/3;
     end
     fprintf("avg(%d) = %4.2f\n", i, avg(i));
end
```

Minimize the use of conditionals in loops

Even though the previous example determines the moving averages using loops and conditionals, we MUST try and avoid using multiple conditionals inside loops, since every check on an array element slows down program execution: The code checks for all three conditions while visiting every single array element. This may not be evident in small arrays but certainly in large arrays (with a few million elements), the time difference could be significant sometimes may result in deep cost and time overruns in large projects. Therefore if possible, ELIMINATE as many conditionals possible inside loops. Here we determine two-point moving averages outside the loop for "boundary" elements.

Example: Compute local (moving) averages - III


```
a = [8 7 4 3 6 2 1 9 3];
avg = zeros(length(a), 1);
% Two-point moving averages
avg(1) = (a(1) + a(2)) / 2;
avg(end) = (a(end-1) + a(end)) / 2;
% Three-point moving averages
for i = 2:length(a)-1
      avg(i) = (a(i-1) + a(i) + a(i+1))/3;
end

% Print moving averages
for i = 1:length(a)
     fprintf("avg(%d) = %4.2f\n", i, avg(i));
end
```

Nested FOR-loop and 2D array

If `a(i)` refers to an element in a 1D array, then one can use `a(i,j)` to refer to an element in a 2D array. 2D arrays as the name suggests contains data in two dimensions, where an element in the i-th row and j-th column is accessed by `a(i,j)`. Instead of one, we use two loop indices one to "move" in x-direction, and the other to "move" in the y-direction by adding an "inner" FOR-loop within an "outer" FOR-loop.

```
a = [8 7 4; 3 6 2; 1 9 3];
for i = 1:3
    for j = 1:3
        fprintf("a(%d, %d) = %d \n", i, j, a(i,j));
    end
end
```

Array masks

Array masks present themselves as alternative to using conditionals and loops, most notably when individual array elements satisfying a set of conditions are to be accessed. Applying filters to an array is one such example. An array mask is a logical array, having the same length as the array on which it is applied on, that select array elements on positions where the mask has a TRUE value. The selected elements can be assigned to a new array of length lesser than or equal to the original array. In the following example we select all elements greater than 5 and less than 8 from a using the `fivetoeight` array mask.

```
a = [8 7 4; 3 6 2; 1 9 3];
fivetoeight = and(a > 5, a < 8);
b = a(fivetoeight)
```

Array masks

To understand how this code "flows", add a breakpoint in front of each line and click on Run and Advance (See the slides on adding breakpoints).

LINE 1 Initialize `a`.

LINE 2 Create an array mask (`fivetoeight`) by providing the two required conditions (a $>$ 5 AND a $<$ 8) to the `and()` function.

LINE 3 Apply array mask to `a`; save results to a new array `b`.

WHILE Loop

- WHILE-loops are controlled by conditionals.
- Statements within this loop will get executed as long a logical condition remains TRUE, which means that it has the potential to run indefinitely - resulting in a condition known as an infinite loop.
- To ensure the loop executes only a finite number of times, the program must change the condition being tested to FALSE or add a BREAK statement.
- WHILE-loops are used when we do not know how many times a loop must be executed in beforehand.
- They may additionally contain FOR-loops, conditionals and array operations.

WHILE-Loop: Infinite series

Perhaps the most important use of a WHILE-loops comes in the form of adding up terms of an infinite series. An infinite series is the workhorse of all mathematical calculations from displaying the value of pi and computing the square root of number to implementing artificial intelligence and automation in modern platforms.

Equation solvers, simulations, graphics require WHILE-loops running on an infinite series. WHILE-loops form the backbone of all iterative numerical methods required in computer simulations and modeling in scientific and engineering applications.

WHILE-Loop: Infinite series

General points to consider while designing a WHILE-loop:

1. Each term of an infinite loop is produced by evaluating the nth term of a general expression within a WHILE-loop, and added to the current series evaluated up to `n - 1` terms.
2. The loop index (n) is incremented at the end of each loop.
3. Quite commonly an error metric is determined within the loop such that loop terminates execution once this error falls below a certain threshold.
4. Intermediate results (iteration number, current series) are printed to monitor loop execution. CAUTION: Printing within loops will drastically slow down execution!
5. Use CTRL+C to halt program execution.

Example: Estimate PI up to 7 decimal places


```
true_pi = pi();
my_pi = 0;
n = 0;
err = true_pi;
fprintf("iter, pi, error \n");
while (err > 1e-8)
    my_pi = my_pi + (4 * (-1)^n)  / (2*n + 1);
    n = n + 1;
    err = abs(true_pi - my_pi);
    if (rem(log10(n),1) == 0)
       fprintf('%-8d, %22.15e, %8.3e \n',n,my_pi,err)
    end
end
```

Example: Estimate PI up to 7 decimal places - II


```
res = 0; n = 0; err = Inf;
tol = 1e-8; maxstep = 1e8;
fID = fopen('pi.csv','w');
fprintf(fID, "iter, pi, error \n");
while (err > tol && n < maxstep)
    res = res + (4 * (-1)^n)  / (2*n + 1);
    n = n + 1;
    err = abs(res - pi());
    if (rem(log10(n),1) == 0)
       fprintf(fID,'%-8d, %22.15e, %8.3e \n',n,res,er
    end
end
fprintf('%-8d, %22.15e, %8.3e \n',n,res,err);
fclose(fID);
```

Example: Estimate PI up to 7 decimal places - III

Convert into a function and save in the `lib/` folder.

```
function [res] = my_pi()
%MY_PI Estimates pi upto 7 decimal places
res = 0; n = 0; err = Inf;
tol = 1e-8;
maxstep = 1e8;

while(err > tol && n < maxstep)
    res= res + (4 * (-1)^n) / (2*n + 1);
    n = n + 1;
    err = abs(pi() - res);
end
```

End