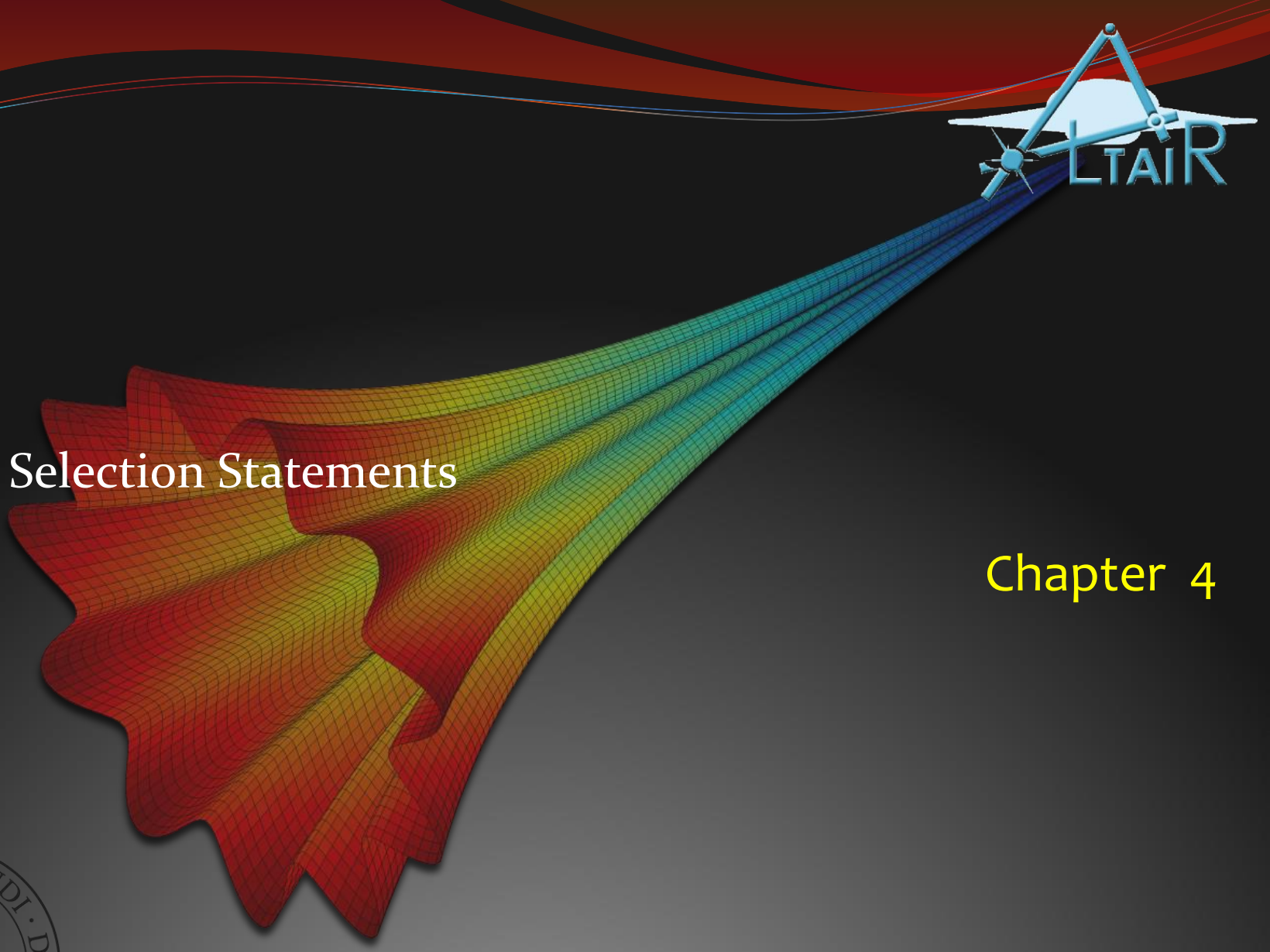


Selection Statements

Chapter 4

Linguaggio Programmazione Matlab-Simulink (2018/2019)

If Statement

- The **if** statement is used to determine whether or not a statement or group of statements is to be executed
- General form:

```
if condition
    action
end
```

- the *condition* is any relational expression
- the *action* is any number of valid statements (including, possibly, just one)
- if the condition is true, the action is executed – otherwise, it is skipped entirely

Representing true/false concepts

- Note: to represent the concept of false, `0` is used. To represent the concept of true, any nonzero value can be used – so expressions like 5 or 'x' result in logical true
- This can lead to some common logical errors
- For example, the following expressions are always true (because the "relational expressions" on the right, 6 and 'N', are nonzero so they are true; therefore, it does not matter what the results of the others are):

```
number < 5 || 6
letter == 'n' || 'N'
```

If-else Statements

- The **if-else** statement chooses between two actions
- General form:

```
if condition
    action1
else
    action2
end
```

- One and only one action is executed; which one depends on the value of the condition (action1 if it is logical true or action2 if it is false)

Throwing an error

- MATLAB has an **error** function that can be used to display an error message in red, similar to the error messages generated by MATLAB

```
if radius <= 0
    error('Sorry; %.2f is not a valid radius\n', radius)
else
    % carry on
end
```

Nested **if-else** Statements

- To choose from more than two actions, ***nested*** **if-else** statements can be used (an **if** or **if-else** statement as the action of another)
- General form:

```
if condition1
   action1
else
    if condition2
        action2
    else
        if condition3
            action3
        % etc: there can be many of these
        else
            actionn    % the nth action
        end
    end
end
```

The **elseif** clause

- MATLAB also has an **elseif** clause which shortens the code (and cuts down on the number of ends)
- General form:

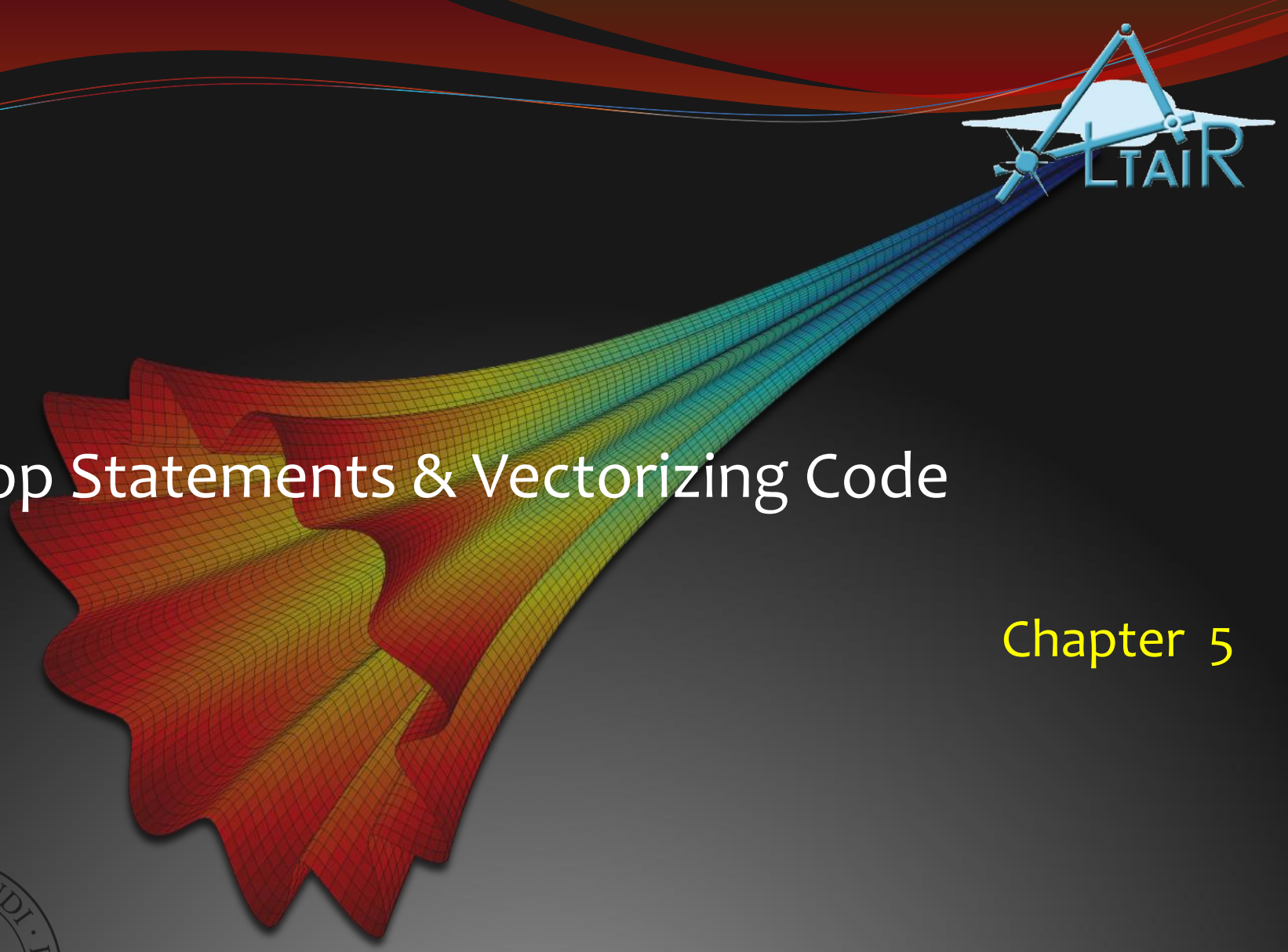
```
if condition1
   action1
elseif condition2
   action2
elseif condition3
   action3
% etc: there can be many of these
else
   actionn    % the nth action
end
```

The “is” functions

- There are many “is” functions in MATLAB that essentially ask a true/false question, and return logical 1 for true or 0 for false
- **isletter** returns 1 or 0 for every character in a string – whether it is a letter of the alphabet or not
- **isempty** returns 1 if the variable argument is empty, or 0 if not
- **iskeyword** returns 1 if the string argument is a keyword, or 0 if not
- **isa** determines whether the first argument is a specified bype

Programming Style Guidelines

- Use indentation to show the structure of a script or function. In particular, the actions in an **if** statement should be indented.
- When the **else** clause isn't needed, use an **if** statement rather than an **if-else** statement



Loop Statements & Vectorizing Code

Chapter 5

Linguaggio Programmazione Matlab-Simulink (2018/2019)

for loop

- used as a ***counted*** loop
- repeats an ***action*** a specified number of times
- an ***iterator*** or loop variable specifies how many times to repeat the action
- general form:

```
for loopvar = range
    Action
end
```

- the range is specified by a vector
- the action is repeated for every value of the loop variable in the specified vector

for loop examples

- Loop that uses the iterator variable:

```
>> for i = 1:3
        fprintf('i is %d\n', i)
    end
i is 1
i is 2
i is 3
```

- Loop that does not use the iterator variable:

```
>> for i = 1:3
        disp('Howdy')
    end
Howdy
Howdy
Howdy
```

Preallocating a Vector

- Preallocating sets aside enough memory for a vector to be stored
- The alternative, extending a vector, is very inefficient because it requires finding new memory and copying values every time
- Many functions can be used to preallocate, although it is common to use **zeros**
- For example, to preallocate a vector *vec* to have N elements:

```
vec = zeros(1,N);
```

for loop uses

- calculate a sum
  - initialize *running sum* variable to zero
- calculate a product
  - initialize *running product* variable to one
- input from user
  - can then *echo print* the input
- sum values in a vector
  - can also use built-in function **sum** for this
- other functions that operate on vectors: **prod**, **cumsum**, **cumprod**, **min**, **max**,**cummin**, **cummax**

For loop application: **subplot**

- The **subplot** function creates a matrix (or vector) in a Figure Window so that multiple plots can be viewed at once
- If the matrix is *m x n*, the function call **subplot(m,n,i)** refers to element *i* (which must be an integer in the range from 1 to m*n)
- The elements in the FW are numbered row-wise
- It is sometimes possible to use a **for** loop to iterate through the elements in the Figure Window

Subplot Example

- For example, if the subplot matrix is 2 x 2, it may be possible to loop through the 4 elements to produce the 4 separate plots

Plot 1	Plot 2
Plot 3	Plot 4

```
for i = 1:4
    subplot(2,2,i)
    % create plot i
end
```

Nested for loops

- A nested **for** loop is one inside of (as the action of) another **for** loop
- General form of a nested **for** loop:

```
for loopvarone = rangeone        <-- outer loop
    % actionone:
    for loopvartwo = rangetwo    <-- inner loop
        actiontwo
    end
end
```

- The inner loop action is executed in its entirety for every value of the outer loop variable

while loop

- used as a conditional loop
- used to repeat an action when ahead of time it is not known how many times the action will be repeated
- general form:

```
while condition
    action
end
```

- the action is repeated as long as the condition is true
- an *infinite loop* can occur if the condition never becomes false (Use Ctrl-C to break out of an infinite loop)
- Note: since the condition comes before the action, it is possible that the condition will be false the first time it is evaluated and therefore the action will not be executed at all

while loop application: error-checking

- with most user input, there is a valid range of values
- a **while** loop can be used to keep prompting the user, reading the value, and checking it, until the user enters a value that is in the correct range
- this is called *error-checking*
- general form of a while loop that error-checks:

```
prompt user and input value
while value is not in correct range
    print error message
    prompt user and input value
end
use value
```

Example: Prompt for radius

```
radius = input('Enter the radius of a circle: ');
while radius <= 0
    radius = input('Invalid! Enter a positive radius: ');
end
area = pi * radius ^ 2;
fprintf('The area is %.2f\n', area)
```

While loop example (*Practice 5.6)*

```
% Error checks until the user enters n positive integers
n = 4;
for i = 1:n
   inputnum = input('Enter a positive integer: ');
   num2 = int32(inputnum);
   while num2 ~= inputnum || num2 < 0
      inputnum = input('Invalid! Enter a positive integer: ');
      num2 = int32(inputnum);
   end
   fprintf('Thanks, you entered a %d \n',inputnum)
end
```

for loops and vectors

- **for** loops can be used to accomplish the same task for every element in a vector
- general form of **for** loop that iterates through a vector:

```
for i = 1:length(vectorvariable)
    do something with vectorvariable(i)
end
```

- if the purpose of the loop is to create a vector variable, it is much more efficient to *preallocate* the variable before the loop (note: the length must be known)

Nested for loops and matrices

- *nested* **for** loops can be used to accomplish the same task for every element in a matrix
- one loop is over the rows, and the other is over the columns
- general form of nested **for** loop that iterates through a matrix:

```
[r c] = size(matrixvariable)
for row = 1:r
    for col = 1:c
        do something with matrixvariable(row,col)
    end
end
```

- Note: this nested loop iterates through the matrix row-by-row; by reversing the for statements it would instead iterate column-by-column

Use MATLAB wisely!!

- Using **for** loops with vectors and matrices is a very important programming concept, and is necessary when working with many languages
- However… Although **for** loops are very useful in MATLAB (e.g., for the **subplot** function), they are almost ***NEVER*** necessary when performing an operation on every element in a vector or matrix!
- This is because MATLAB is written to work with matrices (and therefore also vectors), so functions on matrices and operations on matrices automatically iterate through all elements – no loops needed!

Vectorizing

- The term ***vectorizing*** is used in MATLAB for re-writing code using loops in a traditional programming language to matrix operations in MATLAB
- For example, instead of looping through all elements in a vector vec to add 3 to each element, just use scalar addition:

```
vec = vec + 3;
```

Efficient Code

- In most cases, code that is faster for the programmer to write in MATLAB is also faster for MATLAB to execute
- Keep in mind these important features:
  - Scalar and array operations
  - Logical vectors
  - Built-in functions
  - Preallocation of vectors

Preallocation Question

- Preallocation can speed up code, but in order to preallocate it is necessary to know the desired size. What if you do not know the eventual size of a vector (or matrix)? Does that mean that you have to extend it rather than preallocating?

Preallocation Answer

- If you know the maximum size that it could possibly be, you can preallocate to a size that is larger than necessary, and then delete the "unused" elements. In order to do that, you would have to count the number of elements that are actually used. For example, if you have a vector *vec* that has been preallocated, and a variable *count* that stores the number of elements that were actually used, this will trim the unnecessary elements:
- vec = vec(1:count)

Operations on Vectors & Matrices

- Can perform numerical operations on vectors and matrices, e.g. vec + 3
- Scalar operations e.g. mat * 3
- Array operators operate term-by-term or element-by-element, so must be same size
- Addition + and subtraction -
- Array operators for any operation *based on* multiplication require dot in front .* ./ .\ .^

Useful Efficient functions

- Keep in mind these useful functions:
  - **sum, prod, cumsum, cumprod, min, max**
  - **any, all, find**
  - **diff**
  - "is" functions including **isequal**

- **checkcode**: can check code in both scripts and functions for inefficiencies; same as information in Code Analyzer Reports

Timing Code

- The functions **tic** and **toc** are used to time code
  - Be careful; other processes running in the background will have an effect so should run multiple times and average

```
>> type fortictoc

tic
mysum = 0;
for i = 1:20000000
    mysum = mysum + i;
end
toc

>> fortictoc
Elapsed time is 0.090699 seconds.
>>
```

- There is also a Profiler that will generate detailed reports on execution times of codes

Common Pitfalls

- Forgetting to initialize a running sum or count variable to `0` or a running product to 1
- Not realizing that it is possible that the action of a **while** loop will never be executed
- Not error-checking input into a program
- Forgetting that **subplot** numbers the plots rowwise rather than columnwise.
- Not taking advantage of MATLAB; not vectorizing!

Programming Style Guidelines

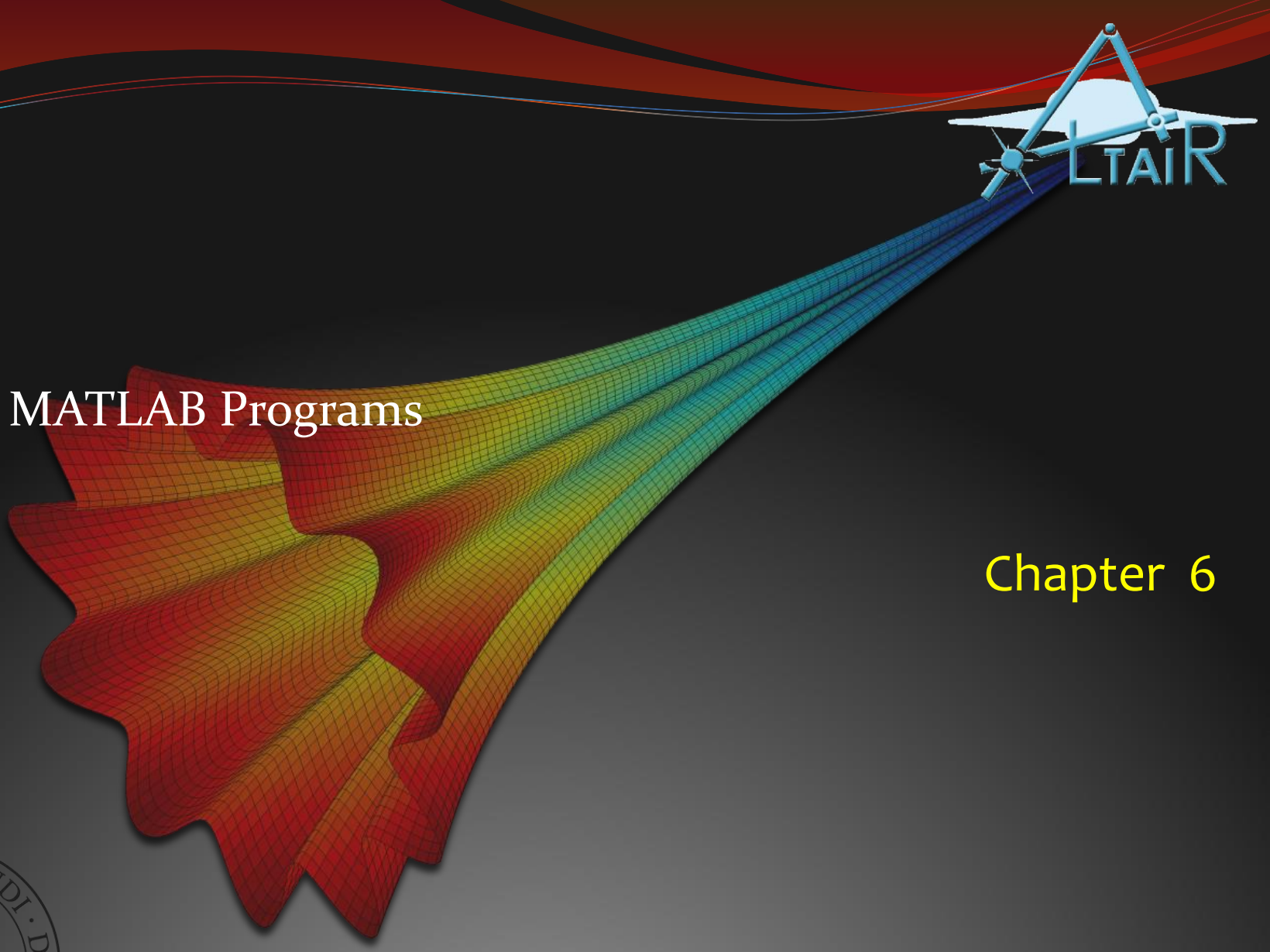
- Use loops for repetition only when necessary
  - **for** statements as counted loops
  - **while** statements as conditional loops
- Do not use *i* or *j* for iterator variable names if the use of the built-in constants **i** and **j** is desired.
- Indent the action of loops
- Preallocate vectors and matrices whenever possible (when the size is known ahead of time).
- If the loop variable is just being used to specify how many times the action of the loop is to be executed, use the colon operator 1:n

Exercises

- Write a **for** loop that will print a column of five *'s.
- Write a function *mymatmin* that finds the minimum value in each column of a matrix argument and returns a vector of the column minimums.
- Write a script *avenegnum* that will repeat the process of prompting the user for negative numbers, until the user enters a zero or positive number. Instead of echo-printing them, however, the script will print the average (of just the negative numbers). If no negative numbers are entered, the script will print an error message instead of the average.

Exercises

- Write a function that imitates the **cumprod** function. Use the method of preallocating the output vector. (Hint: use help **cumprod** first)**.**
- Create a function *matrowsum* to calculate and return a vector of all of the row sums of a matrix, instead of column sums (**sum** function in Matlab returns the column sums)
- Implement vectorized versions of the previous functions and scripts that includes **tic toc** commands to test the efficency of the code.



MATLAB Programs

Chapter 6

Linguaggio Programmazione Matlab-Simulink (2018/2019)

Types of Functions

- Categories of functions:
  - functions that calculate and return one value
  - functions that calculate and return more than one value
  - functions that just accomplish a task, such as printing, without returning any values
- They are different in:
  - the way they are called
  - what the function header looks like
- All are stored in code files with the extension .m

Generic Function Definition

- All function definitions consist of:
  - The function header
    - The reserved word **function**
    - Output arguments and the assignment operator (only if the function returns value(s)
    - Function name and input arguments
  - A block comment describing the function
  - The body of the function which includes all statements, including putting values in all output arguments, if there are any
  - **end**

Functions that Return >1 Value

- General form of a function that returns more than one value; it has multiple output arguments in the header
- The output arguments are separated by commas

functionname.m

```
function [output arguments] = functionname(input arguments)
% Comment describing the function
Statements here; these must include putting values in all
of the output arguments listed in the header
end
```

Calling the function

- Since the function is returning multiple values through the output arguments, the function call should be in an assignment statement with multiple variables in a vector on the left-hand side (the same as the number of output arguments in the function header) in order to capture all of them
- Otherwise, some will be lost

Example Function Call

- For example, if the function header is:

  function [x,y,z] = fnname(a,b)

- This indicates that the function is returning 3 things, so a call to the function might be (assuming a and b are numbers):

  [g,h,t] = fnname(11, 4.3);

- Or using the same names as the output arguments (it doesn't matter since the workspace is not shared):

  [x,y,z] = fnname(11, 4.3);

- This function call would only get the first value returned:

  result = fnname(11, 4.3);

A function *tworan* that returns two random integers, each in the range from 10 to 20

tworan.m

```
function [ranx, rany] = tworan
ranx = randi([10,20]);
rany = randi([10,20]);
end
```

Example Function call:

```
[x, y] = tworan
```

A function *tworanb* that receives two integer arguments a and b and returns two random integers, each in the range from a to b

tworanb.m

```
function [ranx, rany] = tworanb(a,b)
ranx = randi([a,b]);
rany = randi([a,b]);
end
```

Example Function call:

```
[x, y] = tworanb(5, 50)
```

Functions that do not return anything

- A function that does not return anything has no output arguments in the function header, nor does it have the assignment operator
- The statements in the body would typically display or plot information from the input arguments

functionname.m

```
function functionname(input arguments)
% Comment describing the function
  statements here
end
```

Calling a function with no output

- Since no value is returned, the call to such a function is a statement
- For example, if this is the function header:

```
function fnname(x,y)
```

- A call to the function might look like this:

```
fnname(x,y)
```

- This would NOT be a valid call; since the function is not returning anything, there is no value to assign:

```
result = fnname(x,y);  % Invalid!
```

A function *prttworan* that prints two random integers, each in the range from 10 to 20

prttworan.m

```
function prttworan
fprintf( 'One is %d\n' , randi([10,20]))
fprintf( 'The other is %d\n' , randi([10,20]))
end
```

Example Function call:

```
prttworan
```

A function *prttworanb* that receives two integer arguments a and b and prints two random integers, each in the range from a to b

prttworanb.m

```
function prttworanb(a,b)
fprintf( 'One is %d\n' , randi([a,b]))
fprintf( 'The other is %d\n' , randi([a,b]))
end
```

Function call:

```
prttworanb(5,50)
```

Notes on Functions

- You do not always have to have input arguments to a function. If you do not, you can have (both in the function header and in the function call) empty (), or you can just leave them out
- The function header and function call have to match up:
  - the name has to be the same
  - the number of input arguments must be the same
  - the number of variables in the left-hand side of the assignment should be the same as the number of output arguments
  - if there are no output arguments, the function call is a statement
- Functions that return values do not normally print them, also – that is left to the calling function/script

Subfunctions

- When one function calls another, the two functions can be stored in the same code file with the same name as the primary function

primary.m

```
primary function header
    primary function body includes call to subfunction
end
subfunction header
     subfunction body
end
```

- The subfunction can only be called by the primary function

Example: Modular outline

- In a modular program, a script calls functions
- Given the following script (where x,y,z are 3 things)

```
[x,y,z] = getinputs;
result = calcstuff(x,y,z);
displayit(x,y,z, result)
```

- With just that information, we can write the corresponding function headers (not the definitions, just the headers)

Example function headers

- function [x,y,z] = getinputs
- function result = calcstuff(x,y,z)
- function displayit(x,y,z, result)

Types of Errors

- *Syntax errors*: mistakes in language e.g. missing quote at the end of a string
- *Run-time* (or execution-time) errors: errors that are found during execution of a script or function, e.g. referring to an element in a vector that does not exist
- *Logical errors*: mistakes in reasoning e.g. using an expression like (0 < x < 10)

Debugging Methods

- There are several methods that can be used to find errors:
  - *Tracing*: using the **echo** statement which will show all statements as executed
  - Using MATLAB's Editor/Debugger
  - Set breakpoints so values of variables/expressions can be examined at various points
    - **dbstop** sets a breakpoint
    - **dbcont** continues execution
    - **dbquit** quits debug mode

Code Cells and Publishing

- Code in scripts can be broken into sections called ***code cells***
- You can run one code cell at a time
- Code cells are created with comments that start with two %%
- Code in code cells can also be published in HTML format with plots embedded and with formatted equations
- Do this from the Publish tab in the Editor

Exercises

- Write a function *perimarea* that calculates and returns the perimeter and area of a rectangle. Pass the length and width of the rectangle as input arguments.
- Write a function that receives a vector as an input argument and prints the individual elements from the vector in a sentence format.
- Write a function that will prompt the user for a string of at least one character, loop to error-check to make sure that the string has at least one character, and return the string.

Exercises

- For a right triangle with sides *a*, *b*, and *c*, where *c* is the hypotenuse and θ is the angle between sides a and c, the lengths of sides *a* and *b* are given by:

$$a = c * \cos(\theta)$$

$$b = c * \sin(\theta)$$

Write a script *righttri* that calls a function to prompt the user and read in values for the hypotenuse and the angle (in radians), and then calls a function to calculate and return the lengths of sides *a* and *b*, and a function to print out all values in a sentence format.

Exercises

- Modify the *readradius* function to error-check the user's input to make sure that the radius is valid. The function should ensure that the radius is a positive number by looping to print an error message until the user enters a valid radius.

Exercises

- The following script is bad code in several ways. Use **checkcode** first to check it for potential problems, and then use the techniques described in this section to set breakpoints and check values of variables.

debugthis.m

```
for i = 1:5
   i = 3;
   disp(i)
end

for j = 2:4
   vec(j) = j
end
```