vg101: Introduction to Computer Programming

RC 1

CHEN Xiwen

2019/5/17

Outline

- Lectures
  - Computer basics
  - Program design
  - Learning a programming language
    - Data
    - Operations (indexing, calculations, etc.)
      - Visualization
    - I/O
  - MATLAB
- Practice

Lectures

Computer Basics

1. Number system.

Number System	Description
Binary	base 2, digits used: 0, 1
Octal	base 8, digits used: 0 ~ 7
Hexadecimal	base 16, digits used: 0 ~ 9, A ~ F

2. Base conversion.
   - Convert base m to base n:

$$N = m_1 \times m^{l_m-1} + m_2 \times m^{l_m-2} + \cdots + m_{l_m}$$

$$l_n = \lceil \log_n(N+1) \rceil$$

$$n_k = \left\lfloor \frac{N}{n^{l_n-k}} \right\rfloor \mod n, \qquad \text{for } k = 1, \ldots, l_n$$

- Examples:
  - Convert $(100011)_2$ from binary to hexadecimal.

$$N = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 = 35$$
$$l_n = \lceil \log_{16}(36) \rceil = 2$$
$$n_1 = \left\lfloor \frac{35}{16} \right\rfloor \mod 16 = 2, \qquad n_2 = 35 \mod 16 = 3$$

    Therefore, $(100011)_2 = (23)_{16}$.
  - Convert $5F3EC6$ from hexadecimal to binary.

$$(5F3EC6)_{16} = (0101\ 1111\ 0011\ 1110\ 1100\ 0110)_2$$

3. Algorithm.

```
IN: input of the algorithm
OUT: output of the algorithm
RECIPE: what the algorithm computes to obtain the output
COMPLEXITY: in terms of space/time consumption
```

Program Design

> 1. Clearly state or translate the problem
> 2. Define what is know as the *input*
> 3. Define what is know as the *output*
> 4. Develop an *algorithm*, i.e., a systematic way to solve the problem
> 5. Verify the solution on simple input
> 6. Implementing the algorithm

Learning a Programming Language

READ-THE-DOC!

- What can we learn from a documentation?
  1. Data type (integers, double, arrays, chars, etc.) and operations
  2. Functional APIs (input, output, options) and descriptions
- API (abstract programming interface)

```
output = funcName(arg1, arg2, ..., options)
    arg1: description of arg1
    arg2: description of arg2
    ...
    options: available options for the function
```

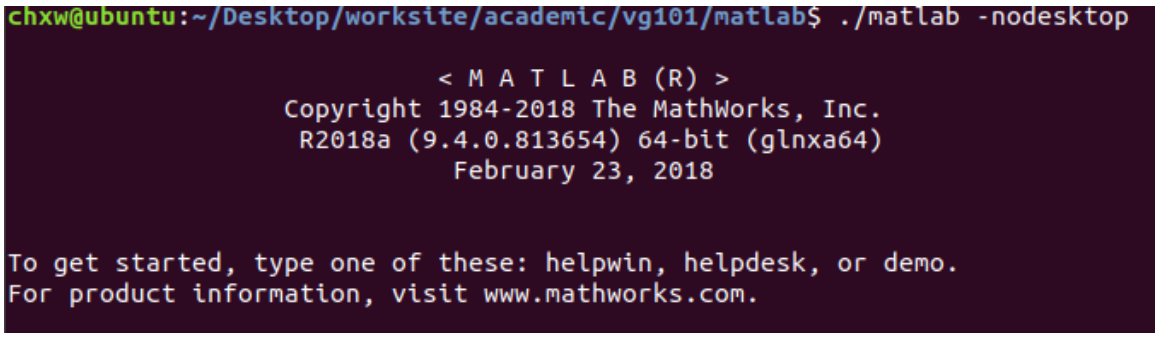
MATLAB Inerface

- Modes:
  - Desktop: graphical interface
    1. Current folder: *default file I/O path, available functions*
    2. Workspace: *current variables and their values*

3. Command window: *View immediate results*
4. Editor: *edit scripts*

- Terminal **(optional)**

1. Start MATLAB from terminal: `$./matlab -nodesktop`



2. Exit MATLAB from terminal: `>> exit`
3. Run MATLAB scripts in terminal mode: `run('path-to-script/file.m')`, or with error handling: `try, run('path-to-script/file.m'), catch e, fprintf('%s\n', e.message), end`
4. To use a MATLAB function that we write:
    1. Add path: `addpath(path-to-function)`
    2. Use function as in command window in desktop mode: `function(args)`

- Command window:
    - Clear command history: `clc`
    - View command history: ↑
    - Clear a certain variable in workspace: `clear varName`
    - Clear all variables in workspace: `clearvars`
    - Query the existence of a variable or a file: `exist varName/fileName`
- Workspace:
    - Access current variable value from command window: `var`
    - Current variables are not cleared when starting running a script: **avoid variable conflicts by clearing the workspace using `clear all;` in the first line of the script**
    - Save variables to `fileName.mat` or load variables from `fileName.mat` to workspace: `save fileName` and `load fileName`
- Editor:
    - Place to define your functions and write your scripts.
    - If the `.m` file is a function definition, the name should be consistent with the function name (e.g., `function myFunc(n)` is defined in `myFunc.m`), one main function definition per file, but can contain sub-functions that are only accessible inside the file
    - Function vs. script:
        - Function: with API (inputs, outputs), can be called within another script or function
        - Script: a series of tasks, with possible function calls
    - Printed results from running the script will be shown in the command window: variable assignment that ends with `;` will not print the value in the command window **(Please end with a `;` for every unnecessary output)**
- Help and documentation:

    - Directly search using keywords
    - In the command window: `help funcName` or `doc keyWord`
- Debugging: *add breakpoints (basics)*
    1. Add a breakpoint by clicking the dash beside a line number
    2. Run the script
    3. The process will stop at the breakpoint, you can view the intermediate values for variables in the workspace
    4. Continue running the script

Scripting in MATLAB

- Coding style
    - `Tab` or `Space` to improve readability
    - End with `;` to suppress unnecessary outputs and begin the following lines with a new line
    - Space after operators, e.g., `a = 2;` instead of `a=2`
    - Space after a `,` or `;`, e.g., `A = [1, 2; 3, 4]` instead of `A = [1,2;3,4]`
    - Use `...` for continuation in the next line, e.g.,

```
if ((a == 2 && b < 3) || (a == 3 && b > 5) ...
            || a == 4)
```

    - You might want to **properly** align for variable assignments, e.g.,

```
myArray = [1, 2, 3];
myInt   = 7;
```

    - Use `%` for comment;
    - Variable/function names: *begin with a character, case sensitive*
        1. Use meaningful names: `publicKey` or `public_key`, `getVerifyData()` or `get_verify_data()` for functions
        2. Avoid built-in names (variables that already have meanings, e.g., `ans`) in some situations
- Variable types
    1. Numeric types: `double` (default), `single`, `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`
    2. Characters: `char` (use ASCII code: `char(65) = 'A'`, `char('A' + 1) = 'B'`),

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

3. Strings: `string`
    1. `strcmp(s1, s2)`: compare `s1` and `s2`
    2. `strrep(str, old, new)`: replace `old` substring with `new`
    3. `strfind(str, subStr)`: return the index of the `subStr` in `str`
4. Logical: **in case of integers, only `0` evaluates to `False`, e.g.**

```
% a = 4;
if -2
    a = 4;
else
    a = 3;
end
```

```
% a = 3;
if 0
    a = 4;
else
    a = 3;
end
```

5. Arrays & matrices
    - Initialization:
        1. `A = [1, 2, 3; 4, 5, 6; 7, 8, 9];`
        2. `A = [1:4];`, with default step 1, same as `A = [1, 2, 3, 4];`
        3. `A = [1:3:7];`, with step 3, same as `A = [1, 4, 7];`
        4. `A = linspace(start, end, num);`

5. `A = zeros(col, row);`
6. `A = ones(col, row);`
7. `A = magic(n);`: $n \times n$ magic matrix

- Indexing:
  1. `A(k)`, `A(r, c)`, **starting from 1**

     (**Q**: what does `A(n + 3)` return when `A` is an $n \times n$ matrix?)
  2. `A(r, :)` to access a row, `A(:, c)` to access a column
  3. `A(r1:r2, :)` to access `r1` to `r2` inclusive, `A(:, c1:c2)` to access `c1` to `c2` inclusive

     (**Q**: How about `A(r1:r2, c1:c2)`?)
  4. Block assignment:

     ```
     A = magic(4);
     B = magic(2);
     A(2:3, 2:3) = B;
     ```
  5. Exchange rows/columns: `A([r1, r2], :) = A([r2, r1], :)` / `A(:, [c1, c2]) = A(:, [c2, c1])`
  6. `A(:)` to access all elements in matrix `A` as an array (column-first)
- Operations:
  1. Element-wise addition: `A + B` (`A` and `B` should have a consistent dimension)
  2. Element-wise multiplication: `A .* B`
  3. Matrix multiplication: `A * B`
  4. Element-wise division: `A ./ B` (`A` and `B` should have a consistent dimension)
  5. Concatenation: `[A, B]` (`A` and `B` should have a consistent row dimension), `[A; B]` (`A` and `B` should have a consistent column dimension)

     (**Q**: What does `A = [1:4]; A = [0, A];` do?)
  6. Reshape:

     ```
     A = [1:30];
     A = reshape(A, 5, 6);
     ```
  7. Transpose, inverse, eigenvalues, determinant: `A'`, `inv(A)`, `eig(A)`, `det(A)`
  8. Flip up and down, left and right: `flipud(A)`, `fliplr(A)`
  9. Number of array elements: `numel(A)`
  10. Sum and mean:
      1. `sum(A, 1)` / `mean(A, 1)`: sum up/average all rows of the columns
      2. `sum(A, 2)` / `mean(A, 2)`: sum up/average all columns of the rows
      3. `sum(A(:))` / `mean(A(:))`: sum up/average all elements in `A`

6. Type conversion: `newVar = type(var)`, e.g.,

```
a = 1.2; % double, 1.2000;
a = int8(a); % a = 1, unsigned integer;
```

- Control flow

  1. `if-else`

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

  2. `switch-case`

```
switch variable
    case value1
        statements1
    case value2
        statements2
    otherwise
        statements3
end
```

  3. `while`

```
while expression
    statements
end
```

  4. `for`

```
for i = start:increment:end
    statements
end
```

  5. `break` and `continue`

     `break` : jump out of the current loop

     `continue` : jump to the next iteration in the current loop

     Q: What is the value of `A` in the following two cases?

```
A = zeros(10);
for i = 1:10
    for j = 1:10
        if i == 3 && j == 4
            break;
        end
        A(i, j) = 1;
    end
end
```

```
A = zeros(10);
for i = 1:10
    for j = 1:10
        if i == 3 && j == 4
            continue;
        end
        A(i, j) = 1;
    end
end
```

- Function definitions: *inputs, outputs, operations*

```
% myFunc.m file;
function [out1, out2, ...] = myFunc(in1, in2, ...)

    % operations defined here;
    ...
    out1 = subFunc(in1); % all the output values should be defined;
    ...
    out2 = subFunc(in2); % all the output values should be defined;
    ...

function out = subFunc(in)
    out = ...;
```

- Function handles:
  1. Usage:

     > 1. pass a function to another function
     > 2. specify call back functions
     > 3. construct handles to functions defined inline instead of stored in a program file
     > 4. call local functions from outside the main function

  2. Defining a function handle:
     1. `f = @(x) x.^2 + 1;`
     2. Via an existing function: `f = @computeSquare`

```
function y = computeSquare(x)
    y = x.^2;
end
```

3. Call function: `f(arg)`

Practices

1. Where is Waldorf?

Write a script that reads from a file `input.txt`. The first line of the file indicates the number of rows r and columns c in the following matrix of letters, separated by a space. The following r lines of the file content is a $r \times c$ matrix of letters. Then the next line includes the number of words w in the rest of the file, followed by w words that need to be found. For instance, a sample input file might look like the following:

> `input.txt`
>
> ```
> 8 11
> a b c D E F G h i g g
> h E b k W a l D o r f
> F t y A w a l d O R k
> F t s i m r L q s r c
> b y o A r B e D e y v
> K l c b q w i k o m k
> s t r E B G a d h r b
> y U i q l x c n B j f
> 4
> Waldorf
> Bambi
> Betty
> Dagbert
> ```

Then the script finds the position of the first letter of a word in the form of `(r, c)` that is found in the matrix. The word can be in horizontal, vertical and diagonal in both directions (left and right), case insensitive. Output the results in the file `output.txt` in the format of `[word] is located at (r, c) in the [direction] direction` (direction: `right/left/up/down/up right/up left/down right/down left`), line separated. For instance, the output of the input file above should be the following:

> `output.txt`
>
> ```
> Waldorf is located at (2, 5) in the right direction
> Bambi is located at (2, 3) in the down right direction
> Betty is located at (1, 2) in the down direction
> Dagbert is located at (7, 8) in the left direction
> ```

For simplicity, the word in the input file is guaranteed to appear at least once in the matrix, and you only need to find one of the locations if multiple solutions possible.

2. Minesweeper.

Write a function `myMinesweeper(r, c, n)` which takes the number of rows $r > 3$ and columns $c > 3$ as inputs, and randomly generate a $r \times c$ truth board, with 0 indicating safe position, and 1 indicating a mine in that position, with a total of n mines. Start from an empty board containing only 0s. Then until the user wins (with all mines identified without touching them) or loses (touched a mine), the user is prompted to input two numbers `r c` indicating a guess (space separated). The the number in the position is updated with the number of mines within a 3×3 grid centered at the guessed location. If the game ends, the user should be prompted with either `Congratulations, you win!`, or `Sorry, you lose!`. For each intermediate round, the updated board should be printed in the command window.