Chapter 4

Branching Statement & Program Design
Top-Down Design

- It starts with a large task and breaks it down into smaller sub-tasks. Each sub-task may in turn be subdivided into even smaller building-blocks if needed.
- Each building-block can be coded and tested independently.
- Once all the building blocks have been verified to work properly, we combine the sub-tasks into a complete project.
- This is as known as **divide and conquer**.
Top-Down Design$^2$

Divide and Conquer
Program Design Process

Start

Define the problem

Define required inputs and outputs

Design the method

Convert method into MATLAB statements

Test the resulting MATLAB program

End

Decomposition

Stepwise refinement

Top-down design process
Program Testing Process

Start

Unit testing of individual subtasks
- Subtasks validated separately

Add subtasks into the main program
- Combined subtasks into one unit
- Repeat as many times as needed

Alpha release
- Major bugs fixed

Beta release
- Minor bugs fixed
- Repeat as many times as needed

End
Pseudo-Code

- To implement an algorithm, it should be described in a understandable form. The descriptions are called **constructs**.
- Each construct can be described in a special way called **pseudocode**.
- Pseudocode should describe the project in plain English, with a separate line for each distinct idea or segments of MATLAB code.

**For example:**

Convert degree in Fahrenheit (F) to degree in Kelvins (K) by $(F-32)*5/9+273.15$
True or False

- The **logical data type** is used to represent two possible concepts called logical values, **true** or **false**.
- Both logical values can be produced by functions **true** and **false**
  - **true** returns **1**
  - **false** returns **0**
True or False

- Logical values are stored in a single byte of memory.
- In an expression, zero represents false while non-zero represents true.
More Operators

- **Relational operators** are operators that compare two numbers and produce a true or false.
- **Logic operators** are operators that compare one or two numbers and produce a true or false.
- General form of a operation $\text{op}$ between the results of two expressions $e_1$ and $e_2$: $e_1 \text{ op } e_2$
## Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equal to</td>
</tr>
<tr>
<td><code>~=</code></td>
<td>not equal to</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
</tr>
</tbody>
</table>

- **Notice:**
  - `==` is a comparison operator.
  - `=` is an assignment operator.
Relational Operators

- Relational operators are element-by-element operations. For example:
  \[
  \begin{bmatrix}
  2 & 1 \\
  -1 & 4
  \end{bmatrix} > \begin{bmatrix}
  -1 & 3 \\
  0 & 2
  \end{bmatrix} = \begin{bmatrix}
  1 & 0 \\
  0 & 1
  \end{bmatrix}
  \]

- If one of the operand is a scalar, then the scalar will be used to compare each element of the array. For example:
  - 'test' > 'k' produces 1 0 1 1

- A string is an array of characters.
Roundoff Errors

- Due to **roundoff errors** during computer calculations, two theoretically equal numbers can differ slightly, causing an equality or inequality test to fail.

- For example:
  - \( \sin(\pi) \) and \( \sin(\pi) \)

```
>> sin(pi)
anans = 1.2246e-16
>> sin(pi)==0
ans = 0
```
Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND with shortcut evaluation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>logical exclusive OR</td>
</tr>
<tr>
<td>~</td>
<td>logical NOT</td>
</tr>
</tbody>
</table>

- The general form of a logical operation is `exp1 op exp2`
- Note that `&&` and `||` are scalar operators
Logical ANDs

- The result of an AND operator is **true** if and only if both operands are **true**.
- If at least one of the operands is **false**, the result is **false**.
- `&&` is an AND operator supports **short-cut evaluations**; i.e.
  - `&&` will returns a **false** if `exp1` is **false**.
  - In contrast, `&` always evaluates both `exp1` and `exp2` before returning an answer.
Logical ORs

- The result of an OR operator is **true** if at least one of the operands is **true**.
- If both operands are **false**, the result is **false**.
- `||` is an OR operator supports short-cut evaluations; i.e.
  - `||` will returns a **true** if `exp1` is **true**.
  - In contrast, `|` always evaluates both `exp1` and `exp2` before returning an answer.
Logical Exclusive ORs

- The result of a XOR operator is true if and only if one operand is true and the other is false.
## Truth Table

- **Truth Table for Logical Operators**

<table>
<thead>
<tr>
<th>inputs</th>
<th>and</th>
<th>or</th>
<th>xor</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>e₁</td>
<td>e₂</td>
<td>e₁&amp; e₂</td>
<td>e₁</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Hierarchy of Operations

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All arithmetic operators are evaluated in the order of previously described in Chapter 2.</td>
</tr>
<tr>
<td>2</td>
<td>All relational operators (==, ~==, &gt;, &gt;=, &lt;, &lt;=) are evaluated, working from left to right.</td>
</tr>
<tr>
<td>3</td>
<td>All ~ operators are evaluated.</td>
</tr>
<tr>
<td>4</td>
<td>All &amp; and &amp;&amp; operators are evaluated, working from left to right.</td>
</tr>
<tr>
<td>5</td>
<td>All</td>
</tr>
</tbody>
</table>

- For example:
  - 5>3 == 5>3 produces 0
  - 5>3 == 5<3 produces 1
Hierarchy of Operations

- It is important to follow the precedence to evaluate the outcome. You will be surprised if you only exam the expression by your instinct.

- For example:
  - $5 > 3 \equiv 5 > 3$ produces 0
  - $5 > 3 \equiv 5 < 3$ produces 1
Logical Functions

<table>
<thead>
<tr>
<th>function</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ischar(x)</td>
<td>Returns true if x is a character array.</td>
</tr>
<tr>
<td>isempty(x)</td>
<td>Returns true if x is an empty array.</td>
</tr>
<tr>
<td>isinf(x)</td>
<td>Returns true if x is Inf (infinite).</td>
</tr>
<tr>
<td>isnan(x)</td>
<td>Returns true if x is NaN (not-a-number).</td>
</tr>
<tr>
<td>isnumeric(x)</td>
<td>Returns true if x is a numeric array.</td>
</tr>
<tr>
<td>logical(x)</td>
<td>Convert numerical value x to logical value.</td>
</tr>
</tbody>
</table>

○ For all the data shown above only characters are not considered as numeric.

```
>> clear
>> A=[1/0,[],NaN,12.5,'xyz']
A =
     [Inf]  []  [NaN]  [12.5000]  'xyz'
>> for i=1:5;c(i)=isnumeric(A{i});end;disp(c)
1 1 1 1 0
```
Homework Assignment #8

- Quiz 3.1
  - Page 103: 1-21
- This assignment is due by 03/05.
- Late submission will not be accepted.
Flowcharts use special shapes to show different types of steps in a process.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start/end</td>
<td>An oval represents a start or end point</td>
</tr>
<tr>
<td></td>
<td>Arrows</td>
<td>A line is a connector that shows relationships between the representative shapes</td>
</tr>
<tr>
<td></td>
<td>Input/Output</td>
<td>A parallelogram represents input or output</td>
</tr>
<tr>
<td></td>
<td>Process</td>
<td>A rectangle represents a process</td>
</tr>
<tr>
<td></td>
<td>Decision</td>
<td>A diamond indicates a decision</td>
</tr>
</tbody>
</table>
Branches

- **Branches** are statements that allow us to select and execute specified sections of code (called blocks) while skipping other sections of code.

- Three constructs of branches:
  - **if**
  - **switch**
  - **try-catch**
if Statement

- The **if** construct has the form

```
if comparison1
    statements_block1
elseif comparison2
    statements_block2
else
    statements_block3
end
```

- where comparisons are logical expressions that control the operation of the **if** construct.
if Statement\(^2\)

- There can be any number of **elseif** clause (0 or more), but there can be at most one **else** clause.
Exercise 1

- The quadratic equation
  1. State the problem:
     Write a program to solve for the roots of a quadratic equation, $ax^2 + bx + c = 0$.
  2. Define the inputs and outputs:
     Inputs will be the coefficients $a$, $b$, and $c$. Outputs will be the roots of equations.
  3. Define the solution:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
Exercise 1²

- **Computational example**
  
  4. Evaluate the discriminant function:

\[ \Delta = b^2 - 4ac \]

  5. If \( \Delta > 0 \), then there are 2 distinct real roots. If \( \Delta = 0 \), then there is a single repeated root. If \( \Delta < 0 \), then there are 2 complex roots.
Exercise 1³

```matlab
% myroots.m

% get coefficients
coef=input('input an array [a b c]: ');

% compute the real part: \(-b/(2*a)\)
real_p=-coef(2)/(2*coef(1));

% compute the discriminant func \(b^2 - 4*a*c\)
disc=coef(2)^2-4*coef(1)*coef(3);

% determine whether there are repeated roots
if disc==0
    fprintf('x1 = x2 = %.f\n', real_p);
elseif disc < 0
    % compute imag part: \(\sqrt{-disc}/(2*a)\)
    imag_p = sqrt(-disc)/(2*coef(1));
    fprintf('x1 = %6.3f + %6.3fi\n',real_p+imag_p);
    fprintf('x2 = %6.3f - %6.3fi\n',real_p-imag_p);
else
    imag_p = sqrt(disc)/(2*coef(1));
    fprintf('x1 = %6.3f\n',real_p+imag_p);
    fprintf('x2 = %6.3f\n',real_p-imag_p);
end
```

Command Window

```
>> clear
>> edit myroots
>> myroots
input an array [a b c]: [1 4 4]
x1 = x2 = -2
>> myroots
input an array [a b c]: [1 2 3]
x1 = -1.000 + 1.414i
x2 = -1.000 - 1.414i
>> myroots
input an array [a b c]: [1 2 -3]
x1 = 1.000
x2 = -3.000
```
Exercise 2

○ Function of two variables
  ● 1. State the problem:
    Evaluate function \( f(x,y) \) which is defined as:
    \[
    f(x, y) = \begin{cases} 
    x + y & x \geq 0 \text{ and } y \geq 0 \\
    x + y^2 & x \geq 0 \text{ and } y < 0 \\
    x^2 + y & x < 0 \text{ and } y \geq 0 \\
    x^2 + y^2 & x < 0 \text{ and } y < 0 
    \end{cases}
    \]
  ● 2. Define the inputs and outputs:
    Inputs are values of \( x \) and \( y \).
    Outputs are \( f(x,y) \)
  ● 3. Evaluate \( f(x,y) \) based on 4-branched if
Exercise 2^2

```matlab
% evatest.m

% get inputs
val=input('input [x,y]: '); 
val=input([x,y]:
); x=val(1); y=val(2);

% evaluate 4 branches of if
if x>=0 && y>=0
    fx = x+y;
elseif x>=0 && y<0
    fx = x+y^2;
elseif x<0 && y>=0
    fx = x^2+y;
else
    fx = x^2+y^2;
end

fprintf('f(x,y) = %.3f\n', fx)
```

Command Window

```
>> clear
>> edit evatest
>> evatest
input [x,y]: [3,5]
f(x,y) = 8.000
>> evatest
input [x,y]: [2,-2]
f(x,y) = 6.000
>> evatest
input [x,y]: [-4,3]
f(x,y) = 19.000
>> evatest
input [x,y]: [-1,-2]
f(x,y) = 5.000
```
Nested if Statement

- Two **if** constructs are said to be nested if one of them is embed in another **if** block.
- It can be done by breaking the keyword **elseif** to **else; if**.

```plaintext
if comparison1
    statements_block1
    if comparison2
        statements_block2
        ...
    end
else
    statements_block4
end
```
Exercise 3

Assigning Letter Grades

State the problem:

Reads a numerical grade and assigns a letter grade in according to the following chart:

<table>
<thead>
<tr>
<th>Grade</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$94 &lt; \text{grade}$</td>
</tr>
<tr>
<td>B</td>
<td>$74 &lt; \text{grade} &lt; 95$</td>
</tr>
<tr>
<td>C</td>
<td>$64 &lt; \text{grade} &lt; 75$</td>
</tr>
<tr>
<td>D</td>
<td>$44 &lt; \text{grade} &lt; 65$</td>
</tr>
<tr>
<td>F</td>
<td>$0 &lt; \text{grade} &lt; 45$</td>
</tr>
</tbody>
</table>
Exercise 3^2
switch Statement

- The `switch` construct has the form

```
switch ctrl_expr
    case case_expr1
        statement_block1
    case case_expr2
        statement_block2
    ...
    otherwise
        statement_block3
end
```

- If the value of `ctrl_expr` is equal to the first `case_expr`, then the first block will be executed and the program will jump to the first statement following `end`. 
switch Statement

- The `ctrl_expr` and `case_expr` may be numeric or string values.
- If more than one `case_expr` should cause the same code to execute, then a combo case can be used.

    ```
    case {case_1, case_2, ...}
    ```
switch Statement\(^3\)

- The **switch** construct is another form of using cascaded **if** constructs.

```
if ctrl == case1
  statements_block1
elseif ctrl == case2
  statements_block2
...
else
  statements_block3
end
```
switch Statement

- **If**-statements and **switch**-statements are similar and sometimes interchangeable.
  - `switch val; case {1, 5, 7, 8}
  - `If (mod(val,2)==1)
  - Both **switch** and **if** have their own strong and weak points.
try-catch Statement

- The **try/catch** construct is designed to trap errors.
- The **try-catch** construct has the form

  ```
  try
  statement_block1
  catch
  statement_block2
  end
  ```

  - If an error occurs in the **try** block, then instead of aborting the execution, the code in the **catch** section will be executed.
try-catch Statement²

- Fatal errors or critical exceptions will be caught by **try-catch** construct. Such as:
  - access to undefined variables
  - array indexing overflow
  - failed to open a file

![Command Window]

```
>> clear
>> x+1
Undefined function or variable 'x'.
>> a=[1:2:6]
a =
    1   3   5
>> a(5)
Index exceeds matrix dimensions.
>> fd=fopen('data_test', 'r')
fd =
    -1
>> fread(fd,10,'int8')
Error using **fread**
Invalid file identifier. Use fopen to generate a valid file identifier.
```
try-catch Statement

```matlab
% catchest
% pre-requisite:
% a is an external array

try
    x = a(5);
    catch
        disp(['a is not defined', ...
             ' or is too short'])
        x = 0;
    end
end
```

Command Window:

```
>> clear
>> edit catchtest
>> x = x + 1
Undefined function or variable 'x'.
>> catchtest
a is not defined or is too short
x =
1
```
Example 4.5: Low Pass Filter

- Given a RC low pass filter:

  - \( V_i \) is the sinusoidal input voltage of frequency \( f \), \( R \) is a resistor in ohms, \( C \) is a capacitor in farads.
  - Evaluate the ratio of the output \( V_o \) to the input \( V_i \)

\[
\frac{V_o}{V_i} = \frac{1}{1+j2\pi fRC} \quad (4.3)
\]
Example 4.5²

- Plot the amplitude and frequency response of this filter over the frequency range $0 \leq f \leq 1000\text{Hz}$ as $R=16\text{k}\Omega$ and $C=1\mu\text{F}$.
- Equ. (4.3) indicates that the ratio $x$ is a complex number, so the amplitude and the phase can be computed as $|x| = \text{abs}(x)$ and $\text{angle}(x)$.
- Since the response can vary over a wide range, therefore we will use a loglog plot for the amplitude response and a semilogx plot for the phase response.
Example 4.5³

- Plot the amplitude and frequency response of this filter over the frequency $f$ of the output $V_o$ to the input $V_i$.
Example 4.6: Ideal Gas Law

- An ideal gas is defined as one in which all collisions between molecules are perfectly elastic; i.e. no kinetic energy lost.
  - The Ideal Gas Law:
    - $PV = nRT$
    - $R$ is $8314 \text{ L} \cdot \text{kPa/mol} \cdot \text{K}$
    - At $T=273 \text{ K}$, $n=1$ mole, estimate $V$ as $P$ varies from 1 to $1000 \text{ kPa}$ and plot $P$ versus $V$. 
Example 4.6\textsuperscript{2}

- Then $T$ is increased to 373 K. Plot $P$ versus $V$. 

```matlab
% gastest

% initialize nRT
n=1;
R=8.314;

% set pressure
P=1:50:1000;

% compute V=nRT/P and plot
T=273;
V=(n*R*T)./P;
loglog(P,V,'-m'); grid
title('P vs V')
xlabel('Pressure (kPa)')
ylabel('Volume (L)')
legend('T=273K', 'T=373K');

hold on
loglog(P,V,'-r.'); grid
T=373;
V=(n*R*T)./P;
loglog(P,V,'-b.'); grid
hold off
```
Homework Assignment #9

- 4.7 Exercises
  - Page 169: 4.1, 4.4, 4.5, 4.14
- This homework is due by next week.
- Late submission will be penalized.