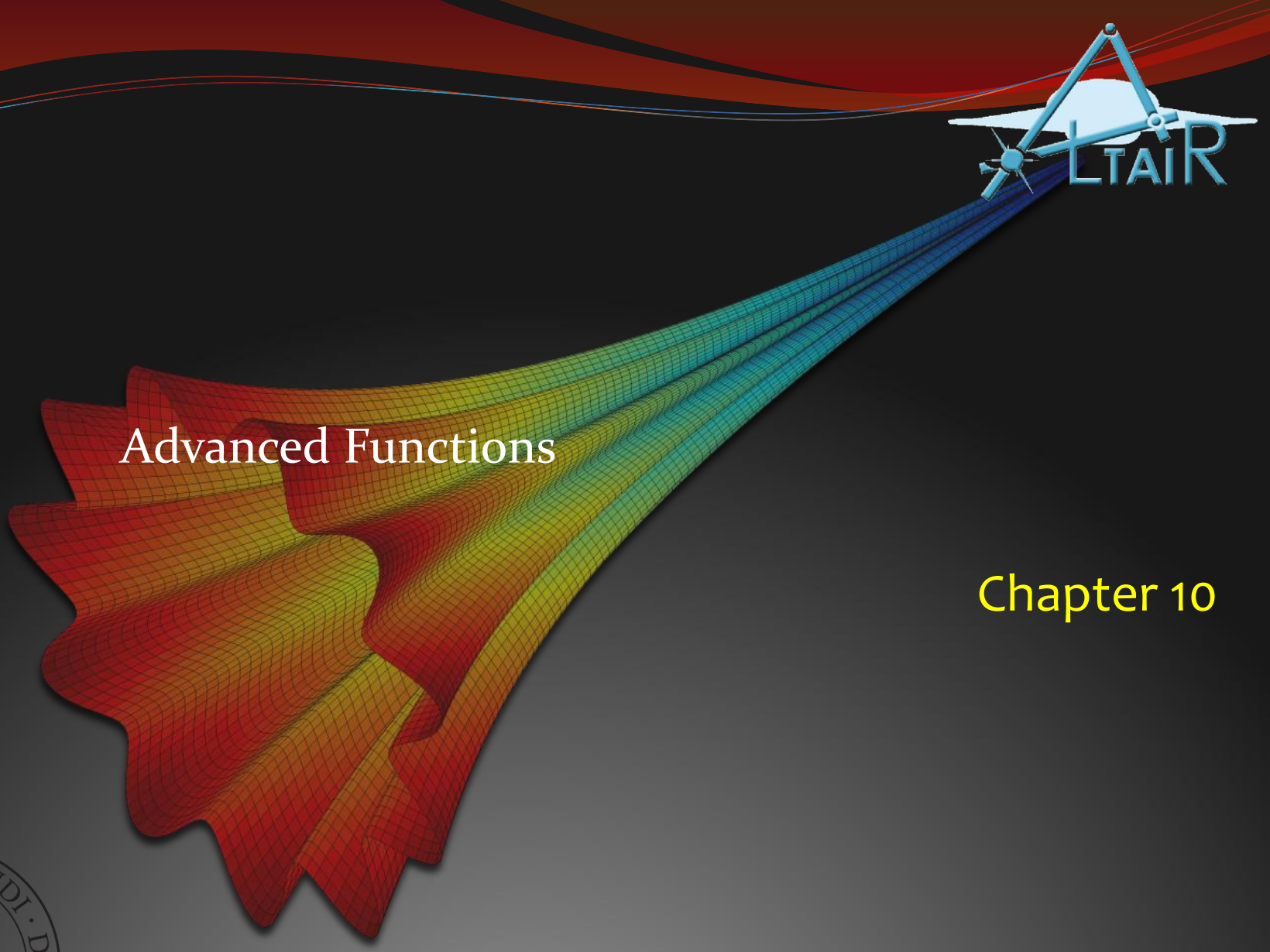

Advanced Functions

Chapter 10

Linguaggio Programmazione Matlab-Simulink (2018/2019)

Variable # of Arguments

- So far in the functions that we've written, there has been a fixed number of input arguments and a fixed number of output arguments
- It is possible to have a ***variable number of arguments***, both input arguments and output arguments
- A built-in cell array **varargin** can be used to store a variable number of input arguments
- a built-in cell array **varargout** can be used to store a variable number of output arguments
- These are cell arrays because the arguments could be different types, and cell arrays can store different kinds of values in the different elements.

Variable # of input arguments

- The cell array **varargin** stores a variable number of input arguments
  - This could be used to store all of the input arguments to the function, or only some of them
- The function **nargin** returns the *total* number of input arguments that were passed to the function (not just the length of **varargin**)
- Since **varargin** is a cell array, use curly braces to refer to the elements, which are the input arguments

Variable input function headers

- Two kinds of function headers
  - All input arguments stored in **varargin**:

```
function outarg = fnname(varargin)
```

  - **varargin** stores some of the input arguments but not all:

```
function outarg = fnname(input args, varargin)
```

Function header example

- For example, if coordinates of a point are being passed to a function, and we know x and y will always be passed, and z might, there are two possibilities:

  function outarg = fnname(varargin)

  - In this case, x is stored in varargin{1}, y is stored in {2}, and if z is passed, it is in varargin{3}

  function outarg = fnname(x,y,varargin)

  - In this case, x and y are stored in input arguments x and y, and if z is passed, it is in varargin{1}
- Note: in both cases, nargin will be 3

Variable # of output arguments

- The cell array **varargout** stores a variable number of output arguments
- As with input arguments, some output arguments can be built in if they are always going to be returned, or **varargout** can store all output arguments – so there are two kinds of function headers:

```
function varargout = fnname(input args)
function [output args, varargout] = fnname(input args)
```

- Since **varargout** is a cell array, use curly braces to refer to the elements, which are the output arguments
- To call this function:

```
[variables] = fnname(input args);
```

Function **nargout**

- The function **nargout** returns the number of output arguments expected from the function (e.g., the number of variables in the vector in the left-hand side of the assignment statement when calling the function)
- For example, if the left side of the assignment statement in which the function is called is:

  [x, y, z] = fnname(…

  - the value of **nargout** would be 3

Examples

```
function acceptVariableNumInputs(varargin)
    disp("Number of input arguments: " + nargin)
    celldisp(varargin)
end

>>acceptVariableNumInputs(ones(3),'some text',pi)
```

Examples

```
function definedAndVariableNumInputs(X,Y,varargin)
    disp("Total number of input arguments: " + nargin)
    [r c] = size (varargin);
    fprintf("Size of varargin cell array: %dx%d\n", r,c);
end
```

```
>> definedAndVariableNumInputs(7,pi,rand(4),datetime('now'),'hello’)
>>  definedAndVariableNumInputs(13,42)
```

Examples

```
function varargout = variableNumInputAndOutput(varargin)
    disp(['Number of provided inputs: ' num2str(length(varargin))])
    disp(['Number of requested outputs: ' num2str(nargout)])

    for k = 1:nargout
        varargout{k} = k;
    end
end
```

```
>> [d,g,p] = variableNumInputAndOutput(6,'Nexus')
```

```
>> variableNumInputAndOutput
```

Examples

```
function varargout = redplot(varargin)
    [varargout{1:nargout}] = plot(varargin{:},'Color',[1,0,0]);
end
```

```
>> x = 0:pi/100:2*pi;
y = sin(x);
redplot(x,y)
```

```
>> h = redplot(x,y,'Marker','o','MarkerEdgeColor','green');
```

Recursive functions

- Recursion is when something is defined in terms of itself
- Recursive functions are functions that call themselves
  - There has to be a way to stop this, otherwise, infinite recursion will occur
- Sometimes either iteration or recursion can be used to implement a solution to a problem

Factorial Example

- An iterative definition for the factorial of an integer n is:

  $n! = 1 * 2 * 3 * \ldots * n$

- A recursive definition is:

  $n! = n * (n - 1)!$ general case

  $1! = 1$ base case

- With a recursive definition, there is always a general case which is recursive, but also a base case that stops the recursion

Code for recursive factorial

- A function that implements the recursive definition has an **if-else** statement to choose between the general and base cases:

```
function facn = fact(n)
% fact recursively finds n!
% Format: fact(n)
if n == 1
    facn = 1;
else
    facn = n * fact(n-1);
end
end
```

Common Pitfalls

- Thinking that **nargin** is the number of elements in **varargin** (it may be, but not necessarily; **nargin** is the total number of input arguments)
- Forgetting the base case for a recursive function

Programming Style Guidelines

- If some inputs and/or outputs will always be passed to/from a function, use standard input arguments/output arguments for them. Use **varargin** and **varargout** only when it is not known ahead of time whether other input/output arguments will be needed.
- Use iteration instead of recursion when possible.