L17: Midterm Review

Fundamental Concepts of Programming

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

February 27, 2017

Version: release

Announcements

Lab 06 is due on March 3 at 12 pm (noon)

Today:

- Midterm review

Wednesday:

- Midterm exam – **Bring your student ID and a scantron!**

Friday:

- Reading and writing data from/to disc (Chapter 10)
- Last lecture on Part 1 of the class

Need to reduce chatting and noise during lecture!

- Noise is distracting for others

Tentative list of topics

- Arithmetic expressions
- Logical expressions
- Operator precedence
- Variable assignment
- Fundamental data classes
  - `double`
  - `logical`
  - `char`
- More data structures: `cell` and `struct` arrays
- Indexing
- Array versus matrix operations
  - Size requirements
- Functions
  - Input arguments
  - Output arguments
  - Separate workspaces
  - Recursion
  - `return`
  - Anonymous functions
  - Function handles
- Control flow
  - `if`-statements (branching)
  - `for` and `while` loops, `break`, `continue`
  - `try`/`catch`
- Binary representations
  - Integers (3 representations)
  - Floating points (IEEE-754)
  - Characters

Operator precedence

Things in parentheses	↑ higher precedence
Exponentiation (.^, ^); transpose (')	
Logical negation (~)	
Multiplication (.*, *); Division (./, /)	
Addition (+); Subtraction (-)	
Colon (*e.g.*, `1:10`)	
Relational operators (<, <=, >, >=, ==, ~=)	
Element-wise logical "and" (&)	
Element-wise logical "or" (|)	
Short-circuit logical "and" (&&)	
Short-circuit logical "or" (||)	↓ lower precedence

Full list at https://www.mathworks.com/help/matlab/matlab_prog/operator-precedence.html

Fundamental data classes

```
>> % The default numerical class is double
>> class(1)
ans =
double

>> % Other fundamental data classes are: logical
>> a = [2, -3, 0];
>> class(a >= 0)
ans =
logical

>> % and char
>> s = 'Hello!';
>> class(s)
ans =
char
```

Indexing works the same way with arrays of class `double`, class `logical`, and class `char`

Indexing

Indexing: accessing elements in arrays

Use parentheses to index elements in arrays of class `double`, `logical`, `char`, and `struct`

`cell` arrays:

- Use curly braces to index a single element
- Use parentheses to index multiple elements (get a `cell` array)

Types of indices:

- (row, column) index
- Single index (linear indexing, column by column)
- Array of linear indices
- Array of logicals (logical indexing)
- Colon (`:`) to access all the elements along a dimension
- Keyword `end` to access the last element
- Can index relative to `end` (*e.g.*, `end-2`, `end+1`)

Indexing: examples

```
>> a = [2, -1, 0; -3, 4, 9]
a =
     2    -1     0
    -3     4     9

>> % (row,column) index
>> a(2, 1)
ans =
    -3

>> % Linear indexing (single index)
>> a(3)
ans =
    -1

>> % Linear indexing (array of linear indices)
>> a([3, 4, 6])
ans =
    -1     4     9
```

Indexing: examples

```
>> a = [2, -1, 0; -3, 4, 9]
a =
     2    -1     0
    -3     4     9

>> % Indexing using a colon (:)
>> a(1,:)
ans =
     2    -1     0

>> % Logical indexing, example 1)
>> a([true, true, false; false, false, true])
ans =
     2
    -1
     9
>> % Logical indexing, example 2)
>> a(a < 0 | a > 5)
ans =
    -3
    -1
     9
```

Indexing: examples

```
>> v = [2, -1, 0, -3, 4, 9]
v =
     2    -1     0    -3     4     9

>> % Removing an element from a vector
>> v(2) = []
v =
     2     0    -3     4     9

>> % Example of using the keyword end
>> v(end-2)
ans =
    -3
>> % Another example: adding a value to a vector
>> v(end+1) = 6
v =
     2     0    -3     4     9     6
>> % Another example: reversing a vector
>> v(end:-1:1)
ans =
     6     9     4    -3     0     2
```

Array versus matrix operations

Element-wise operations: +, -, .*, ./, .^

- Between a scalar and an array: always possible
- Between two non-scalar arrays: arrays must have the same size

```
>> a = [2, -1, 0; -3, 4, 9];
>> b = [1, 8, 3; -7, -1, 0];
>> c = [0; -2; 7];

>> 7 .* a.^2
ans =
    28     7     0
    63   112   567

>> a .* b
ans =
     2    -8     0
    21    -4     0

>> % "a ./ c" yields error: Matrix dimensions must agree.
```

Array versus matrix operations

Matrix multiplication: *

- Between a scalar and a matrix: always possible
- Between two non-scalar matrices: inner dimensions must agree

```
>> a = [2, -1, 0; -3, 4, 9];
>> b = [1, 8, 3; -7, -1, 0];
>> c = [0; -2; 7];

>> 7 * a
ans =
    14    -7     0
   -21    28    63

>> a * c
ans =
     2
    55

>> % "a * b" yields error: Inner matrix dimensions must agree.
```

Array versus matrix operations

Matrix exponentiation: ^

- Between a square matrix and a scalar only

```
>> a = [4, 5, 1; -1 -7, -8; 0, 0, 5]
a =
     4     5     1
    -1    -7    -8
     0     0     5

>> a^3      % Note: a.^3 yields a different result
ans =
    59   160   -24
   -32  -293  -274
     0     0   125

>> % Would yield errors:
>> % a^a (because one of the operands must be a scalar)
>> % a(1,:)^3 (because the matrix must be square)
```

Functions

```
function [x, y] = my_whats_this_workspace(a, b)
output = 10;
x = a;
y = b;
return
y = a + b;
end
```

```
>> a = 2; b = 3; output = 1;
>> [y, x] = my_whats_this_workspace(a, b)
y =
     2
x =
     3
>> [b, a] = my_whats_this_workspace(y, x)
b =
     2
a =
     3
>> output
output =
     1
```

Functions

```
function [x, y] = my_whats_this_workspace(a, b)
output = 10;
x = a;
y = b;
return
y = a + b;
end
```

```
>> a = 2; b = 3; output = 1;
>> [y, x] = my_whats_this_workspace(a, b)
y =
     2
x =
     3
>> [b, a] = my_whats_this_workspace(y, x)
b =
     2
a =
     3
>> output
output =
     1
```

- Functions use their own separate workspace
- Functions communicate with their caller's workspace through input and output (I/O) arguments
- When calling a function, it is not the names of the variables passed as I/O arguments that matter, it is the order in which they are passed

Anonymous functions and function handles

Function handle: variable (class: `function_handle`) that contains a reference to a function. One can call the function using the handle's name as if it were the name of the function

Anonymous function: function defined in one line (gives a handle) inside the code, without a function header (as opposed to: function defined in a separate m-file with a header and everything)

```
>> % Create a function handle
>> % to an existing function
>> h = @max;
>> class(h)
ans =
function_handle
>> h([2, 4, 5])
ans =
     5
>> h([1, -5, -4])
ans =
     1
```

```
>> % Define and use an
>> % anonymous function
>> f = @(x) max(x, 10);
>> class(f)
ans =
function_handle
>> f(3)
ans =
    10
>> f(15)
ans =
    15
```

Recursion

- **A recursive function is a function that calls itself**

Recursion

- **A recursive function is a function that calls itself**
- **Often, the input arguments to the recursive calls are "reduced", "simplified", or "half-processed" versions of the original inputs**

Recursion

- **A recursive function is a function that calls itself**
- **Often, the input arguments to the recursive calls are "reduced", "simplified", or "half-processed" versions of the original inputs**

  For example, see the solution for `my_calculator_inverse_precedence` (lab 04): (`my_calc` for short)

  - `my_calc('3+5*2')`:

    ```
    result = my_calc('3+5') * my_calc('2')
    ```

Recursion

- **A recursive function is a function that calls itself**
- **Often, the input arguments to the recursive calls are "reduced", "simplified", or "half-processed" versions of the original inputs**

  For example, see the solution for `my_calculator_inverse_precedence` (lab 04): (`my_calc` for short)

  - `my_calc('3+5*2')`:

    ```
    result = my_calc('3+5') * my_calc('2')
    ```

- **A recursive function needs a base case** (often: smallest possible problem that the function can solve) that must be resolved without recursion. A recursive function will call itself indefinitely if there is no base case

  From previous example:
  `my_calc('2')` $\rightarrow$ base case (use `str2num('2')`)

Branching (if statements)

- For a given `if`-statement, only the first clause which condition is satisfied is "activated"
- Sometimes, none of the conditions are satisfied, in which case none of the clauses are "activated"

```
>> a = 5;
>> if a > 0
>>      a = a + 2;
>> elseif a > 2
>>      a = a + 25;
>> end

>> a
a =
     7
```

```
>> a = 5;
>> if a > 0
>>      a = a + 2;
>> end
>> if a > 2
>>      a = a + 25;
>> end

>> a
a =
    32
```

```
>> a = -2;
>> if a > 0
>>      a = a + 2;
>> elseif a > 2
>>      a = a + 25;
>> end

>> a
a =
    -2
```

Examples of for loops

Example of a "running sum":

```
function [approx] = my_sin_approx_fixed(x, n)

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 1.1 of Lab 04.
%
% Version: release.

approx = x;
for i = 1:n
    approx = approx + (-1)^i * x^(2*i+1) / factorial(2*i+1);
end

end
```

$$\sin(x) \approx \sum_{i=0}^{n} \frac{(-1)^i x^{2i+1}}{(2i+1)!}$$

Examples of for loops

Example: Find the minimum of a vector and the corresponding index

```
function [minimum, index] = my_minimum_index(vector)

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 2.1 of Lab 04.
%
% Version: release.

index = 1;
minimum = vector(1);

for i = 2:numel(vector)
    if vector(i) < minimum | isnan(minimum) & ~isnan(vector(i))
        index = i;
        minimum = vector(i);
    end
end

end
```

Examples of for loops

The “looping array” does not have to be a row vector of equally spaced values such as `1:10`

- I recommend to always use a row vector for the looping array
- Using something other than equally spaced values can be useful

Example where the “looping array” contains non-equally spaced values:

```
prime_numbers = [2, 3, 5, 7, 11];
sum_of_primes = 0;
for number = prime_numbers
    sum_of_primes = sum_of_primes + number;
end
```

Examples of for loops

The "looping array" does not have to be a row vector of equally spaced values such as `1:10`

- I recommend to always use a row vector for the looping array
- Using something other than equally spaced values can be useful

Example where the "looping array" contains non-equally spaced values:

```
prime_numbers = [2, 3, 5, 7, 11];
sum_of_primes = 0;
for number = prime_numbers
    sum_of_primes = sum_of_primes + number;
end
```

After executing the code above, the value of the variable `sum_of_primes` will be $2 + 3 + 5 + 7 + 11 = 28$

while loops

while loop: repeat a piece of code while a condition is true

Example: "hang" for approximately 10 seconds:

```
wait_time = 10;
tic;
while toc() < wait_time
end
```

Ways to end a `while` loop:

- The condition checked by the `while` loop becomes false
- `break`
- `return` (terminates the execution of the entire function)
- Raise an error inside the loop and catch it outside of the loop

For loop versus while loop

When to use a `for` loop versus a `while` loop?

For loop versus while loop

When to use a `for` loop versus a `while` loop?

If you know the number of steps you will need:

- Use a `for` loop

For loop versus while loop

When to use a `for` loop versus a `while` loop?

If you know the number of steps you will need:

- Use a `for` loop

If you know the maximum number of steps you may need, but you may need fewer steps than that:

- Use a `for` loop with a `break`, or a `while` loop

For loop versus while loop

When to use a `for` loop versus a `while` loop?

If you know the number of steps you will need:

- Use a `for` loop

If you know the maximum number of steps you may need, but you may need fewer steps than that:

- Use a `for` loop with a `break`, or a `while` loop

If you do not know the number of steps that you will need:

- Use a `while` loop

More control flow: break, continue, return

- `break` terminate early the current `for` or `while` loop
- `continue`: terminate early the current iteration of a `for` or `while` loop
- `return`: terminate early the execution of the current function

A given n-bit binary representation can represent at most 2^n different numbers

Binary representation of data

A given n-bit binary representation can represent at most 2^n different numbers

Representing integers (you should know these three representations):

- **Unsigned**: positive integers only, only one way to represent 0
- **Sign-magnitude**: positive and negative integers, two different ways to represent 0
- **Two's complement**: positive and negative integers, only one way to represent 0

Binary representation of data (continued)

Floating point numbers (IEEE-754 standard):

- Format akin to scientific notation: $\text{sign} \times (1 + f) \times 2^{\text{exponent}}$
  - The first bit represents the sign
  - The following bits represent the magnitude
  - The remaining bits represent the fractional part (significant figures)
- **The gap between consecutive representable numbers increases with the magnitude of the numbers**
- Accuracy is high around small numbers and low around big numbers **but relative accuracy is high for all numbers**
- Matlab's function `eps` measures the gap around a given number

Binary representation of data (continued)

Floating point numbers (IEEE-754 standard):

- Format akin to scientific notation: $\text{sign} \times (1 + f) \times 2^{\text{exponent}}$
  - The first bit represents the sign
  - The following bits represent the magnitude
  - The remaining bits represent the fractional part (significant figures)
- **The gap between consecutive representable numbers increases with the magnitude of the numbers**
- Accuracy is high around small numbers and low around big numbers **but relative accuracy is high for all numbers**
- Matlab's function `eps` measures the gap around a given number

Representing characters:

- Associate a numerical code to each character (*e.g.*, ASCII, unicode)
- Often, these numerical codes are integers only
- Represent these numerical codes in binary format

Resources

- Lectures: slides, diaries, scripts, and functions
- Textbook
- GSIs in lab section
- My office hours
- Lab assignments and solutions
- Past midterm and final exams
- bcourses Pages:
  - List of required functions
  - Common error messages
  - Specific topics (*e.g.*, & versus && and | versus ||)
  - ...