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
– Solutions –
Version: release

Most of the time, if not always, there are more than one correct way to implement the functions for E7 lab assignments. This set of solutions proposes only one, or a few, of the possible implementations for each function.

For this assignment, the required submissions were:

- `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 (20 points)

1.1. Approximation with a fixed number of terms (10 points)

We propose three different implementations as solutions for this question. The first proposed implementation uses a `for` loop:

my_sin_approx_fixed.m

```
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
```

The second implementation that we propose as a solution for this question uses recursion:

my_sin_approx_fixed_with_recursion.m

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

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

if n == 0
    approx = x;
else
    approx = my_sin_approx_fixed_with_recursion(x, n-1) + ...
        (-1)^n * x^(2*n+1) / factorial(2*n+1);
end

end
```

The third implementation that we propose as a solution for this question uses neither iteration nor recursion, but uses array operations instead:

my_sin_approx_fixed_arrays.m

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

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

ns = 0:n;
approx = sum((-1).^ns .* x.^(2*ns+1) ./ factorial(2*ns+1));

end
```

The published test cases were:

```
>> % Published test case number 1 (1 point)
>> approx = my_sin_approx_fixed(0, 0)

approx =
     0

>> % Published test case number 2 (1 point)
>> approx = my_sin_approx_fixed(2*pi/3, 2)
```

```
approx =
    0.8990

>> % Published test case number 3 (1 point)
>> approx = my_sin_approx_fixed(2*pi/3, 4)

approx =
    0.8661

>> % Published test case number 4 (1 point)
>> approx = my_sin_approx_fixed(5*pi/2, 3)

approx =
 -189.6141

>> % Published test case number 5 (1 point)
>> approx = my_sin_approx_fixed(5*pi/2, 10)

approx =
    1.0135
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> approx = my_sin_approx_fixed(-1.2, 0)

approx =
   -1.2000

>> % Hidden test case number 2 (1 point)
>> approx = my_sin_approx_fixed(-1.2, 5)

approx =
   -0.9320

>> % Hidden test case number 3 (1 point)
>> approx = my_sin_approx_fixed(10, 10)

approx =
    2.7611

>> % Hidden test case number 4 (1 point)
>> approx = my_sin_approx_fixed(-10, 15)

approx =
    0.5441

>> % Hidden test case number 5 (1 point)
>> approx = my_sin_approx_fixed(pi/5, 7)

```

```
approx =
    0.5878
```

1.2. Approximation with a specified tolerance (10 points)

The proposed solution for this question is:

my_sin_approx_tolerance.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 1.2 of Lab 04.
%
% Version: release.

n = 0;
approx = x;
exact = sin(x);

while abs(approx-exact) > tolerance
    n = n + 1;
    approx = approx + (-1)^n * x^(2*n+1) / factorial(2*n+1);
end

end
```

The published test cases were:

```
>> % Published test case number 1 (1 point)
>> [approx, n] = my_sin_approx_tolerance(pi/3, 1e-2)

approx =
    0.8663
n =
     2

>> % Published test case number 2 (1 point)
>> [approx, n] = my_sin_approx_tolerance(4*pi/5, 1e-3)

approx =
    0.5884
n =
     4

>> % Published test case number 3 (1 point)
>> [approx, n] = my_sin_approx_tolerance(pi/3, 1e-10)

approx =
    0.8660
n =
```

```
     6

>> % Published test case number 4 (1 point)
>> [approx, n] = my_sin_approx_tolerance(11*pi/2, 1e-5)

approx =
   -1.0000
n =
    26
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> [approx, n] = my_sin_approx_tolerance(pi/5, 1e-4)

approx =
    0.5878
n =
     2

>> % Hidden test case number 2 (1 point)
>> [approx, n] = my_sin_approx_tolerance(-pi/5, 1e-4)

approx =
   -0.5878
n =
     2

>> % Hidden test case number 3 (1 point)
>> [approx, n] = my_sin_approx_tolerance(-1.2, 1e-8)

approx =
   -0.9320
n =
     5

>> % Hidden test case number 4 (1 point)
>> [approx, n] = my_sin_approx_tolerance(-1.2, 0.01)

approx =
   -0.9327
n =
     2

>> % Hidden test case number 5 (1 point)
>> [approx, n] = my_sin_approx_tolerance(3*pi+0.5, 1)

approx =
   -0.9465
n =
    11
```

```
>> % Hidden test case number 6 (1 point)
>> [approx, n] = my_sin_approx_tolerance(3*pi+0.5, 1e-6)

approx =
   -0.4794
n =
    17
```

2. Sorting (25 points)

2.1. Minimum of a vector and its index (10 points)

The proposed solution for this question is:

my_minimum_index.m

```
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
```

The published test cases were:

```
>> % Published test case number 1 (1 point)
>> [minimum, index] = my_minimum_index(5)

minimum =
     5
index =
     1

>> % Published test case number 2 (1 point)
>> [minimum, index] = my_minimum_index([5, 6, 1])

minimum =
     1
```

```
index =
     3

>> % Published test case number 3 (1 point)
>> [minimum, index] = my_minimum_index([5, 6, 1, -1, 1])

minimum =
    -1
index =
     4

>> % Published test case number 4 (1 point)
>> [minimum, index] = my_minimum_index([4, 10, NaN, 2, 5, 50])

minimum =
     2
index =
     4
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> [minimum, index] = my_minimum_index([-10])

minimum =
   -10
index =
     1

>> % Hidden test case number 2 (1 point)
>> [minimum, index] = my_minimum_index([-10, 7, 0, -20.3, 5, 2, 0, -20])

minimum =
  -20.3000
index =
     4

>> % Hidden test case number 3 (1 point)
>> [minimum, index] = my_minimum_index([-5*pi, -2*pi, 0, 1, 2, pi, 5*pi])

minimum =
  -15.7080
index =
     1

>> % Hidden test case number 4 (1 point)
>> [minimum, index] = my_minimum_index([NaN, 3, 6, 0])

minimum =
     0
index =
```

```
     4

>> % Hidden test case number 5 (1 point)
>> [minimum, index] = my_minimum_index([NaN, NaN, NaN, NaN])

minimum =
   NaN
index =
     1

>> % Hidden test case number 6 (1 point)
>> [minimum, index] = my_minimum_index([-10, 7, NaN, 0, -20.3, 5, NaN, 0, -20])

minimum =
  -20.3000
index =
     5
```

2.2. Sorting a row vector (15 points)

We propose two different implementations as solutions for this question. Neither of these implementations is particularly efficient (*i.e.* they sort vectors rather slowly) when compared to other, more advanced, algorithms. The two proposed implementations are, however, significantly less complex to implement than other, more advanced, algorithms. We will talk about more complex but more efficient sorting algorithms toward the end of the semester.

The first proposed solution for this question uses the function `my_minimum_index` that we wrote for the previous question:

my_sort.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 2.2 of Lab 04.
%
% Version: release.

sorted = zeros(size(vector));
for i = 1:numel(vector)
    [minimum, index] = my_minimum_index(vector);
    sorted(i) = minimum;
    vector(index) = [];
end

end
```

The second implementation proposed as a solution for this question uses a sorting algorithm known as "bubble sort". This algorithm consists of traversing the vector and swapping any two consecutive elements that are not in order. This process is repeated until the entire vector is traversed without any swapping taking place (which indicates that the vector is fully sorted). Note that this implementation does not use the function `my_minimum_index` that we wrote for the previous question. The proposed implementation of the bubble sort algorithm is:

my_sort_bubble.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 2.2 of Lab 04.
%
% This implementation uses the bubble sort algorithm.
%
% Version: release.

n = numel(vector);
keep_sorting = true;
while keep_sorting
    % We start by assuming that we will not need to keep sorting after this
    % iteration of the while loop. We will set keep_sorting to true later
    % on if necessary
    keep_sorting = false;
    for i = 1:n-1
        % Swap consecutive elements if they are not in order
        if vector(i) > vector(i+1) | isnan(vector(i)) & ~isnan(vector(i+1))
            vector(i:i+1) = vector(i+1:-1:i);
            % We keep sorting if at least one swap took place while
            % traversing the vector
            keep_sorting = true;
        end
    end
end

sorted = vector;

end
```

The published test cases were:

```
>> % Published test case number 1 (1.5 point)
>> [sorted] = my_sort([0])

sorted =
     0

>> % Published test case number 2 (1.5 point)
>> [sorted] = my_sort([2, 1])

sorted =
     1     2

```

```
>> % Published test case number 3 (1.5 point)
>> [sorted] = my_sort([2, 1, 9, -10, pi])

sorted =
  -10.0000    1.0000    2.0000    3.1416    9.0000

>> % Published test case number 4 (1.5 point)
>> [sorted] = my_sort([6, 3, 7, 1, 0, 1, 7])

sorted =
     0     1     1     3     6     7     7

>> % Published test case number 5 (1.5 point)
>> [sorted] = my_sort([6, 3, 7, 1, NaN, 0, 1, 7])

sorted =
     0     1     1     3     6     7     7   NaN
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1.5 point)
>> [sorted] = my_sort(1:0.5:5)

sorted =
  Columns 1 through 7
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000
  Columns 8 through 9
    4.5000    5.0000

>> % Hidden test case number 2 (1.5 point)
>> [sorted] = my_sort([5:-1:1, 5:-1:1])

sorted =
     1     1     2     2     3     3     4     4     5     5

>> % Hidden test case number 3 (1.5 point)
>> [sorted] = my_sort([14, -100, -16, -68, -81, 40, 52, 57, 44, -69, 12, 21])

sorted =
  -100   -81   -69   -68   -16    12    14    21    40    44    52    57

>> % Hidden test case number 4 (1.5 point)
>> [sorted] = my_sort([7, -12.5, 71, NaN, 30, -40, 42, 49, -83, 71.25])

sorted =
  Columns 1 through 7
  -83.0000  -40.0000  -12.5000    7.0000   30.0000   42.0000   49.0000
  Columns 8 through 10
   71.0000   71.2500       NaN

>> % Hidden test case number 5 (1.5 point)
```

```
>> [sorted] = my_sort([-22, 84, 5, -77, -82, 75, 17, -48, -26, -80, 15, NaN])

sorted =
   -82   -80   -77   -48   -26   -22     5    15    17    75    84   NaN
```

3. Character string manipulation (55 points)

3.1. Reversing a vector without using recursion (10 points)

We propose two implementations as solutions for this question. The first proposed implementation uses a `for` loop:

my_reverse_without_recursion.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.1 of Lab 04.
%
% Version: release.

reversed = vector;
for i = 1:numel(vector)
    reversed(i) = vector(end-i+1);
end

end
```

The second implementation that we propose as a solution for this question does not use iteration. In the following implementation, the first clause of the `if` statement is used to ensure that if the size of `vector` is 0×0, then the size of `reversed` is also 0×0:

my_reverse_without_recursion_without_loop.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.1 of Lab 04.
%
% Version: release.

if numel(vector) == 0
    reversed = vector;
else
    reversed = vector(end:-1:1);
end

end
```

The published test cases were:

```
>> % Published test case number 1 (2 points)
>> reversed_char = my_reverse_without_recursion('Hello E7!')

reversed_char =
!7E olleH

>> % Published test case number 2 (2 points)
>> reversed_logical = my_reverse_without_recursion([true; true; false; true])

reversed_logical =
  4x1 logical array
   1
   0
   1
   1

>> % Published test case number 3 (2 points)
>> reversed_double = my_reverse_without_recursion(0:2:10)

reversed_double =
    10     8     6     4     2     0
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> reversed = my_reverse_without_recursion('')

reversed =
  0x0 empty char array

>> % Hidden test case number 2 (1 point)
>> reversed = my_reverse_without_recursion('thgiR ot tfeL')

reversed =
Left to Right

>> % Hidden test case number 3 (1 point)
>> reversed = my_reverse_without_recursion([-2, 40, 31, 100, -92] > -10)

reversed =
  1x5 logical array
   0   1   1   1   1

>> % Hidden test case number 4 (1 point)
>> reversed = my_reverse_without_recursion([-2; 40; 31; 100; -92])

reversed =
   -92
   100
    31
    40
```

```
    -2
```

3.2. Reversing a vector using recursion (10 points)

The proposed solution for this question is:

my_reverse_with_recursion.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.2 of Lab 04.
%
% Version: release.

n = numel(vector);
s = size(vector);

if n < 2
    reversed = vector;
elseif s(1) == 1
    reversed = [vector(end), my_reverse_with_recursion(vector(1:end-1))];
else
    reversed = [vector(end); my_reverse_with_recursion(vector(1:end-1))];
end

end
```

The published test cases were:

```
>> % Published test case number 1 (2 points)
>> reversed_char = my_reverse_with_recursion('Hello E7!')

reversed_char =
!7E olleH

>> % Published test case number 2 (2 points)
>> reversed_logical = my_reverse_with_recursion([true; true; false; true])

reversed_logical =
  4x1 logical array
   1
   0
   1
   1

>> % Published test case number 3 (2 points)
>> reversed_double = my_reverse_with_recursion(0:2:10)

reversed_double =
    10     8     6     4     2     0
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> reversed = my_reverse_with_recursion('')

reversed =
  0x0 empty char array

>> % Hidden test case number 2 (1 point)
>> reversed = my_reverse_with_recursion('thgiR ot tfeL')

reversed =
Left to Right

>> % Hidden test case number 3 (1 point)
>> reversed = my_reverse_with_recursion([-2, 40, 31, 100, -92] > -10)

reversed =
  1x5 logical array
   0   1   1   1   1

>> % Hidden test case number 4 (1 point)
>> reversed = my_reverse_with_recursion([-2; 40; 31; 100; -92])

reversed =
   -92
   100
    31
    40
    -2
```

3.3. Character string parsing (15 points)

The proposed solution for this question is:

my_parser.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.3 of Lab 04.
%
% Version: release.

% We look at each character of the string until we find one of the
% delimiters
n = numel(string);
i = 1;
while i <= n & ~any(delimiters==string(i))
    i = i + 1;
end

```

```
% If i is equal to n+1, it means that we did not find any delimiter in the
% string
if i == n + 1
    delimiter = '';
    left = string;
    right = '';
else
    delimiter = string(i);
    left = string(1:i-1);
    right = string(i+1:end);
end

% Adjust the size of the empty character strings, if any
if numel(left) == 0
    left = '';
end
if numel(right) == 0
    right = '';
end

end
```

The published test cases were:

```
>> % Published test case number 1 (2.5 points)
>> [delimiter, left, right] = my_parser('Hello+World', '+')

delimiter =
+
left =
Hello
right =
World

>> % Published test case number 2 (2.5 points)
>> [delimiter, left, right] = my_parser('Another-test', ',-')

delimiter =
-
left =
Another
right =
test

>> % Published test case number 3 (2.5 points)
>> [delimiter, left, right] = my_parser('NO DELIMITER??', '!()')

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

```
right =
  0x0 empty char array
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1.5 point)
>> [delimiter, left, right] = my_parser('2*3!+6^2', '!')

delimiter =
!
left =
2*3
right =
+6^2

>> % Hidden test case number 2 (1.5 point)
>> [delimiter, left, right] = my_parser('No seven here.', '7')

delimiter =
  0x0 empty char array
left =
No seven here.
right =
  0x0 empty char array

>> % Hidden test case number 3 (1.5 point)
>> [delimiter, left, right] = my_parser('2.5*3.14+6^2/10', '+-*/^')

delimiter =
*
left =
2.5
right =
3.14+6^2/10

>> % Hidden test case number 4 (1.5 point)
>> [delimiter, left, right] = my_parser('First delimiter', 'Ff')

delimiter =
F
left =
  0x0 empty char array
right =
irst delimiter

>> % Hidden test case number 5 (1.5 point)
>> [delimiter, left, right] = my_parser('Last delimiter', 'Rr')

delimiter =
r
left =
```

```
Last delimite
right =
  0x0 empty char array
```

3.4. The order of operations has been changed! (10 points)

For this question, it is convenient to use a function that is similar to the function `my_parser` that we wrote for the previous question, but that parses from the right instead of from the left (*i.e.* it parses the character string around the last delimiter that appears in the character string). Such a function is implemented as a sub-function of the proposed solution for this question:

my_calculator_inverse_precedence.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.4 of Lab 04.
%
% Version: release.

% This question will be solved using recursion. The stopping criterion for
% the recursion is that there is no operator in the expression
[delimiter, left, right] = my_parser(expression, '+-*/^');
if numel(delimiter) == 0
    result = str2num(expression);
    return
end

% We are going to calculate recursively expressions of the form:
%
% left (some operation) right
%
% For example:
%
% left * right
% left ^ right
%
% The values of "left" and "right" are calculated using recursion
%
% To respect the order of operations using this approach, we have to find
% the operators with the least precedence first, meaning starting with the
% right-most ^

% The operator with lowest precedence is exponentiation
[delimiter, left, right] = my_parser_from_right(expression, '^');
if strcmp(delimiter, '^')
    left = my_calculator_inverse_precedence(left);
    right = my_calculator_inverse_precedence(right);
    result = left ^ right;
    return
end
```

```

% Then come multiplication and division, which have equal precedence
[delimiter, left, right] = my_parser_from_right(expression, '*/');
if numel(delimiter) > 0
    left = my_calculator_inverse_precedence(left);
    right = my_calculator_inverse_precedence(right);
    if strcmp(delimiter, '*')
        result = left * right;
    elseif strcmp(delimiter, '/')
        result = left / right;
    end
    return
end

% Finally come addition and subtraction, which have equal precedence. If we
% reach this point of the code, we are sure that there is at least one
% addition or one subtraction
[delimiter, left, right] = my_parser_from_right(expression, '+-');
left = my_calculator_inverse_precedence(left);
right = my_calculator_inverse_precedence(right);
if strcmp(delimiter, '+')
    result = left + right;
elseif strcmp(delimiter, '-')
    result = left - right;
end

end

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

% This function is similar to the function my_parser, except that it parses
% from the right of "string" instead of from the left. This implementation
% is not the most efficient implementation, but is possibly one of the
% least complex ones, since we already wrote my_parse and my_reverse (with
% or without recursion).

[delimiter, l, r] = my_parser(my_reverse_without_recursion(string), delimiters);
left = my_reverse_without_recursion(r);
right = my_reverse_without_recursion(l);

end
```

The published test cases were:

```
>> % Published test case number 1 (1 point)
>> result = my_calculator_inverse_precedence('4-3.14')

result =
    0.8600

>> % Published test case number 2 (1 point)
```

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

result =
     0

>> % Published test case number 3 (1 point)
>> result = my_calculator_inverse_precedence('3+5*2')

result =
    16

>> % Published test case number 4 (1 point)
>> result = my_calculator_inverse_precedence('2^3/3')

result =
     2

>> % Published test case number 5 (1 point)
>> result = my_calculator_inverse_precedence('8-2-2*4^2')

result =
   256
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> result = my_calculator_inverse_precedence('4')

result =
     4

>> % Hidden test case number 2 (1 point)
>> result = my_calculator_inverse_precedence('3.1416')

result =
    3.1416

>> % Hidden test case number 3 (1 point)
>> result = my_calculator_inverse_precedence('4+2+2/10')

result =
    0.8000

>> % Hidden test case number 4 (1 point)
>> result = my_calculator_inverse_precedence('4+2/3/20.5')

result =
    0.0976

>> % Hidden test case number 5 (1 point)
>> result = my_calculator_inverse_precedence('42+2/10.5-1*0.5^2+1.1')
```

```
result =
   13.5073
```

3.5. Calculator with undo (10 points)

We propose two implementations as solutions for this question. The first proposed implementation uses the concept of stacks. This implementation parses the expression from left to right around operators. When an operator that is not an undo is found, the number to its left is added to a stack of numbers and the operator itself is added to a stack of operators. When an undo operator is found, the last number on the stack of numbers is removed and the last operator on the stack of operators is removed. Once the parsing is completed, we have a stack of number and a stack of operators where none of the operators are undo operators. Additionally, these stacks do not contain the operations that were undone by undo operators. The result of the expression is calculated from these two stacks using an iterative process without recursion. This proposed implementation is:

my_calculator_with_undo.m

```matlab
function [result] = my_calculator_with_undo_other(expression)

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.5 of Lab 04.
%
% Version: release.

% Create empty stacks for numbers and for operators
numbers = [];
operators = '';

% We parse the expression from the left, adding the corresponding numbers
% and operators to their respective stacks
[delimiter, left, right] = my_parser(expression, '+-*/^!');
while numel(delimiter) > 0

    % The character string "left" might be empty (if, for example, there
    % were two undo operators in a row). If it is not empty, we need to add
    % the corresponding number to the stack of numbers
    if numel(left) > 0
        numbers(end+1) = str2num(left);
    end

    % If we encounter an undo operator, we remove the last number and
    % operator from their respective stacks. Otherwise, we add the operator
    % to the stack
    if strcmp(delimiter, '!')
        operators(end) = [];
        numbers(end) = [];
    else
        operators(end+1) = delimiter;
    end
```

```

    % Keep parsing what remains of the expression (i.e. the right-hand side)
    [delimiter, left, right] = my_parser(right, '+-*/^!');

end

% If the last operation was not an undo, we need to add the last number to
% the corresponding stack ("left" will be an empty character string if and
% only if the last operator was an undo)
if numel(left) > 0
    numbers(end+1) = str2num(left);
end

% At this point of the code, we have stacks of numbers and of operators
% without any undos. We calculate the result using iteration
result = numbers(1);
for i = 1:numel(operators)
    operator = operators(i);
    if strcmp(operator, '+')
        result = result + numbers(i+1);
    elseif strcmp(operator, '-')
        result = result - numbers(i+1);
    elseif strcmp(operator, '*')
        result = result * numbers(i+1);
    elseif strcmp(operator, '/')
        result = result / numbers(i+1);
    elseif strcmp(operator, '^')
        result = result ^ numbers(i+1);
    end
end

end
```

The second implementation proposed as solution for this question does not rely on the concept of stacks. Rather, it first removes all the operations that need to be undone from the expression, and then uses recursion to evaluate the remaining expression (which does not contain any undo operator):

my_calculator_with_undo_alternative.m

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

% E7 Spring 2017, University of California at Berkeley.
% Solution function for question 3.5 of Lab 04.
%
% Version: release.

% First, we remove all the undos (and the corresponding operations that are
% undone) from the expression
[delimiter, left, right] = my_parser(expression, '!');
while strcmp(delimiter, '!')

```

```
    % Remove the last operation before the undo operator (i.e. the last
    % operation in "left")
    [d, left, r] = my_parser_from_right(left, '+-*/^');

    % Attach the updated left-hand side and the right-hand side of the
    % expression together
    expression = [left, right];

    % Look for another undo operator
    [delimiter, left, right] = my_parser(expression, '!');

end

% At this point of the code, there is no more undo operators in the
% expression. We calculate the expression from left to right using
% recursion. To respect the order of operations using this approach, we
% have to find the operators with the least precedence first, meaning
% starting with the right-most operator
[delimiter, left, right] = my_parser_from_right(expression, '+-*/^');
if strcmp(delimiter, '+')
    result = my_calculator_with_undo_alternative(left) + str2num(right);
elseif strcmp(delimiter, '-')
    result = my_calculator_with_undo_alternative(left) - str2num(right);
elseif strcmp(delimiter, '*')
    result = my_calculator_with_undo_alternative(left) * str2num(right);
elseif strcmp(delimiter, '/')
    result = my_calculator_with_undo_alternative(left) / str2num(right);
elseif strcmp(delimiter, '^')
    result = my_calculator_with_undo_alternative(left) ^ str2num(right);
else
    % The base case for the recursion is that there is no operator in the
    % expression
    result = str2num(right);
end

end

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

% This function is similar to the function my_parser, except that it parses
% from the right of "string" instead of from the left. This implementation
% is not the most efficient implementation, but is possibly one of the
% least complex ones, since we already wrote my_parse and my_reverse (with
% or without recursion).

[delimiter, l, r] = my_parser(my_reverse_without_recursion(string), delimiters);
left = my_reverse_without_recursion(r);
right = my_reverse_without_recursion(l);

end
```

The published test cases were:

```
>> % Published test case number 1 (1 point)
>> result = my_calculator_with_undo('2.5-2')

result =
    0.5000

>> % Published test case number 2 (1 point)
>> result = my_calculator_with_undo('2.5-2!')

result =
    2.5000

>> % Published test case number 3 (1 point)
>> result = my_calculator_with_undo('2*3+6!^2')

result =
    36

>> % Published test case number 4 (1 point)
>> result = my_calculator_with_undo('2*3+6!^2!')

result =
     6

>> % Published test case number 5 (1 point)
>> result = my_calculator_with_undo('2*3+6!!^2')

result =
     4
```

The hidden test cases are:

```
>> % Hidden test case number 1 (1 point)
>> result = my_calculator_with_undo('0')

result =
     0

>> % Hidden test case number 2 (1 point)
>> result = my_calculator_with_undo('3.1416+3.1416!')

result =
    3.1416

>> % Hidden test case number 3 (1 point)
>> result = my_calculator_with_undo('1+2-3!*4*5^6!')

result =
    60

>> % Hidden test case number 4 (1 point)
```

```
>> result = my_calculator_with_undo('4+2/3!!/20.5')

result =
    0.1951

>> % Hidden test case number 5 (1 point)
>> result = my_calculator_with_undo('42+2/10.5-1*0.5^2+1.1!!!^2-4')

result =
    6.1791
```