MATLAB Basics

WDRP - Simple Discrete Models in Biology and MATLAB

Contents

1 Conditionals

Sometimes a problem is more complex than a straightforward calculation. How does a video game know when to add points to a score? Or how does your phone know to open an app when you tap the screen. Of course there is first some input, in the videogame maybe the character touches the coin, or on your phone you tapped the icon. But then what? This is where conditionals come into play. That is, we want the computer to perform an action when a certain condition is met.

1.1 If Statements

Here is the structure of a simple conditional statement.

if *expression*
    statement
end

The "expression" is some statement outputting a truth value, like $x > 3$. The "statement" is whatever block of code you want the computer to do given the "expression" has truth value 1. If the truth value is 0 then the computer ignores the block of code. Here is an example

```
>> EvenNum = input('Enter an even number: ');

if mod(EvenNum,2) ~= 0
    fprintf('Hey! Thats not even! \n');
    EvenNum = EvenNum+1;
    fprintf('Instead, %1.0f is an even number \n', EvenNum);
end
Enter an even number: 13
Hey! Thats not even!
Instead, 14 is an even number
```

Remember, a number is even if it's divisible by 2. Equivalently, a number is even if the remainder when dividing by 2 is 0. Since a number is either even or odd, if its not even, it must be odd. This is what the "expression" part is checking, if the number entered is odd. Turns out knowing some math is useful!

The problem with this code is that when you enter an even number, nothing happens:

```
>> EvenNum = input('Enter an even number: ');

if mod(EvenNum,2) ~= 0
    fprintf('Hey! Thats not even! \n');
    EvenNum = EvenNum+1;
    fprintf('Instead, %1.0f is an even number \n', EvenNum);
end
Enter an even number: 2
>>
```

It would be nice if we could have the computer say something when the number entered is even. However, we can't just throw in something at the end, since if we entered an odd number it would still output whatever we tell it to. Like this

```
>> EvenNum = input('Enter an even number: ');

if mod(EvenNum,2) ~= 0
    fprintf('Hey! Thats not even! \n');
    EvenNum = EvenNum+1;
    fprintf('Instead, %1.0f is an even number \n', EvenNum);
end

fprintf('Congrats! You can read and know what "even" means! \n');
Enter an even number: 13
Hey! Thats not even!
Instead, 14 is an even number
Congrats! You can read and know what "even" means!
```

Instead, we would have to put in another conditional statement that checks if the number entered is even. But why should we have to do another calculation when we already know if the number is not odd, it has to be even? This is where the next section comes in handy.

1.2 Using elseif and else

As said in the last section, we want to be able to combine multiple conditional statements into one block of text. To do this we use "elseif" and "else". Here is a general set up:

if *expression*
    statement
elseif *expression*
    statement
        ⋮
elseif *expression*
    statement
else
    statement
end

"elseif" is used as a secondary "if". Secondary is a bit misleading since we can do as many "elseif" as we want. So if the first "if" expression is not satisfied, the "elseif" is checked.

Now, if all of the "elseif" expressions are not satisfied, the last thing is the "else". For "else" we don't need to include an expression. If the "if" or all of the "elseif" are not satisfied, the "else" statement is automatically used. Here's an example, along with some additional cases:

```
>> var = input('Enter a guess for the code: ');
code = 4;

if var == code
    fprintf('you got it right! \n')

elseif var > code && mod(var,2) == 0
    fprintf('Close! your guess was too big, but you guessed correctly that the code is even \n')

elseif var > code && mod(var,2) ~= 0
    fprintf('Try again, its too big, and the code should be even \n')

elseif var < code && mod(var,2) == 0
    fprintf('Close! your guess was too small, but you guessed correctly that the code is even \n')

else
    fprintf('Try again, its too small, and the code should be even \n')

end
Enter a guess for the code: 3
Try again, its too small, and the code should be even

Enter a guess for the code: 6
Close! your guess was too big, but you guessed correctly that the code is even

Enter a guess for the code: 13
Try again, its too big, and the code should be even

Enter a guess for the code: 2
Close! your guess was too small, but you guessed correctly that the code is even

Enter a guess for the code: 4
you got it right!
```

1.3 cases

The main technique for figuring out the kinds of cases to include for conditional statements is truth tables. In the last example, we can figure out the cases using the following table

var > code	var even	var > code $\wedge$ var even
T	T	T
T	F	F
F	T	F
F	F	F

These are exactly the cases in order (the last row corresponds to the "else" in the conditional statement). In particular, row 3 corresponds to the case var < code && mod(var,2)==0 since if var > code is false, then that is the same as var < code (ignoring equality).

With that in mind, we can figure out what to write for the expressions with the following steps

1. Determine the conditions you want to consider (eg. var > code, var even)
2. Write the truth table for the statements including the $\wedge$ or $\vee$
3. negate the statements as necessary so that the truth value of the $\wedge$ or $\vee$ statements always have a truth value of T.

   eg. In the third row if we negate var > code, then the $\wedge$ has truth value T.
4. write these cases for the expressions in your code

Notice we didn't include the == case in the truth table, though we could.

var = code	var > code	var even	P = var > code $\wedge$ var even	P $\vee$ var = code
T	T	T	T	T
T	T	F	F	T
T	F	T	F	T
T	F	F	F	T
F	T	T	T	T
F	T	F	F	F
F	F	T	F	F
F	F	F	F	F

But, the first four cases are essentially the same, and the last four is the same as the first table. Basically, as soon as var = code, it doesn't matter what everything else is, so it's easy to just treat it as a separate (singular) case. Also, the first table is much easier to read!

2 For Loops

Remember Riemann sums? More specifically, how the indexing works, by that I mean the "i =" part under the sigma symbol

$$\sum_{i=1}^{n} i^2 - 1$$

So we read this as, when $i = 1$, calculate $i^2 - 1$ to be 0. When $i = 2$ calculate $i^2 - 1$ to be 3, and so on and so on. A for loop is basically the same idea! The difference is we can do more than just add together the things we calculate, so in a way for loops are even better. Here is a general setup

for i =*list of numbers*
    statement
end

Here is an example where we do exactly a Riemann Sum

```
>> Int = 0;
n = 6;
xi = 0:2/6:2;
Deltax = 2/6;

f = @(x) x^2-1;

for i = xi(1:length(xi)-1)
    Int = Int + f(i)*Deltax;
end

fprintf(['The left Riemann Sum approximation for the function x^2-1 \n' ...
    ' on the interval [0,2] with %1.0f subdivisions is %4.4f \n'], n, Int);
The left Riemann Sum approximation for the function x^2-1
 on the interval [0,2] with 6 subdivisions is 0.0370
```

Let's discuss some of the things used in the code. First off, at the beginning we started with "Int = 0". This is called *initializing* the variable "Int". We have to do this otherwise the computer won't know a variable "Int" exists, and it wouldn't just define it on its own during the for loop.

Also, notice that for the Riemann sum in the beginning of the section we can only have i be whole numbers, but in the for loop we could use a list of ANY numbers.

Just as a mathematical note, we wrote "i = xi(1:length(x)-1)". As usual, we wanted to have i take on the values of xi, but since we were calculating a LEFT Riemann sum, we shouldn't include the right endpoint. To exclude the right endpoint we had to stop 1 entry before the last, where the last entry would be the length(x) entry.

Just as a coding note, how about we make this code more general and user friendly. Namely, let's make it so that a user can run the script and choose the details of the Riemann sum.

```
>> f = input('Enter the function to approximate in terms of x: ');
n = input('Enter the number of subdivisions in the approximation: ');
min = input('Enter the left bound: ');
max = input('Enter the right bound: ');

Deltax = (max-min)/n;
xi = min:Deltax:max;

Int = 0;

for i = xi(1:length(xi)-1)
    Int = Int + f(i)*Deltax;
end
c = func2str(f);
fprintf(['The left Riemann Sum approximation for the function %s \n' ...
    ' on the interval [%3.3f,%3.3f] with %1.0f subdivisions is %.4f \n'],c, min, max, n, Int);
Enter the function to approximate in terms of x: @(x) x^3-1
Enter the number of subdivisions in the approximation: 20
Enter the left bound: -10
Enter the right bound: 10
The left Riemann Sum approximation for the function @(x)x^3-1
 on the interval [-10.000,10.000] with 20 subdivisions is -1020.0000
```

There's lots of other ways and situations in which we can use a for loop, as you will see in the exercises.

3 While Loops

Consider the following approximation problem

Suppose a drug is administered to a patient which is then metabolized by the patient's body. The concentration of the drug in the bloodstream is modeled by the differential equation

$$\frac{dC}{dt} = -kC$$

where k is a the metabolization constant (ie. how quickly or slowly this particular drug is metabolized)
This differential equation has solution $C(t) = A_0 e^{-kt}$ where t is measured in hours.
If a drug with metabolization constant $k = 0.67$ is administered with an initial dose of

$10g/ml$ how many hours does it take for the concentration to drop below $0.01g/ml$?

This is a relatively simple problem to solve when analytically using regular algebra, but this is for demonstration purposes!

Now, the easiest approach would be to just check values until the function has value less than 0.01. This *feels* like a for loop, since we have a list of numbers we want to calculate with. But we don't know HOW many numbers we need to check, that is what number do we choose for the "max" in $i = min : max$? Moreover, we want the for loop to stop once we've hit the bound 0.01, which makes it sound like we will need a conditional statement. The tool for this is a "While loop", which combines the power of conditional statements and for loops. here is the general setup

While *expression*
    statement
end

A "While loop" continues to loop while the expression is true and stops as soon as it's false. Then we can solve this problem in the following way:

```
>> A = 10;
k = 0.67;
Bound = 0.01;

f = @(t) A*exp(-k*t);

CheckVal = f(0);
hour = 0;

while CheckVal >= Bound
    hour = hour + 1;
    CheckVal = f(hour);
end

fprintf('After %1.0f hour(s) the concentration of the drug is below %1.2f \n', hour, Bound);
After 11 hour(s) the concentration of the drug is below 0.01
```

An important technique to point out is the use of the variable "hour". In a for loop we have the index to keep track of how many steps we've done, but a while loop doesn't have this implicitly. Instead we use something called a *counter*. So the variable "hour" is a counter, which counts the hours we've checked.

4 Functions and Structure

This section is not necessary for coding, in the sense that you can still calculate things without this technique. But it is necessary if you want to write usable code.

Consider the following problem

We want to model a "Guess the number" game between two friends. Your friend (the computer) will come up with a number between 1 and 10, your job is to guess what number your friend is thinking of.

Let's break this problem into a couple of pieces

1. The computer should come up with a number between 1 and 10

2. The player should be asked for a guess
3. There is a way to check if the guess is correct
    a) If its correct the game should end and display a congratulatory message
    b) If it's incorrect the game should continue and the user prompted for another guess and the game repeats.
4. Most importantly, we should also have a way of checking whether a guess has been made already and print a message if it has.
5. Second most importantly, check that the guess is within the correct range and print a message if it's not.

A key thing to notice is that there are two operations which may be repeated multiple times, namely, the prompting the user for a guess and the checking if the guess is correct.

To accomplish this we will used "Functions" (different from a function handle). Using functions we can call a block of code multiple times. Here is a set up

function[Y1,Y2,...,Ym] = *name of function*(X1,X2,...,Xn)
block of code
end

The variables X1,...,Xn are the variables that the function takes in, sort of the external data you want to give the function. These are used in the block of text. The variables Y1,...,Ym are the output for the function. If there is more than one output variable, you'll get back an array of outputs, with each entry of the array being the variables.

You can view a solution to the problem either on page 10 or you can view and download the .m file on Github. I suggest downloading the file and trying it out yourself!

Notice we used multiple functions with names that clearly indicate their role. Moreover, they were called multiple times, and sometimes even inside themselves. Looking at the function "CheckPrevGuess", at one point it calls itself. This is known as *nesting*. Be careful, sometimes nests go on forever if you don't have a way for it to stop!

You can also see that when a function has multiple outputs, the quick way to assign them is by writing an array with your variables in the order you want them assigned and setting that equal to the function call.

Don't worry too much about what the function itself does at each line, the point is to see that some problems require establishing an operation and utilizing it multiple times, and the way to do it is by functions.

```
Num = randi([1,10]);
Range = 1:10;
Past = [0];
check = 0;
IndP = 0;
IndR = 0;

while check == 0
    Guess = GetGuess();

    while IndR == 1
        [IndR, Guess, Past] = CheckRange(Range, Guess, Past);
    end

    [Guess, Past] = CheckPrevGuess(Past,Guess);
    check = CheckGuess(Num,Guess);
    Past = [Past Guess];
end

fprintf('You got it! The number was %1.0f \n', Num);

%=========================================%
function [I] = CheckGuess(N,g)
    I = g == N;
    if I == 0
        fprintf('That was not right. Try again. \n')
    end
end

%=========================================%
function [g] = GetGuess()
    g = input('Enter a number between 1 and 10 as your guess: ');
end

%=========================================%
function [I,g,P] = CheckRange(R,g,P)
    I = g < R(1) && g > R(length(R));

    while I == 1
        fprintf('Your guess was not in the correct range. \n')
        g = GetGuess();
        P = [P g];
        I = g < R(1) && g > R(length(R));
    end
end

%=========================================%
function [g,P,I] = CheckPrevGuess(P,g)
I = 0;
i = 1;
    while I == 0 && i <= length(P)
        I = P(i) == g;
        i = i + 1;
    end

    while I == 1
        fprintf('You have already guessed that, try again \n');
        g = GetGuess();
        [P,g,I] = CheckPrevGuess(P,g);
        P = [P, g];
    end
end

%=========================================%
```