This programming assignment may seem really long. However, we are giving a very detailed explanation to help you become familiar with MATLAB®, M-script (the scripting language used by MATLAB®), and the way we will be building a small library of functions.

This particular exercise does everything for you if you follow along with the video and type in the code that is discussed there. In future exercises you will do more of the programming yourself.
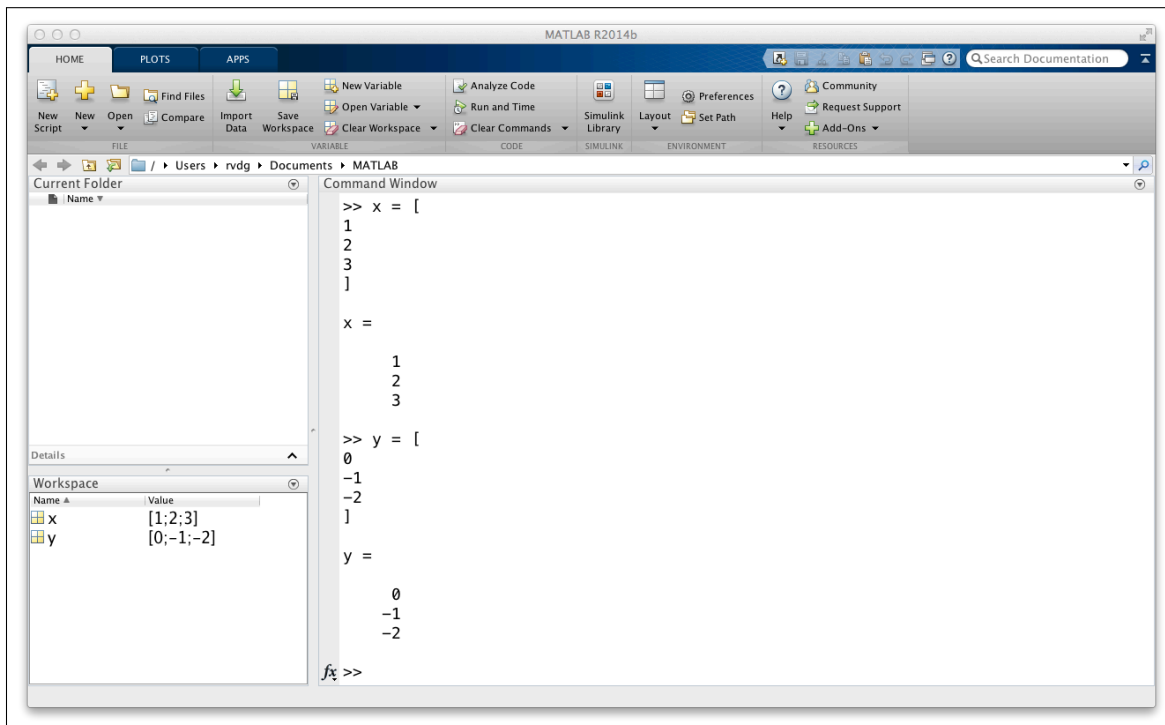
Preliminaries

Start MATLAB® and in the `Command Window` start by creating vectors

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} 0 \\ -1 \\ -2 \end{pmatrix}.$$

Type

```
1  >> x = [
2  1
3  2
4  3
5  ];
6  >> y = [
7  0
8  -1
9  -2
10 ];
```

into the `Command Window`:

```
x = [1;2;3];
y = [0;-1;-2];
```

This still creates column vectors. The ";"s separate rows in x and y) and the ";" at the end of the line suppresses the printing of the result. However, we usually like to emphasize that *x* and *y* are column vectors by entering them as columns.

At this point, you may want to have a look at what is in variables x and y:

```
>> x

x =

        1
        2
        3

>> y

y =

        0
       -1
       -2
```

Now, copying one vector into another is easy with M-script:

```
>> x_old = x

x_old =

        1
        2
        3

>> y_old = y;
>>
```

Notice how if one adds ";" after a command, the result of that command is *not* printed.

We have created new variables x_old and y_old because this will make it convenient later to restore the contents of x and y.

Indexing into a vector

In our course we index starting at 0:

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \chi_2 \end{pmatrix}$$

so that in this example $\chi_0 = 1$, $\chi_1 = 2$, and $\chi_2 = 3$. M-script instead starts indexing at 1:

```
>> x( 1 )

ans =

    1
```

The point of most of our exercises that use M-script is to link coding algorithms to the abstractions that we use. An abstraction on the math side is the notion of the vector, x, as an ordered list of numbers that we call elements and that we index starting at 0: $\chi_0, \ldots, \chi_{n-1}$. The corresponding abstraction in M-script is the array x that we index into starting at 1: $x(1), x(2), x(3)$.

On the math side, we can assign the elements of a vector x to another vector y. To do so, the sizes of vectors x and y have to be equal and then assigning $y := x$ (y becomes x) means that

$$
\begin{aligned}
\psi_0 &:= \chi_0 \\
\psi_0 &:= \chi_0 \\
&\vdots \qquad \vdots \\
\psi_{n-1} &:= \chi_{n-1}
\end{aligned}
$$

After the assignment $y := x$, the equalities

$$
\begin{aligned}
\psi_0 &= \chi_0 \\
\psi_0 &= \chi_0 \\
&\vdots \qquad \vdots \\
\psi_{n-1} &= \chi_{n-1}
\end{aligned}
$$

hold. Another way of saying this uses the *"for all" quantifier* (universal quantifier):

$$
\forall_{i=0}^{n-1} (\psi_i = \chi_i),
$$

which one should read as "ψ_i equals χ_i for all indexes i from 0 to $n-1$." This is shorthand for

$$
\psi_0 = \chi_0 \quad and \quad \psi_1 = \chi_1 \quad and \quad \cdots \quad and \quad \psi_{n-1} = \chi_{n-1}.
$$

We can code the assignment of one vector to another as

```
>> y = x;
```

In this case, the output array y is automatically made of the same size as x. If we want to more explicitly assign individual elements, we could instead execute

```
>> y( 1 ) = x( 1 );
>> y( 2 ) = x( 2 );
>> y( 3 ) = x( 3 );
```

Using a `for` loop

Simple assignment works fine if *x* and *y* have only a few elements. But if they are long and/or you don't know how many elements they have, you may want to write a *loop*:

```
>> for  i =1:3
y( i ) = x( i )
end

y =

      1
     −1
     −2


y =

      1
      2
     −2


y =

      1
      2
      3
```

The "for all" abstraction in the math translates to a `for` loop in M-script. The array `y` is printed every time. This can be avoided by executing

```
>> for  i =1:3
y( i ) = x( i );
end
>> y

y =

      1
      2
      3
```

instead (notice the ";"s).

Notice how the "for all" abstraction $\forall_{i=0}^{n-1}$ translates into the M-script loop

```
for  i =1:n
    ...
end
```

Functions

Let us pretend that M-script does not have the built-in (instrinsic) ability to copy one array to another by executing $y = x$. It sure would be inconvenient to always write a loop every time we wanted to copy one array to another. For this reason, programming languages typically include the ability to create functions. To do so, create a new function in MATLAB®:
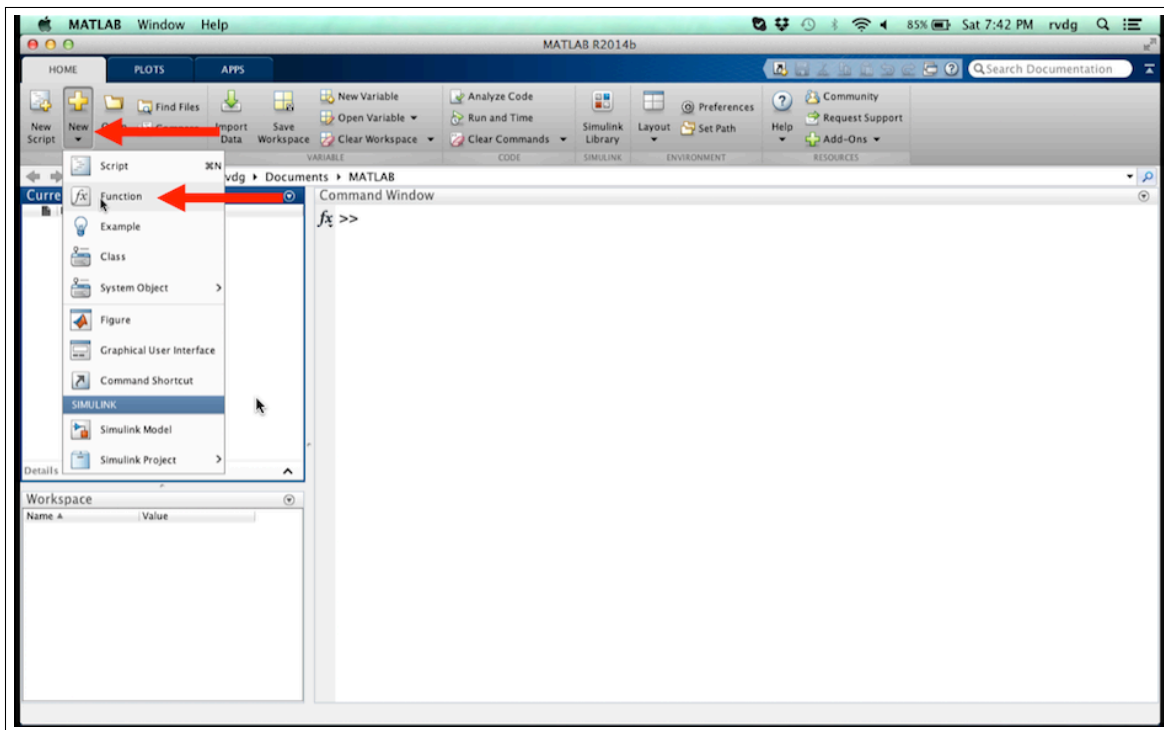
yielding
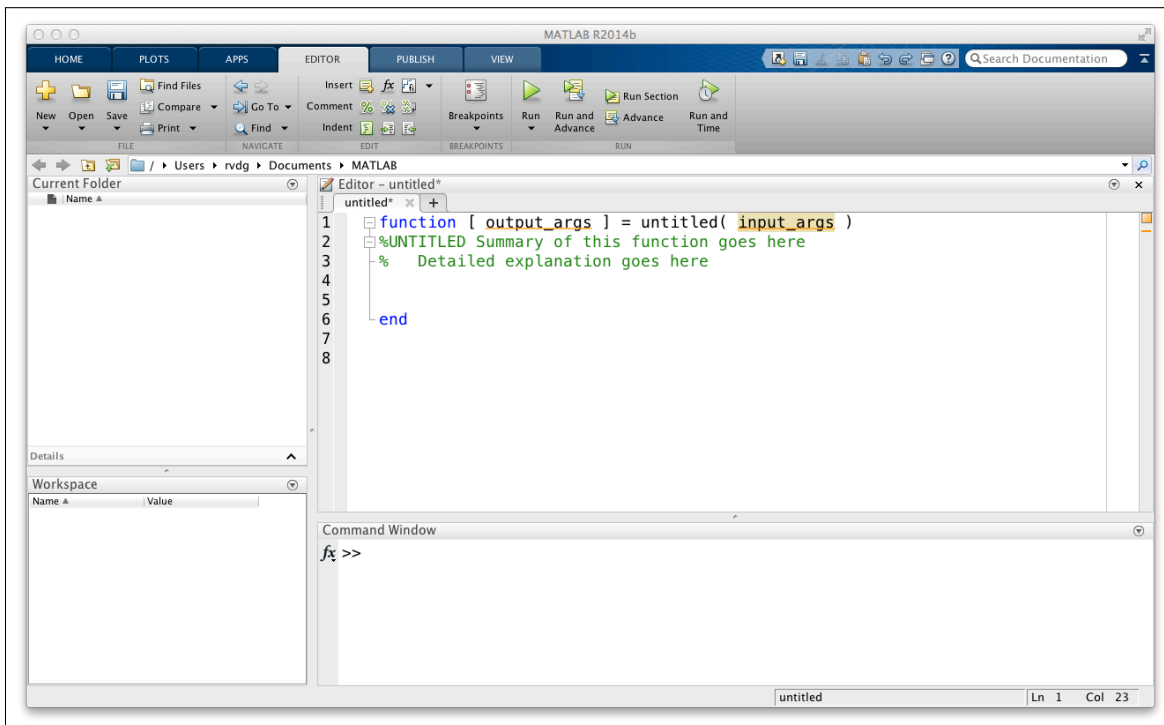
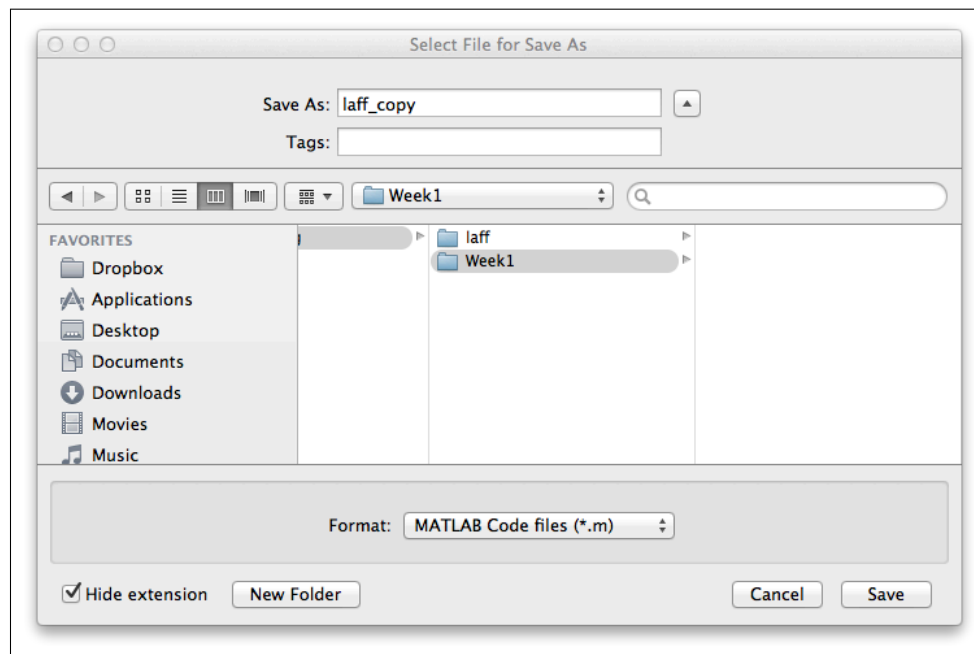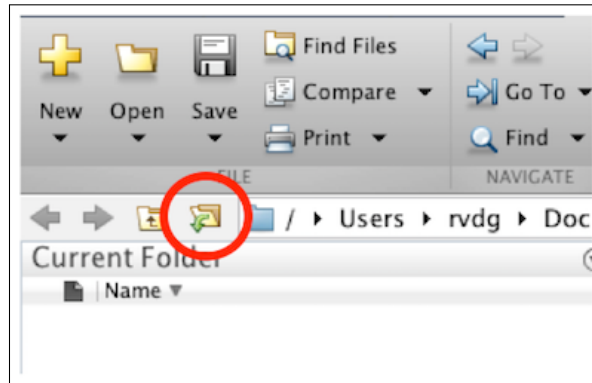We now create a function that copies one vector to another:
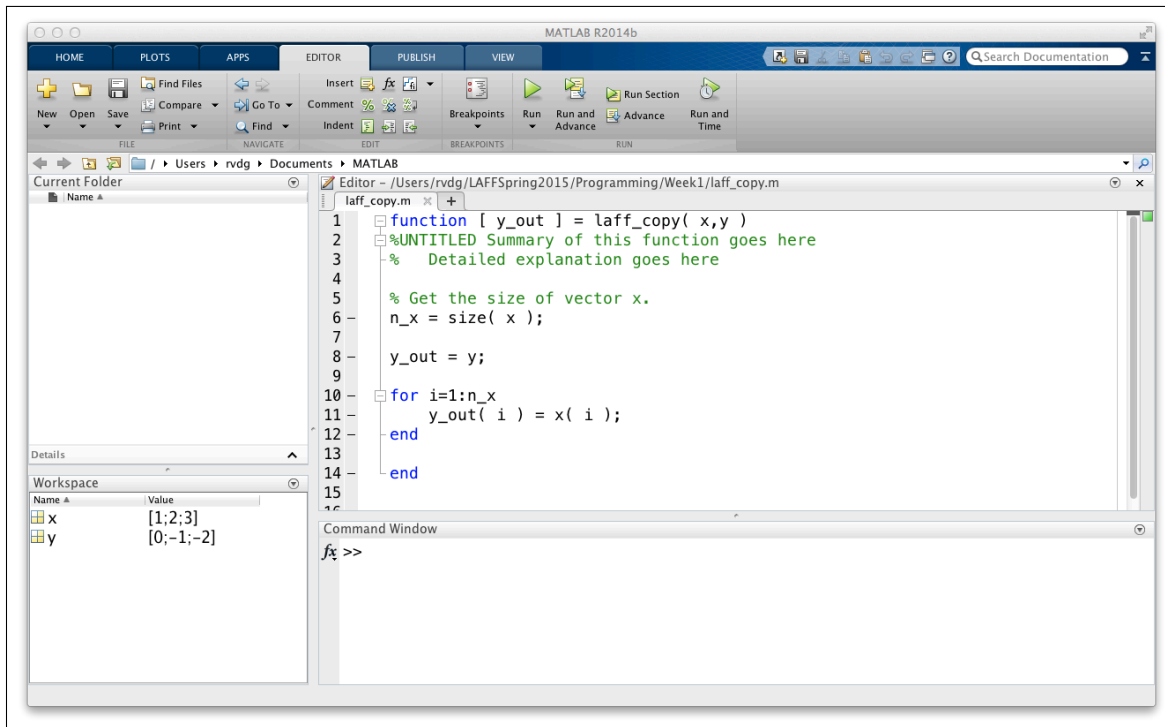
which you save somewhere convenient, for example in

$$LAFFSpring2015 \to Code \to \to \text{laff\_copy}.$$

Next, click on "browse for folder" (the picture of the folder with the green arrow):

and choose the folder in which you saved `laff_copy.m`. By picking this folder, when one uses the function in the `Command Window`, it is found in the file in which it was stored (which should always have the same name as the name of the function, with ".m" added: `laff_copy` should be stored in file `laff_copy.m`). Notice how `laff_copy.m` shows in `Current Folder`:
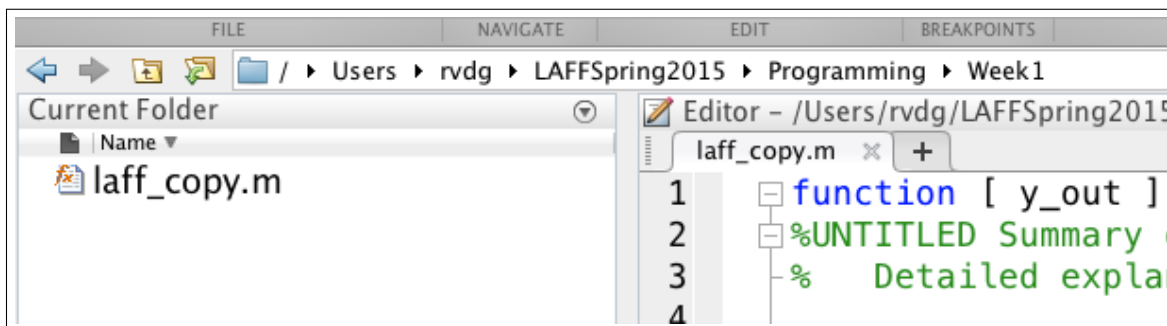
We are ready to use the function to copy the vector. Execute `y = laff_copy( x, y )` in the `Command Window`:
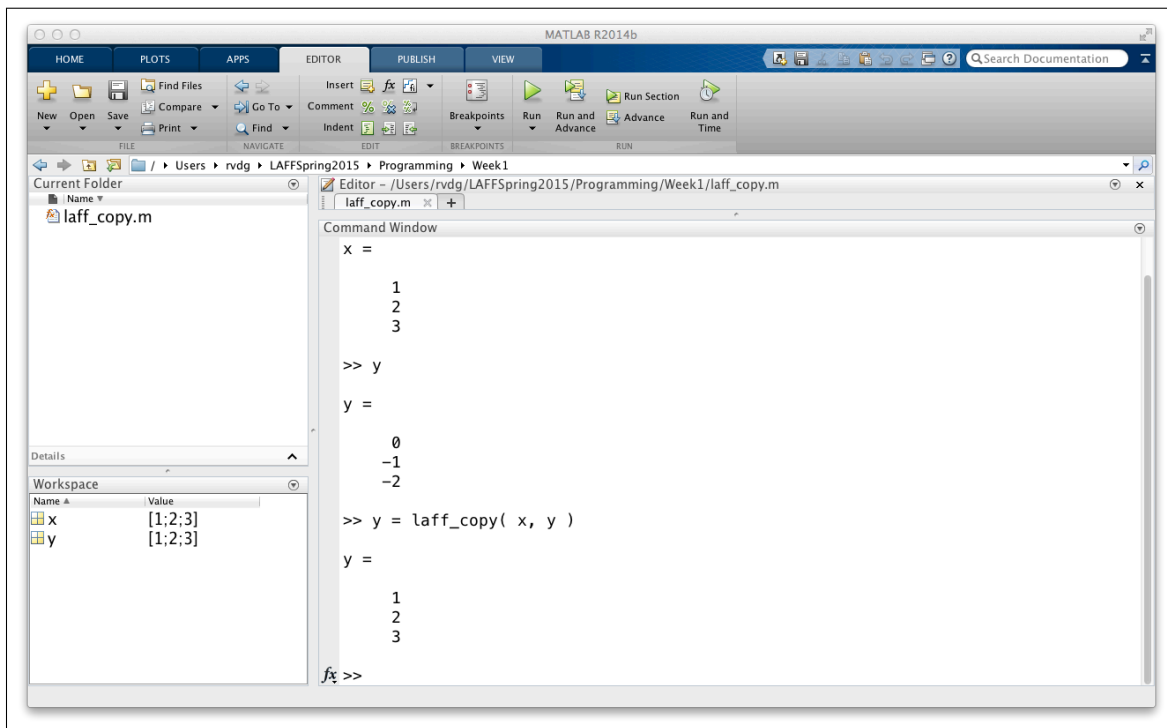
We are going to use the `laff_copy` routine to copy not just column vectors to column vectors, but also row vectors to column vectors, row vectors to row vectors, and column vector to row vectors. Adding this functionality

```matlab
function [ y_out ] = laff_copy( x, y )

% y = copy( x, y ) copies vector x into vector y
%    Vectors x and y can be a mixture of column and/or row vector.  In other
%    words, x and y can be n x 1 or 1 x n arrays.  However, one size must
%    equal 1 and the other size equal n.

% Extract the row and column sizes of x and y
[ m_x, n_x ] = size( x );
[ m_y, n_y ] = size( y );

% Make sure x and y are (row or column) vectors of equal length
if ( m_x ~= 1 & n_x ~= 1 ) | ( m_y ~= 1 & n_y ~= 1 )
    y_out = 'FAILED';
    return
end
if ( m_x * n_x ~= m_y * n_y )
    y_out = 'FAILED';
    return
end

if ( n_x == 1 )      % x is a column vector
    if ( n_y == 1 )      % y is a column vector
        % Copy the elements of x into the elements of y
        for i=1:m_x
            y( i,1 ) = x( i,1 );
        end
    else      % y is a row vector
        % Copy the elements of x into the elements of y
        for i=1:m_x
            y( 1,i ) = x( i,1 );
        end
    end
else      % x is a row vector
    if ( n_y == 1 )      % y is a column vector
        % Copy the elements of x into the elements of y
        for i=1:n_x
            y( i,1 ) = x( 1,i );
        end
    else      % y is a row vector
        % Copy the elements of x into the elements of y
        for i=1:n_x
            y( 1,i ) = x( 1,i );
        end
    end
end

% Return the updated y in y_out
y_out = y;

return
end
```

Figure 1: Implementation of `laff_copy` that can copy a row or column vectors *x* to row or column vectors *y*. For the complete routine, see `LAFFSpring2015 -> Code -> laff -> vecvec -> laff_copy.m`.

to the function we just wrote makes the function a bit more complex, as illustrated in Figure **??**. Here we notice that in M-script vectors are special cases of matrices, which have row and column sizes. Thus, treating x and y as arrays that store matrices (that happen to be column or row vectors), we can extract the row and column size (e.g., m_x and n_x for x) with the M-script function `size`. This can then be used to determine whether x and/or y are row and/or column vectors.

Testing

We now suggest that you copy the file `laff_copy.m` from `LAFFSpring2015 -> Code -> laff -> vecvec` to `LAFFSpring2015 -> Code ->` so that we all have the same implementation of `laff_copy`. We now test this routine. In the Command Window type

```
>> x = [
1
2
3
];

>> y_old = [
0
-1
-2
];

>> z_old = [
4
3
2
1
]
```

This gives us a couple of vector with which to check the behavior of our function. Next, try the following:

```
>> y = y_old;
>> y = laff_copy( x, y )
```

Does it give the expected output? Go on and also try

```
>> y = y_old';
>> y = laff_copy( x, y )
```

Here y_old' (notice the ') creates a row vector from the column vector y_old. This is known as *transposing* the (column) vector, which yields a row vector. Did you get the expected output? Finally, try

```
>> x_row = x';
>> y = y_old;
>> y = laff_copy( x_row, y )
>> y = y_old';
>> y = laff_copy( x_row, y )
>> z = z_old;
>> z = laff_copy( x, z )
```

With this last command, you should getting a warning about the sizes of the vectors not matching by z equaling 'FAILED'.

Scripts

What if you had made a mistake in the implementation of laff_copy? You would have had to test an updated version all over again, executing all the commands to do so. Wouldn't it be much more convenient to have a file of commands that you can execute? This is called a script. To create one, you either use your favorite editor, or you can click on New Script. You may want to enter the commands that we give in Figure **??**. Alternatively, copy the file ⋯ -> laff -> vecvec -> test_copy.m. Save this script in test_copy.m (don't forget the .m) in directory ⋯ -> and then in the Command Window execute

```
>> test_copy
```

While we use isequal to check whether two vector are equal in test_copy, one must be careful using this function: later in the course we will work with vectors that are real valued. In a computer, these are stored as floating point numbers. This means that is often the case that two vectors would have been equal in exact arithmetic, but due to round off error that is incurred when computing with floating point numbers, they many not be exactly equal.

Summary

We have now created a routine laff_copy that copies one vector to another as well as a script to test it. In the process, you have learned a few things about MATLAB® and M-script programming.

```matlab
% Create some vectors
x = [ 1; 2; 3; ]
y = [ 0; -1; -2 ]
z = [ 4; 3; 2; 1 ]

% test column -> column copy
disp( 'column -> column copy' )
if ( isequal( laff_copy( x, y ), x ) )
    disp( 'PASSED' )
else
    disp( 'FAILED' )
end

% test column -> row copy
disp( 'column -> row copy' )
if ( isequal( laff_copy( x, y' ), x' ) )
    disp( 'PASSED' )
else
    disp( 'FAILED' )
end

% test row -> column copy
disp( 'row -> column copy' )
if ( isequal( laff_copy( x', y ), x ) )
    disp( 'PASSED' )
else
    disp( 'FAILED' )
end

% test row -> row copy
disp( 'row -> row copy' )
if ( isequal( laff_copy( x', y' ), x' ) )
    disp( 'PASSED' )
else
    disp( 'FAILED' )
end

% test column -> column copy (wrong size)
disp( 'column -> column copy (wrong size)' )
if ( isequal( laff_copy( x, z ), 'FAILED' ) )
    disp( 'PASSED' )
else
    disp( 'FAILED' )
end

% test column -> row copy (wrong size)
disp( 'column -> row copy (wrong size)' )
if ( isequal( laff_copy( x, z' ), 'FAILED' ) )
    disp( 'PASSED' )
else
    disp( 'FAILED' )
end
%%%% MORE TESTS HERE %%%%
```

Figure 2: Test script for laff_copy. (The complete script can be found in LAFFSpring2015 -> Code -> laff -> vecvec -> test_copy.m.)

If anything in MATLAB® that we did puzzles you, try using `help` in the MATLAB® `Command Window`:

```
>> help for
>> help if
>> help ==
>> help isequal
```

function is

$$y\_out = laff\_copy(x, y).$$

The reasons are

- The input parameter y indicates whether the output variable y_out is a row or a column vector.

- Almost always, we will use this routine to overwrite an existing (row or column) vector with the (row or column) vector x. Thus, the typical use is

$$y = laff\_copy(x, y).$$

- If we were to use a language like the C programming language, as would be common practice in scientific computing, then the routine would have been called as

$$laff\_copy(x, y)$$

where the vector y would be overwritten with vector x.