E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Lab Assignment 04: Iteration and Recursion
Version: release

Due date: Friday February 17th 2017 at 12 pm (noon).

General instructions, guidelines, and comments:

- For each question, you will have to write and submit one or more Matlab functions. We provide a number of test cases that you can use to test your function. The fact that your function works for all test cases provided does not guarantee that it will work for all possible test cases relevant to the question. It is your responsibility to test your function thoroughly, to ensure that it will also work in situations not covered by the test cases provided. During the grading process, your function will be evaluated on a number of test cases, some of which are provided here, some of which are not.
- Submit on bCourses one m-file for each function that you have to write. The name of each file must be the name of the corresponding function, with the suffix `.m` appended to it. For example, if the name of the function is `my_function`, the name of the file that you have to submit is `my_function.m`. **Carefully check the name of each file that you submit.** Do not submit any zip file. If you re-submit a file that you have already submitted, bCourses may rename the file by adding a number to the file's name (*e.g.,* rename `my_function.m` into `my_function-01.m`). This behavior is okay and should be handled seamlessly by our grading system. Do not rename the file yourself as a response to this behavior.
- A number of optional Matlab toolboxes can be installed alongside Matlab to give it more functionality. All the functions that you have to write to complete this assignment can, however, be implemented without the use of any optional Matlab toolboxes. We encourage you to not use optional toolboxes to complete this assignment. All functions of the Matlab base installation will be available to our grading system, but functions from optional toolboxes may not. If one of your function uses a function that is not available to our grading system, you will loose all points allocated to the corresponding part of this assignment. To guarantee that you are not using a Matlab function from an optional toolbox that is not available to our grading system, use one or both of the following methods:
  - ◇ Only use functions from the base installation of Matlab.
  - ◇ Make sure that your function works on the computers of the 1109 Etcheverry Hall computer lab. All the functions available on these computers will be available to our grading system.

- For this assignment, the required submissions are:
  - `my_sin_approx_fixed.m`
  - `my_sin_approx_tolerance.m`
  - `my_minimum_index.m`
  - `my_sort.m`
  - `my_reverse_without_recursion.m`
  - `my_reverse_with_recursion.m`
  - `my_parser.m`
  - `my_calculator_inverse_precedence.m`
  - `my_calculator_with_undo.m`

1. Approximation of the sine function

The following equality holds for any real number x:

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

It is virtually impossible to calculate all the terms of the sum in the right-hand side of Equation 1. Instead, one often calculates only the first few terms of this sum to obtain an approximate value of $\sin(x)$, according to the formula:

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

where n is either zero or a positive integer.

1.1. Approximation with a fixed number of terms

Write a function with the following header:

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

where:

- `x` is a scalar of class `double` that represents the value of x in Equation 2. You can assume that `x` is neither `NaN`, `Inf`, nor `-Inf`.
- `n` is a scalar of class `double` that represents the value of n in Equation 2. You can assume that `n` is either zero or a positive integer.
- `approx` is a scalar of class `double` that represents the approximation of $\sin(x)$ according to Equation 2.

Test cases:

```
>> approx = my_sin_approx_fixed(0, 0)
approx =
     0

>> approx = my_sin_approx_fixed(2*pi/3, 2)
approx =
    0.8990

>> approx = my_sin_approx_fixed(2*pi/3, 4)
approx =
    0.8661

>> approx = my_sin_approx_fixed(5*pi/2, 3)
approx =
 -189.6141

>> approx = my_sin_approx_fixed(5*pi/2, 10)
approx =
    1.0135
```

1.2. Approximation with a specified tolerance

Write a function with the following header:

```
function [approx, n] = my_sin_approx_tolerance(x, tolerance)
```

where:

- `x` is a scalar of class `double` that represents the value of x in Equation 2. You can assume that `x` is neither `NaN`, `Inf`, nor `-Inf`.
- `tolerance` is a scalar of class `double` that is greater than zero and that is neither `NaN` nor `Inf`.
- `n` is a scalar of class `double` that represents the smallest possible value of n such that the approximation of $\sin(x)$ as calculated by Equation 2 is within `tolerance` of the value calculated by Matlab using `sin(x)`.
- `approx` is a scalar of class `double` that represents the approximation of $\sin(x)$ as calculated by Equation 2, where n is equal to `n`.

Note: Consider three real numbers a, b, and ϵ such that $\epsilon > 0$. "a is within ϵ of b" if and only if $|a - b| \leqslant \epsilon$.

Test cases:

```
>> [approx, n] = my_sin_approx_tolerance(pi/3, 1e-2)
approx =
    0.8663
n =
     2
```

```
>> [approx, n] = my_sin_approx_tolerance(4*pi/5, 1e-3)
approx =
    0.5884
n =
     4

>> [approx, n] = my_sin_approx_tolerance(pi/3, 1e-10)
approx =
    0.8660
n =
     6

>> [approx, n] = my_sin_approx_tolerance(11*pi/2, 1e-5)
approx =
   -1.0000
n =
    26
```

2. Sorting

2.1. Minimum of a vector and its index

Write a function with the following header:

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

where:

- `vector` is a row vector of class `double` that contains at least one value.
- `minimum` is a scalar of class `double` that represents the minimum value of `vector`.
- `index` is a scalar of class `double` that represents the index of the first occurrence in `vector` of the minimum value of `vector`.

If all the values in `vector` are `NaN`, then `minimum` should have the value `NaN` and `index` should have the value `1`. In any other case, `NaN` should never be considered the minimum value of `vector`.

You may **not** use Matlab's built-in functions `min`, `max`, `find`, and `sort` in this question.

Test cases:

```
>> [minimum, index] = my_minimum_index(5)
minimum =
     5
index =
     1

>> [minimum, index] = my_minimum_index([5, 6, 1])
minimum =
```

```
     1
index =
     3

>> [minimum, index] = my_minimum_index([5, 6, 1, -1, 1])
minimum =
    -1
index =
     4

>> [minimum, index] = my_minimum_index([4, 10, NaN, 2, 5, 50])
minimum =
     2
index =
     4
```

2.2. Sorting a row vector

Write a function with the following header:

```
function [sorted] = my_sort(vector)
```

where:

- `vector` is a (possibly empty) row vector of class `double`.
- `sorted` is a row vector of class `double` that has the same size as `vector` and that contains the same values as `vector`, but ordered in increasing order.

If present in `vector`, `NaN` values should be placed at the end of `sorted`. You may **not** use Matlab's built-in functions `min`, `max`, `find`, and `sort` in this question.

Hint: You can remove the element of index `i` from a row or column vector `vector` of class `char`, `double`, or `logical`, by using the syntax: `vector(i) = []`.

Test cases:

```
>> [sorted] = my_sort([0])
sorted =
     0

>> [sorted] = my_sort([2, 1])
sorted =
     1     2

>> [sorted] = my_sort([2, 1, 9, -10, pi])
sorted =
  -10.0000    1.0000    2.0000    3.1416    9.0000

>> [sorted] = my_sort([6, 3, 7, 1, 0, 1, 7])
sorted =
```

```
     0     1     1     3     6     7     7

>> [sorted] = my_sort([6, 3, 7, 1, NaN, 0, 1, 7])
sorted =
     0     1     1     3     6     7     7   NaN
```

3. Character string manipulation

In this question, you will manipulate character strings, which can also be seen as row vectors of characters.

The concept of "stacks" may help you implement some of the functions required for this question. We call a "stack" a type of data structure where only the last element added to the data structure can be accessed. In order to access the second-to-last element added to a stack, one has to remove the last element from this stack first. Access to data in the stack is therefore described as "last-in-first-out" (LIFO), since the last element added to the stack is necessarily the one that is removed from the stack first. Two applications of stacks are:

- Reversing a character string.
- Undoing operations (for example in a text editor).

In this problem, you will write functions that parse character strings. Although using stacks is not required to implement any of the functions for this question, it may facilitate the implementation of some of these functions.

3.1. Reversing a vector without using recursion

Write a function with the following header:

```
function [reversed] = my_reverse_without_recursion(vector)
```

where:

- `vector` is a (possibly empty) row or column vector of class `char`, `double`, or `logical`.
- `reversed` is a vector that has the same size and class as `vector`, and that contains the same values as `vector`, but in reverse order.

You may **not** use Matlab's built-in functions `flip`, `fliplr`, and `flipud` in this question. Additionally, **you must not use recursion to implement this function.**

Hints:

- You can remove the element of index `i` from a row or column vector `vector` of class `char`, `double`, or `logical`, by using the syntax: `vector(i) = []`.
- You can create an empty array of class `logical` using the syntax: `logical([])`. You can use a similar syntax to create empty arrays of class `char` and `logical`.

Test cases:

```
>> reversed_char = my_reverse_without_recursion('Hello E7!')
reversed_char =
!7E olleH

>> reversed_logical = my_reverse_without_recursion([true; true; false; true])
reversed_logical =
  4x1 logical array
   1
   0
   1
   1

>> reversed_double = my_reverse_without_recursion(0:2:10)
reversed_double =
    10     8     6     4     2     0
```

3.2. Reversing a vector using recursion

Write a function with the following header:

```
function [reversed] = my_reverse_with_recursion(vector)
```

This function should have the same functionality as `my_reverse_without_recursion`, but you must implement `my_reverse_with_recursion` **using a recursive algorithm**.

You may **not** use Matlab's built-in functions `flip`, `fliplr`, and `flipud` in this question.

Test cases:

```
>> reversed_char = my_reverse_with_recursion('Hello E7!')
reversed_char =
!7E olleH

>> reversed_logical = my_reverse_with_recursion([true; true; false; true])
reversed_logical =
  4x1 logical array
   1
   0
   1
   1

>> reversed_double = my_reverse_with_recursion(0:2:10)
reversed_double =
    10     8     6     4     2     0
```

3.3. Character string parsing

Character string parsing is the operation of splitting apart strings of text into useful segments that a computer can make use of. Character string parsing is often useful to interpret textual

input read from a file or typed by a user. Character string parsing is analogous to splitting apart a sentence into each of its component words. The characters used to separate distinct expressions from one another are called delimiters. Examples of common delimiters are the space character and the newline character.

Write a function with the following header:

```
function [delimiter, left, right] = my_parser(string, delimiters)
```

where:

- `string` is a (possibly empty) row vector of class `char`.
- `delimiters` is a row vector of class `char`, which contains at least one value, and where each character is a distinct delimiter.
- `delimiter` is a 1 by 1 array of class `char` whose value should be the first delimiter (among the delimiters specified by `delimiters`) that appears in `string`.
- `left` is a row vector of class `char` whose value is the part of `string` located to the left of the first delimiter that appears in `string` (excluding the delimiter itself).
- `right` is a row vector of class `char` whose value is the part of `string` located to the right of the first delimiter that appears in `string` (excluding the delimiter itself).

If none of the delimiters specified by `delimiters` are present in `string`, then `left` should have the value of `string`, and both `right` and `delimiter` should be empty character strings. It is possible for `string` to start with one of the delimiters. In this case, `left` should be an empty character string.

You may **not** use Matlab's built-in functions `split`, `strsplit`, `find`, and `strfind` in this question.

Test cases:

```
>> [delimiter, left, right] = my_parser('Hello+World', '+')
delimiter =
+
left =
Hello
right =
World

>> [delimiter, left, right] = my_parser('Another-test', ',-')
delimiter =
-
left =
Another
right =
test

>> [delimiter, left, right] = my_parser('NO DELIMITER??', '!()')
```

```
delimiter =
  0x0 empty char array
left =
NO DELIMITER??
right =
  0x0 empty char array
```

3.4. The order of operations has been changed!

In this question, you will write a function that calculates the result of an arithmetic expression involving scalars, where the arithmetic expression is given as a character string (*e.g.,* `'2*30+5^4'`). The operators allowed in the arithmetic expression are `+`, `-`, `*`, `/`, and `^`, which represent addition, subtraction, multiplication, division, and exponentiation, respectively. However, **the order of operations has been changed!** Addition and subtraction now have precedence over multiplication and division, which now have precedence over exponentiation. Addition and subtraction still have equal precedence. Similarly, multiplication and division still have equal precedence. Operators of equal precedence should be evaluated from left to right.

Write a function with the following header:

```
function [result] = my_calculator_inverse_precedence(expression)
```

where:

- `expression` is a row vector of class `char` that represents an arithmetic expression as described above.
- `result` is a scalar of class `double` that represents the value of the arithmetic expression described by `expression`.

You can assume that:

- `expression` is not empty.
- `expression` is a valid arithmetic expression.
- `expression` contains only characters among: `0123456789.+-*/^`. In particular, `expression` does not contain spaces nor parentheses.
- The first character in `expression` is one of the 10 digits.
- There are no two operators in a row (*e.g.,* `+-`) in `expression`.

You may **not** use Matlab's built-in functions `split`, `strsplit`, `find`, `strfind`, `eval`, and `feval` in this question.

Hints:

- Consider using recursion for this question.
- You can use the function `str2num` to convert a number given as a character string into the

corresponding Matlab object of class `double`. For example:

```
>>  x = str2num('3.14')
x =
    3.1400

>> class(x)
    ans =
double
```

- Do not use the function `double` to convert a number given as a character string into the corresponding Matlab object of class `double`. For example:

```
>>  x = double('3.14')
x =
    51    46    49    52
```

Note that this (perhaps surprising) behavior will be explained later in the semester, when discussing the binary representation of data.

Test cases:

```
>> result = my_calculator_inverse_precedence('4-3.14')
result =
    0.8600

>> result = my_calculator_inverse_precedence('4-2-2')
result =
     0

>> result = my_calculator_inverse_precedence('3+5*2')
result =
    16

>> result = my_calculator_inverse_precedence('2^3/3')
result =
     2

>> result = my_calculator_inverse_precedence('8-2-2*4^2')
result =
   256
```

3.5. Calculator with undo

In this question, you will also write a function that calculates the result of an arithmetic expression between scalars, where the arithmetic expression is given as a character string (*e.g.*, `'2*30+5^4'`). The operators allowed in the arithmetic expression are `+`, `-`, `*`, `/`, and `^`, which represent addition, subtraction, multiplication, division, and exponentiation, respectively. Additionally, we introduce the undo operator `!`. The operator `!` signifies "cancel the previous operation". Two successive undo operators (`!!`) signify "cancel the previous two operations", and so on. The operation that the undo operator removes consists of the combination of the number and the operator that precede the undo operator. For exam-

ple, in the expression `'2*30!+5^4'`, the undo operator cancels the multiplication by `30`, and the arithmetic expression reduces to `'2+5^4'`. As another example, in the expression `'1+2*30!!+5^4'`, the undo operators cancel the multiplication by `30` and the addition of `2`, and the arithmetic expression reduces to `'1+5^4'`. The arithmetic expression should be evaluated from left to right, **assuming that all operators have equal precedence**.

Write a function with the following header:

```
function [result] = my_calculator_with_undo(expression)
```

where:

- `expression` is a row vector of class `char` that represents an arithmetic expression as described above.
- `result` is a scalar of class `double` that represents the value of the arithmetic expression described by `expression`.

You can assume that:

- `expression` is not empty.
- `expression` is a valid arithmetic expression.
- There are enough operations before undo operators such that there remains at least one scalar to the left of where the undo operators were, after the corresponding operations have been undone.
- `expression` contains only characters among: `0123456789.+-*/^!`. In particular, `expression` does not contain spaces nor parentheses.
- The first character in `expression` is one of the 10 digits.
- There are no two operators (among the list: `+`, `-`, `*`, `/`, and `^`) in a row (*e.g.,* `+-`) in `expression`. There can be, however, one or more undo operators directly before (but not after) any other operator, and/or at the end of `expression`.

You may **not** use Matlab's built-in functions `split`, `strsplit`, `find`, `strfind`, `eval`, and `feval` in this question.

Note: there are many ways to solve this question, including using a stack to keep track of each operation and to facilitate the undoing of operations.

Test cases:

```
>> result = my_calculator_with_undo('2.5-2')
result =
    0.5000

>> result = my_calculator_with_undo('2.5-2!')
result =
    2.5000
```

```
>> result = my_calculator_with_undo('2*3+6!^2')
result =
    36

>> result = my_calculator_with_undo('2*3+6!^2!')
result =
     6

>> result = my_calculator_with_undo('2*3+6!!^2')
result =
     4
```

4. General hints

The functions that you write for this lab assignment can call each other.