Matlab Basics

1 The Matlab Integrated Development Environment

When Matlab is first opened you are presented with an integrated development environment (IDE), which is depicted in Figure 1 below.

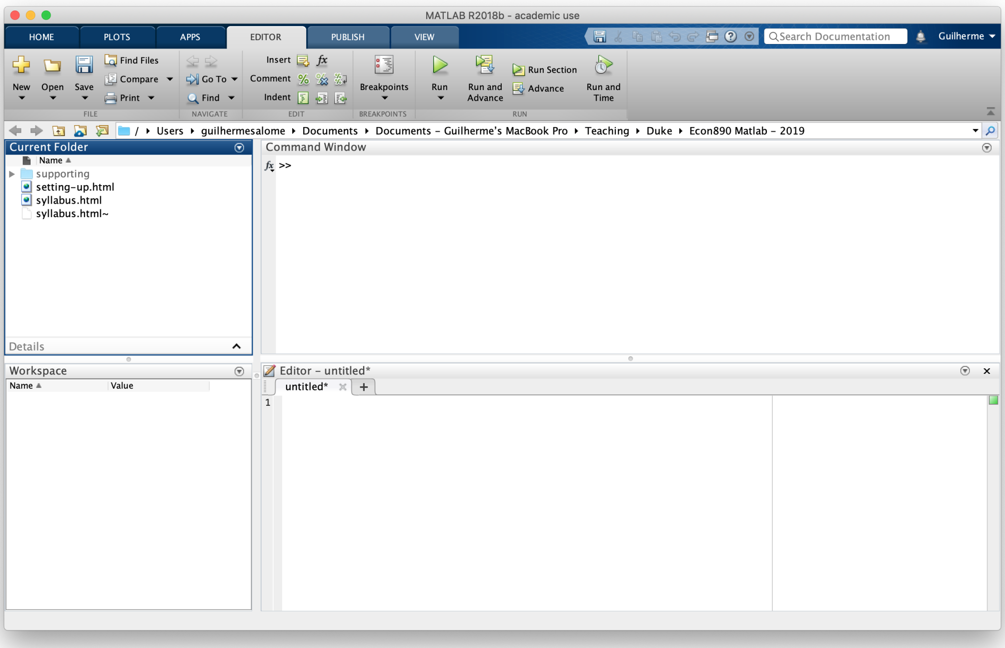


Figure 1: The Matlab IDE.

The figure shows four windows from left-to-right: Current Folder, Command Window, Workspace, Editor. The `Current Folder` window displays all files in the working directory Matlab is currently at. This working directory is where Matlab will search for user-defined functions and other files (more on this later). You can change the working directory by using the navigation bar above the window.

The `Command Window` is known as a repeat-eval-print-loop (REPL), and is where we can interactively type and execute Matlab commands. As you type commands in, Matlab will execute them and then wait for more. For example, type the following in the `Command Window` and hit enter to execute:

```
cd ~/
```

Matlab will change the working directory to the home folder of your computer. Here, we evaluated a special function `cd` with the argument `~/`.

The `Workspace` window shows a list of all variables created so far. The list is empty because we have not created any variable yet. Create a new variable by typing:

```
x = 1
```

Now the `Workspace` window should show the name and value of the variable.

Last, the `Editor` window is where we edit `.m` files. These files are used to create Matlab scripts and to define functions. Notice that the `Editor` window can open multiple files by using separate tabs.

All four windows can be rearranged and even removed from view. Feel free to customize the IDE to your needs. I often remove the `Workspace` window, and rearrange the `Editor` window for it be on top, leaving a smaller space for the `Command Window` on the bottom of the IDE. Figure 2 shows the IDE with the windows rearranged.

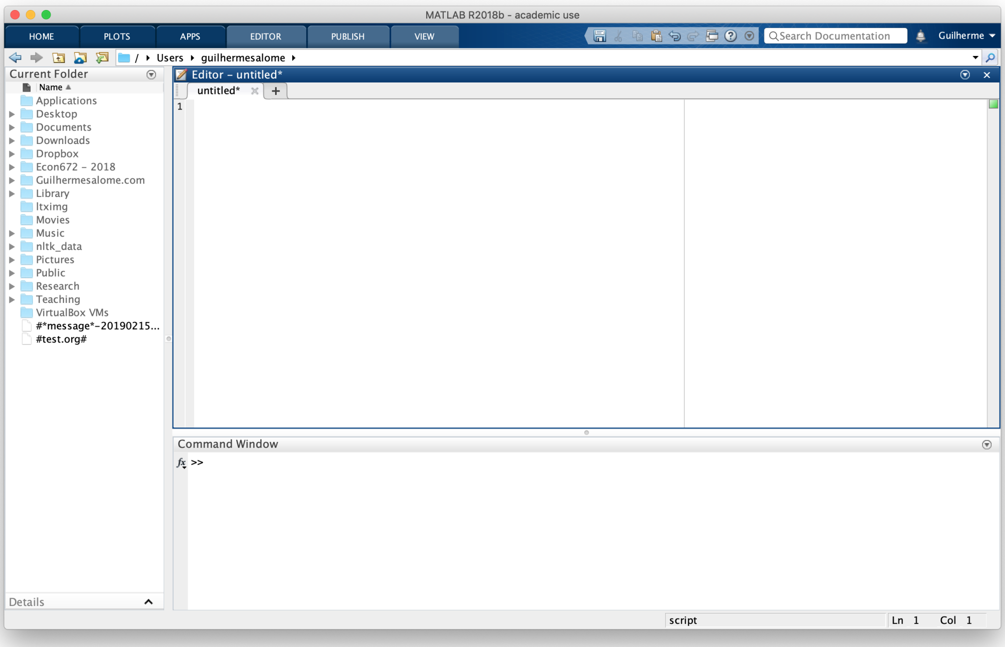


Figure 2: A possible way to organize the windows in the IDE.

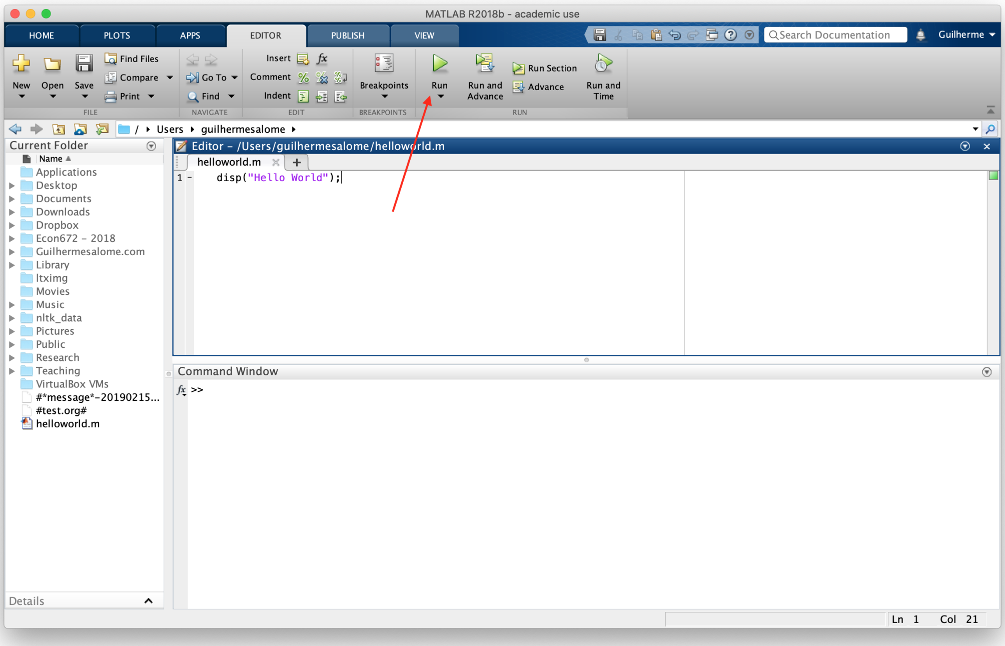
2 Hello World!

We will now create a "hello world" program in Matlab. Go on the `Editor` window and type the following:

```
% helloworld.m
disp("Hello World!");
```

Save the file with a descriptive name, like `helloworld.m`. Now, we would like to run our `helloworld.m` program. There are two ways of doing so:

- Click on `Editor` and then on the green button `Run` (see Figure 3);
- Or, on the `Command Window` type the name of the file without the `.m` extension and hit enter.



```
disp("Hello World");
```

Figure 3: Running a Matlab script.

When we run the file, Matlab displays the result in the `Command Window`. The first line of the file starts with a percentage symbol, which indicates a comment line. In the second line, the function `disp` is called. This function takes a string as an argument and displays it on the screen.

3 Linear Regression with Ordinary Least Squares

To learn the fundamentals of the Matlab programming language we will implement the ordinary least squares (OLS) estimator for the parameters of a linear regression under the classical assumptions.

Let's assume that the economic model of interest is given by:

$$\underbrace{y}_{n\times 1} = \underbrace{X}_{n\times K}\underbrace{\beta}_{K\times 1} + \underbrace{\varepsilon}_{n\times 1}$$

where y is a $n \times 1$ vector of observed dependent variables, X is an $n \times K$ matrix of observed explanatory variables, β is a $K \times 1$ vector of unknown parameters and ε is a $n \times 1$ vector of unobserved explanatory variables.

We know that the OLS estimator of β (see Equation 1.2.5 in Hayashi (2000)) is given by:

$$\hat{\beta} = (X'X)^{-1}X'y \tag{1}$$

Our first objective is to implement this estimator.

Using a bottom-up approach, we need to understand:

1. The data types for numbers in Matlab;

2. How to create vectors and matrices;
3. How to do matrix transposition, inversion and multiplication;
4. How to create a function that given X and y produces $\hat{\beta}$;
5. How to generate data to test out our code.

3.1 Numeric Data Types for Numbers

Numeric values are stored in the double-precision floating point format, by default. The class name of this type of numeric value is `double`. We can create doubles by simply assigning them to variables:

```
x = 10;
y = 5
```

Whenever you see code in a box as above, you should run it yourself by typing the commands line by line, in the `Command Window`.

Notice that when the semicolon is omitted, Matlab displays the result of the assignment on the screen. If you want to suppress this display you need the semicolon. The use of semicolon is important when constructing scripts, so that the `Command Window` is not filled with information we do not need to see. However, omitting the semicolon when typing in the `Command Window` is useful to see the result of operations.

```
% The two lines below
y = 5;
disp(y);
% are equivalent to:
y = 5
```

The usual mathematical operations can be performed with numbers:

```
x + y
x - y
% multiplication
x*y
% floating point division
y/x
x/y
% exponentiation
x^y
% remainder of integer division
mod(13, 5)
% integer division
floor(13/5)
% absolute value
abs(-1)
% maximum
max(x, y)
% minimum
min(x, y)
```

```
% exponential
exp(4)
% narutal logarithm
log(exp(1))
% square root
sqrt(4)
4^0.5
% factorial
factorial(x)
% pi
pi
sqrt(pi)
% trigonometry
sin(0)
cos(0)
tan(0)
```

The result of the operations above are still `double`.

You can verify the type and size in memory of a variable with the special[1] function `whos`:

```
whos x;
whos x y;
```

It is important to notice that the number displayed in the `Command Window` is a formatted version of the actual number stored in a variable. For example, π is displayed as 3.1416. By default, Matlab displays numbers using at most 5 digits, and it may use scientific notation for numbers that are too big or too small.

```
pi
0.0000009809809
213124412123
```

We can change the display format for numeric values between 5 digits and 15 digits using the special command `format`:

```
pi
% Use 15-digits to display numbers
format long;
pi
% Use 5-digits to display numbers
format short;
pi
% Force scientific notation with 5-digits
format short e;
pi
```

[1]The function `whos` is being called with a special syntax called the `command syntax`. Usually, in a function call the parameters are passed inside parentheses, as in `mod(x, y)`. However, the `command syntax` allows us to pass parameters without using parentheses, as in `whos x y`. It is not often used, except with a few Matlab commands. More details about this `command syntax` are available at this page.

```
% Force scientific notation with 15-digits
format long e;
pi
```

Observe that, as with other implementations of floating-point arithmetic (as in C or FORTRAN), Matlab is also subject to the usual rounding errors.

```
3*(4/3 - 1) - 1
```

However, this is not a bug, but an issue resulting from not being possible to write $\frac{4}{3}$ as a binary number using a finite number of bits.

There are other numeric types available, such as integers, complex numbers, infinity, not a number, among others. It is worth mentioning that infinity and not a number can result from some arithmetic operations:

```
% Finding infinity in Matlab
1/0
-1/0
% Not a number
1/0 - 1/0
```

Using `double` is sufficient for most applications, and for this reason we will skip the other numeric types. However, applications that are memory bound or that require a lot of optimization may benefit from the other numeric types. The official documentation on all numeric types is available on this page.

3.2 Vectors and Matrices

Vectors and matrices are structured collections of numbers of type `double` (by default). We can create vectors and matrices by:

```
% A 5x1 vector
y = [10; -2; 3.4; exp(3); log(4.5)]
% A 5x3 matrix
X = [1, 2, 3;
     1,-3, 5;
     1, 4, 0;
     1, 0,-1;
     1, 4.5, 3.3]
```

Notice that commas separate columns and semicolons separate rows. The use of commas to separate columns is optional.

We can very obtain the dimension of a vector with the function `length`, and the dimensions of a matrix with the function `size`:

```
length(y)
size(X)
```

What would be the result of calling `length` on a matrix? We can figure that out by reading the `help` documentation on the `length` function.

```
help length
```

In the case of a matrix, it will return the value of the highest dimension.

Notice that the function `size` applied on a matrix returns two numbers. We can access these two numbers separately by: saving them on two separate variables, or asking the function `size` to return only one of the numbers.

```matlab
A = ones(2, 3)
help size
% save the dimensions to two variables
[d1, d2] = size(A)
% ask size to return only the 1st dimension
d1 = size(A, 1)
% ask size to return only the 2nd dimension
d2 = size(A, 2)
```

To access the elements of vectors and matrices we use the following syntax:

```matlab
% The first element of a vector has the index 1
y(1)
% The second element of a vector has the index 2
y(2)
% And so on until the last element
y(5)
% For a matrix we need to define both dimensions
% The very first value at the first row and first column
X(1, 1)
% The value on the first row and second column
X(1, 2)
% The value on the last row and the last column
X(5, 3)
% There is a special syntax to get all rows of a column
X(:, 2) % all rows of 2nd column
X(:, 1) % all rows of 1st column
% The same notation applies to get all columns of a row
X(4, :) % all columns of the 4th row
X(1, :) % all columns of the 1st row
% We can use a similar notation to access slices of the
   matrix
X(2:4, 4) % rows 2 to 4 of the 4th column
X(3, 1:2) % columns 1 to 2 of the 3rd row
X(2:4, 1:2) % columns 1 to 2 of rows 2 to 4
```

The values stored in arrays (vectors, matrices and higher-dimensional matrices) can be modified in place:

```matlab
a = [0 1; 2 3]
a(2, 1) = 500
% multiple values can be updated at once
a(:, 1) = [30; 30] % modifying an entire column
a(2, :) = [20 20] % modifying an entire row
```

There are a few functions that assist in the creation of matrices:

```
% Matrix full of zeros
help zeros
zeros(2)
zeros(4, 1)
% Matrix full of ones
ones(3)
ones(2, 3)
% Identity matrix
eye(5)
eye(2, 3)
% Sparse Identity Matrix
A = eye(1000)
B = speye(1000) % optimized
whos A B
% Diagonal matrix
diag([1 2 3 4 5])
```

We can also generate a vector by using a range of numbers:

```
% start:stop
A = 1:4
% the values are incremented by 1 until the value is greater
    than
% the stop, which is 4
B = -5: 3.5
% we can change the increment using the syntax start:
    increment:stop
C = -5:0.5:3.5
```

We can combine other matrices to create new ones:

```
A = [X(:, 1), X(:, 3)]
```

3.3 Matrix Operations

We can now create matrices and modify them. Now we will cover the basic matrix operations. Let's start with matrix and scalar operations:

```
X = [1 2 3; 4 5 6]
% operations with a scalar are elementwise
X + 3
X - 2
% special notation for elementwise multiplication, division
    and exponentiation
X.*3
X./3
X.^3
% log, exp and sqrt apply elementwise
log(X)
```

```
exp(X)
sqrt(X)
```

Matrix and matrix operations:

```
A = [10 20 30; 0 -5 -10];
B = [2 2 2; 5 10 20];
% elementwise operations (matrices have the same dimensions)
A + B
A - B
A./B
A.*B
% concatenation
% concatenate rows (vertical concatenation)
C = vertcat(A, B)
C = [A; B]
% concatenate columns (horizontal concatenation)
D = horzcat(A, B)
D = [A B]
% repetition
repmat(A, 2, 3) % repeats matrix A: 2 times in row, 3 times
    in column
```

Now, for linear algebra:

```
A = [10 2 30;
     0 -5 -10]
B = ones(size(A, 2), 1)
% Matrix multiplication
C = A*B
% Matrix tranposition
transpose(A)
A'
% Matrix inversion
C = [A; B']
C^(-1)
inv(C)
% Solving linear system Cx=B
solution = C\B
% Eigenvalues and eigenvectors
eigenvalues = eig(C)
[eigenvectors, eigenvalues] = eig(C)
% Singular value decomposition
[U, S, V] = svd(C)
% Kronecker Product
kron(A, B)
```

There are many other useful linear algebra functions. The full documentation is available on the linear algebra reference page.

3.4 Functions (and Conditionals)

We now have the main ingredients to implement the OLS estimator from Equation 1:

```matlab
y = [10; -2; 3.4; exp(3); log(4.5)]
X = [1, 2, 3;
     1,-3, 5;
     1, 4, 0;
     1, 0,-1;
     1, 4.5, 3.3]
% The OLS estimator for the regression y = X*beta + epsilon
beta_hat = inv(X'*X)*X'*y
```

Now that we have the estimator, we want to be able to use it for whatever matrices X and y we have. To do that, we create a function.

Functions in Matlab are created in separate files ending in `.m`. Each function has its own `.m` file[2]. The syntax for a function is `function [output1, ..., outputN] = foo(input1, ..., inputM)`, then the actual function code, and last we finish with the keyword `end`. Let's create a simple function for the OLS estimator:

```matlab
% linreg_ols.m
function [beta] = linreg_ols(y, X)
beta = inv(X'*X)*X'*y;
end
```

Whenever you see a box as above where the first line starts with a comment of the type `% somename.m`, it means that you should type the code in a new file and save it with the name given in the comment. Save the file on the current folder Matlab is using.

Notice that the name of the function must be the same name of the file. After saving the file, we can use the function by just calling its name:

```matlab
beta = linreg_ols(y, X)
```

Matlab can access all `.m` files in the current folder and on sub-folders within it.[3]

There are a couple of modifications we need to do to the `linreg_ols` function. First, we need to add documentation about it, so that our future selves know what it actually does. We do so by adding comments to the function right after the line where the function is declared. These comments should accomplish the following:

- A description of the function does in one line (or very few lines);
- A description of the function inputs;
- A description of the function outputs;
- Examples of usage.

[2]Strictly speaking there can be more than one function in a single `.m` file, but there must be one function with the same name as the file name, and the other functions in the file are known as local functions. These local functions are not available outside of the `.m` file, and their use is to break down the code in smaller pieces.

[3]It is possible to use functions and other files in folders outside the current folder. To do so, you need to add the path to that specific folder to the search path Matlab uses. This can be done with the function `addpath`. See the `addpath` reference for details.

After finishing core functions it is absolutely necessary to add documentation. If you stop using your code, even for a few days, it is very easy to forget what you did and what different functions do. Having the documentation is fundamental to quickly remember how to operate your code. We can add comments in Matlab with a percentage symbol. Let's add comments so that the usage of the function is clear:

```matlab
% linreg_ols.m
function [beta] = linreg_ols(y, X)
% LINREG_OLS estimates beta in the linear regression y = X*beta + epsilon
% via ordinary least squares
%
% Args:
%     y: A nx1 vector of dependent variables
%     X: A nxK matrix of independent variables
%
% Returns:
%     beta: A Kx1 vector of the estimated beta coefficients
%
% Examples:
%     y = [1;2;3];
%     X = rand(3);
%     beta = linreg_ols(y, X)
%
% Reference:
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn: 9780691010182.

beta = inv(X'*X)*X'*y;

end
```

You can see the documentation of the function with the special command `help`.

```matlab
help linreg_ols
```

Often we want to add an intercept to the linear regression. Let's extend our `linreg_ols` function to automatically add an intercept. To conserve space in the lecture notes, I will remove the documentation for our function, but you should not.

```matlab
% linreg_ols.m
function [beta] = linreg_ols(y, X)
% add intercept to X
X = horzcat(ones(size(X, 1), 1), X); % X is a local variable to this function
beta = inv(X'*X)*X'*y;
end
```

If we want the intercept to be optional, then we need to add a new input to the function that will represent whether or not we want the intercept to be added to X. We can do so by using the conditional statement `if`.

3.4.1 Conditionals

The syntax for conditional statements is:

```
if condition
    % code to run if condition is true
elseif another_condition
    % code to run if condition is false and
    % another_condition is true
else
    % code to run if condition and another_condition are
       false
end
```

In this case the `condition` is pretty simple, we can add an input with the name `add_intercept` and if its value is `true` we add the intercept, otherwise we do not. In Matlab, the possible logical (boolean) values are `true` and `false`, but the numbers `1` and `0` can also be used to the same extent. The basic relational operators are:

```
% equal to
true == 1
% not equal to
true ~= false
% greater than
3 > 2
% greater or equal than
2 >= 2
% less than
3 < 2
% less or equal than
-1 <= 0
```

It is also possible to combine different conditions with these logical operators:

```
% and
(1 > 0) && (2 > 1)
% or
(1 < 0) || (2 < 1)
% not
~(1 > 0)
```

3.4.2 Optional Intercept

Let's modify `linreg_ols` to make the intercept optional:

```
% linreg_ols.m
function [beta] = linreg_ols(y, X, add_intercept)
if add_intercept:
    X = horzcat(ones(size(X, 1), 1), X);
```

```
end
beta = inv(X'*X)*X'*y;
end
```

Finally, we update the documentation:

```
% linreg_ols.m
function [beta] = linreg_ols(y, X, add_intercept)
% LINREG_OLS estimates beta in the linear regression y = X*
   beta + epsilon
% via ordinary least squares
%
% Args:
%     y: A nx1 vector of dependent variables
%     X: A nxK matrix of independent variables
%     add_intercept: A boolean variable. If true, then a
   column of
%        ones is added to X, so that an intercept for the
   linear
%        regression is estimated.
%
% Returns:
%     beta: A Kx1 vector of the estimated beta coefficients
%
% Examples:
%     y = [1;2;3];
%     X = rand(3);
%     beta = linreg_ols(y, X)
%
% Reference:
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn:
   9780691010182.
if add_intercept:
    X = horzcat(ones(size(X, 1), 1), X);
end
beta = inv(X'*X)*X'*y;
end
```

3.5 Generating Data (and Loops and Text)

To test if our function is actually doing what we think it is doing, we need to test it. Testing is imperative when programming, since generating code that is 100\

In this case, we are going to simulate data for X, ε and y, while fixing true values for the parameters β. First, to simulate the data we will assume that In Matlab, the basic random number generation functions can generate values from the uniform and normal distributions. However, in the "Statistics and Machine Learning Toolbox" there are random number generators for many other distributions. Let's assume X comes from the uniform distribution on the interval $(0, 1)$, while ε comes from the standard normal distribution. We will generate these values in a loop.

3.5.1 Loops

In Matlab, we can create `for` loops and `while` loops. The syntax for `for` loops is:

```
% syntax for for loops
for variable = range
    % variable takes the value in the range
    % run some code
end
% example
for i = 1:10
    disp(i)
end
% example with nested loops
X = zeros(3)
for i = 1:size(X, 1)
    for j = 1:size(X, 2)
        X(i, j) = i + j;
    end
end
```

For loops will terminate when all values in the range are exhausted.

The syntax for `while` loops is similar:

```
% syntax for while loops
while condition
    % if condition is true
    % run some code
end
% the condition must be a boolean value

% example
i = 0
while i < 10
    disp(i)
    i = i + 1 % no i++ or += on Matlab
end
```

While loops will terminate when the condition is no longer true.

If you write an infinite loop, you can break out of it with `Ctrl + C`. In addition, if you need a loop to terminate earlier you can use the keyword `break`. For example:

```
numb_iter = 0
max_iter = 10
distance = 1000
while distance > 0.01
    distance = distance/2
    if numb_iter >= max_iter
```

```
        disp("I will break now")
        break
    else
        numb_iter = numb_iter + 1
    end
end
```

3.5.2 Generating Data in a Loop

Let's now create our test data. We will generate a matrix X with 100 rows and 5 columns, and ε will be a vector with 100 rows. We will create this data in a script, so that we can run it multiple times with different number of rows and columns.

```
% script: test_linreg_ols.m
% Generates data from the linear model y = X*beta + epsilon,
% given the true beta values. Estimates beta using the
   function
% linreg_ols and displays a table comparing true values to
% parameters.

% number of rows and columns
nrows = 100;
ncols = 5;
% true betas
beta = [-1; 2; 0.5; 3; 0.1];
% generate data for X, epsilon and y
X = zeros(nrows, ncols);
epsilon = zeros(nrows, 1);
y = zeros(nrows, 1);
for i = 1:nrows
    X(i, :) = rand(1, 5);
    epsilon(i) = randn();
    y(i) = X(i,:)*beta + epsilon(i);
end
% estimate the betas from X and y
beta_hat = linreg_ols(y, X, true);
```

Whenever we run the script above, we will generate data and estimate the values of β. Notice that we added an intercept in the estimation, even though the true model does not have an intercept. To make the comparison between true parameters values and their estimates, we would like to display the values side by side, with some text indicating what are the true parameters and what are the estimates.

3.5.3 Characters and Strings

In Matlab, there are two data types for text: `char` and `string`. A `char` holds a sequence of characters. It works like a vector of numbers, but holds characters instead. To create a `char` we use single quotes ':

```
name = 'Guilherme'
% name is an array of characters, the individual letters of
   my name
length(name)
name(1)
name(length(name))
% we can also concatenate character arrays
disp(['My name is ' name '!'])
```

Characters are meant to store short pieces of text, and are used to specify file names, labels and titles for plots, and inputs for functions.

A `string` array is meant to hold and manipulate larger pieces of text. Matlab provides various functions for manipulating text stored in `string` arrays. String arrays are created in a fashion similar to matrices, but with brackets and text in double quotes ":

```
% creating a simple string array
full_name = ["Guilherme", "Salome"];
% full_name is a string array
size(full_name)
full_name(1, 1)
full_name(1, 2)
% a string array with many rows and columns
students_names = ["First Name", "Last Name";
                  "A", "B";
                  "C", "D"];
size(students_names)
students_names(2, 2)
% a string array can also contain a single element
% in this case, we do not need the brackets
my_name = "Guilherme Salome"
size(my_name) % it is still an array
% we can find the length of a string with the strlength
   function
strlength(my_name)
% creating a string array that also has numbers will
% implicitely convert numbers to strings
data = ["Parameters", 1, 2, 3];
```

Let's explore some of the common operations we might perform with strings. A full list of functions that operate on strings are available on the reference page for characters and strings.

```
% we can append a string to string arrays
first_names = ["Mary", "John", "Paul"]
full_names = first_names + " Smith"
% split strings
split_sentence = split("This is a sentence about to be split
   ")
% split can be used to clean csv data
```

```
csv_data = "1.2, 3.2, -0.5, 5.3"
split_data = split(csv_data, ',')
% join strings
a_burguer = join(["Bread", "Tomato", "Lettuce", "Burguer", "
    Cheese", ...
                  "Bread"])
% if a command is too big for a single line, we can use
% the ... notation to continue it in the next line

% joining strings can be useful to store data in the csv file
data = [1.2; 3.2; -0.5; 5.3];
% the text manipulation functions apply to string arrays
% so we need to convert the numbers to a string array
data_string = string(data)
csv_data = join(data_string, ',')
% sorting strings
sort(full_names)
```

We can now display our parameter estimates using string arrays:

```
% compare the true betas to the estimates
disp(horzcat(["True Values"; 0; beta], ["Estimates"; beta_hat
    ]))
```

3.5.4 Table

We can create a better display by using a `table`. A `table` contains rectangular data that is column-oriented. We can create a table to display the true parameter values and their estimates. We create a table by passing it vectors with the values, each vector will be a column in the table. Additionally, we can pass a string array containing the column names.

```
% script: test_linreg_ols.m
% Generates data from the linear model y = X*beta + epsilon,
% given the true beta values. Estimates beta using the
    function
% linreg_ols and displays a table comparing true values to
% parameters.

% number of rows and columns
nrows = 100;
ncols = 5;
% true betas
beta = [-1; 2; 0.5; 3; 0.1];
% generate data for X, epsilon and y
X = zeros(nrows, ncols);
epsilon = zeros(nrows, 1);
y = zeros(nrows, 1);
for i = 1:nrows
```

```matlab
    X(i, :) = rand(1, 5);
    epsilon(i) = randn();
    y(i) = X(i,:)*beta + epsilon(i);
end
% estimate the betas from X and y
beta_hat = linreg_ols(y, X, true);
% display a table with the true parameters and estimates
estimates = table([0; beta], beta_hat, 'VariableNames', ["
   Parameters", ...
                        "Estimates"]);
disp(estimates)
```

We can now modify this script to see what happens to the estimates when the number of data points increases, or when the number of parameters increases and so on. We have covered most of the basics of Matlab by coding the `linreg_ols` function and the `test_linreg_ols.m` script.

4 Standard Errors for OLS Estimates

We will now extended the `linreg_ols` function to compute standard errors for the parameter estimates and the t-statistics.

Under the classical assumptions, the standard error of the OLS estimates (Equation 1.4.4 in Hayashi (2000)) is given by:

$$SE(\hat{\beta}_i) = \sqrt{s^2 \cdot [(X'X)^{-1}]_{[i,i]}}$$

where s^2 (Equation 1.2.13 in Hayashi (2000)) is computed from the residuals of the estimation:

$$e \equiv y - X\hat{\beta}$$
$$s^2 \equiv \frac{e'e}{n-K}$$

Let's implement this estimator in the `linreg_ols` function:

```matlab
% linreg_ols.m
function [beta, s2, stderr] = linreg_ols(y, X, add_intercept)
% Returns:
%     beta: A Kx1 vector of the estimated beta coefficients
%     s2: Estimate of the residual variance
%     stderr: A Kx1 vector of the estimators of the standard
   errors
%        of the beta estimates
if add_intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically
   stable
```

```
XXinverse = XX\eye(length(XX));
beta = XXinverse*X'*y;
% estimate Var(beta_hat|X)
e = y - X*beta;
s2 = (e'*e)/(size(X, 1) - size(X, 2));
stderr = diag((s2*XXinverse)^0.5);
end
```

Now, the t-statistic for the null-hypothesis of insignificant coefficients is:

$$t_i \equiv \frac{\hat{\beta}_i}{SE(\hat{\beta}_i)} \sim t_{n-K}$$

To compute the p-value for the test (see Pages 38 and 39 of Hayashi (2000)) we need the cumulative density function (cdf) of the student's T distribution, which is available in the function `tcdf`.

```
% linreg_ols.m
function [beta, s2, stderr, t_stat, p_value] = linreg_ols(y,
   X, add_intercept)
% LINREG_OLS estimates beta in the linear regression y = X*
   beta + epsilon
% via ordinary least squares
%
% Args:
%     y: A nx1 vector of dependent variables
%     X: A nxK matrix of independent variables
%     add_intercept: A boolean variable. If true, then a
   column of
%         ones is added to X, so that an intercept for the
   linear
%         regression is estimated.
%
% Returns:
%     beta: A Kx1 vector of the estimated beta coefficients
%     s2: Estimate of the residual variance
%     stderr: A Kx1 vector of the estimators of the standard
   errors
%         of the beta estimates
%     t_stat: A Kx1 vector with the t-statistics for the null
%         hypothesis that each beta is zero (separately)
%     p_value: A Kx1 vector with the p-values for the t-test
%
% Examples:
%     y = [1;2;3];
%     X = rand(3);
%     beta = linreg_ols(y, X, true)
%
% Reference:
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn:
    9780691010182.
if add_intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically
    stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
beta = XXinverse*X'*y;
% estimate Var(beta_hat|X)
e = y - X*beta;
s2 = (e'*e)/(n - K);
stderr = diag((s2*XXinverse)^0.5);
% t-statistics
t_stat = beta./stderr;
p_value = (1 - tcdf(abs(t_stat), n - K))*2;
end
```

5 Organizing the Regression Results in an Object

We could further extend the `linreg_ols` function to compute all the other statistics commonly found in other specialized statistics software. Every time we add a new computation to the function, we would have to extend the number of outputs. When there are too many outputs in a function, we need a better way to organize them. To organize multiple outputs of a function we can use a `struct`.

A `struct` (structure array) is a data type that groups other data by fields. Each field has a name and can contain any type of data. We access fields in a structure array with the `.` notation. Let's create a simple `struct`:

```
% data to store
beta_hat = [1;2;3];
stderr = [0.5, 0.3, 0.43];
% create a structure named results
results.b = beta_hat;
results.se = stderr;
% access the values in the struct
results.b
results.se
% creating a struct using the struct keyword
results = struct('b', beta_hat, 'se', stderr);
results.b
results.se
```

```
% creating a struct with empty fields (possibly to be filled
   later)
results = struct('b', [], 'se', []);
results.b
results.b = beta_hat;
results.b
```

We can use a `struct` to easily organize the output of `linreg_ols`:

```
% linreg_ols.m
function results = linreg_ols(y, X, add_intercept)
% LINREG_OLS estimates beta in the linear regression y = X*
   beta + epsilon
% via ordinary least squares
%
% Args:
%     y: A nx1 vector of dependent variables
%     X: A nxK matrix of independent variables
%     add_intercept: A boolean variable. If true, then a
   column of
%        ones is added to X, so that an intercept for the
   linear
%        regression is estimated.
%
% Returns:
%     results: A struct with the following fields:
%       beta: A Kx1 vector of the estimated beta coefficients
%       s2: Estimate of the residual variance
%       stderr: A Kx1 vector of the estimators of the
   standard
%       errors of the beta estimates
%       t_stat: A Kx1 vector with the t-statistics for the
   null
%          hypothesis that each beta is zero (separately)
%       p_value: A Kx1 vector with the p-values for the t-
   test
%
% Examples:
%     y = [1;2;3];
%     X = rand(3);
%     beta = linreg_ols(y, X, true)
%
% Reference:
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn:
   9780691010182.
if add_intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
```

```
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically
   stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
beta = XXinverse*X'*y;
% estimate Var(beta_hat|X)
e = y - X*beta;
s2 = (e'*e)/(n - K);
stderr = diag((s2*XXinverse)^0.5);
% t-statistics
t_stat = beta./stderr;
p_value = (1 - tcdf(abs(t_stat), n - K))*2;
% create results struct
results = struct('b', beta, 's2', s2, 'stderr', stderr, ...
                 't_stat', t_stat, 'p_value', p_value);
end
```

While a `struct` can be very useful to organize data, it is not the fastest data type available in Matlab and should not be overused.

6 Validating Input and Debugging

The `linreg_ols` function is well documented and provides the bare bones for linear regressions. However, it is missing input validation. What should `linreg_ols` return if the matrix X has more columns than lines? Or if y and X do not have the same number of rows?

6.1 Validating Input

We can validate data with conditionals, and if something fails the validation, we throw an error. In Matlab, we can throw an error with the function `error`, which takes a string as input and displays it to the user while interrupting the execution of the code. We can add the following lines to the beginning of the `linreg_ols` function:

```
% linreg_ols.m
function results = linreg_ols(y, X, add_intercept)
%% Input validation
if size(y, 1) ~= size(X, 1)
    error("y and X have a different number of rows")
end
if size(X, 1) < size(X, 2)
    error("X has less rows than columns")
end
%% Code to run after inputs are validated ...
end
```

If the inputs fail the validation, the code is interrupted and a message is displayed in red so that the user can take action to fix the issues. It is important that the message is clear and identifies the issue as directly as possible.

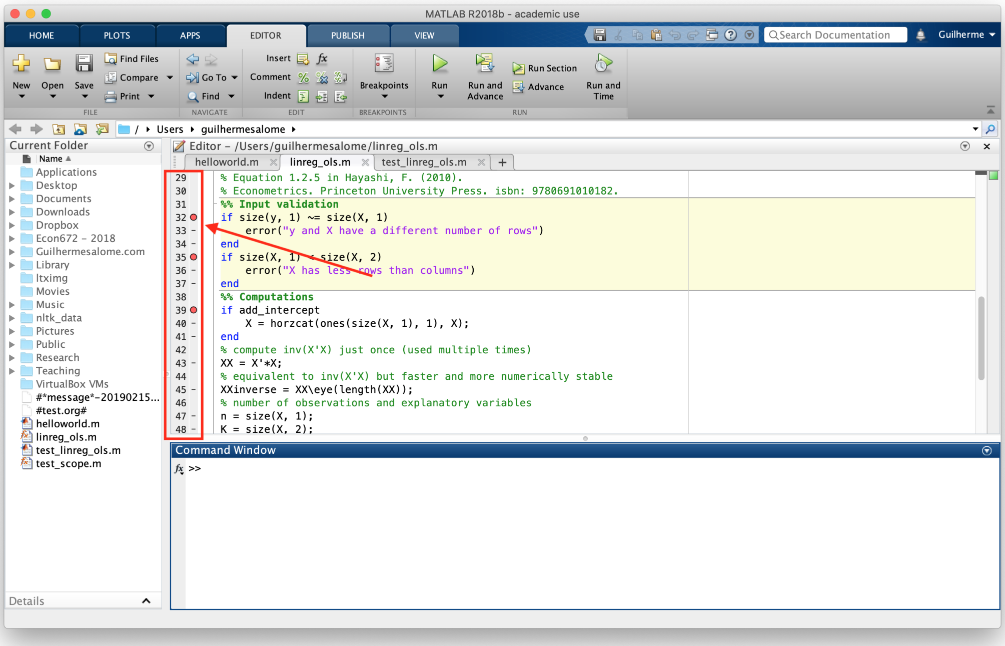
Observe the use of two percentage symbols instead of one. Lines that start with `%%` are also comments, but Matlab consider these lines as the start of a section. A section is just a tool to organize code in `.m` files and has no effect on the code.

Now, what happens if we forget to specify whether we want an intercept? That is, we execute `linreg_ols(y, X)`. In this case, `add_intercept` is a required input, so Matlab will warn you that there are not enough input arguments. We can see what happens in detail by using the `debugging` features Matlab provides.

6.2 Debugging

Debugging in the Matlab IDE is done by setting breakpoints, and then executing your code normally. When the code reaches a breakpoint, the execution is paused and you have access to the variables available at that point in your code. You can then choose to continue executing the code in different ways. You can continue the execution line by line, continue until where the cursor is at, or continue until the code finishes or finds another breakpoint.

You can add a breakpoint by clicking right next to the number line in the code editor. See Figure 4 on how to add a breakpoint. When you click a red circle will show up indicating that a breakpoint was added. You can remove breakpoints by clicking on the red circles.

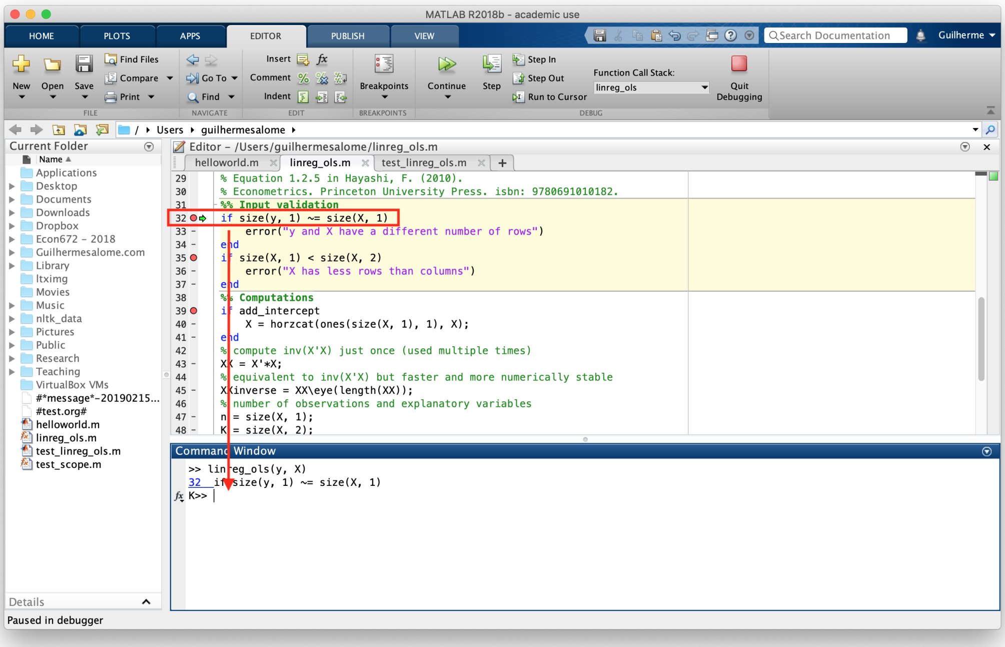


```
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn: 9780691010182.
%% Input validation
if size(y, 1) ~= size(X, 1)
    error("y and X have a different number of rows")
end
if size(X, 1) < size(X, 2)
    error("X has less rows than columns")
end
%% Computations
if add_intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
```

```
>>
```

Figure 4: Debugging in Matlab by Adding Breakpoints.

Now, in the `Command Window` execute `linreg_ols(y, X)`. The code will start executing and it will stop at the first breakpoint. A green arrow will indicate where the execution paused, and at the `Command Window` you will be able to run any code and inspect the values of the variables. See Figure 5 where the code was paused at the first breakpoint.



```
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn: 9780691010182.
%% Input validation
if size(y, 1) ~= size(X, 1)
    error("y and X have a different number of rows")
end
if size(X, 1) < size(X, 2)
    error("X has less rows than columns")
end
%% Computations
if add_intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
```

```
>> linreg_ols(y, X)
32  if size(y, 1) ~= size(X, 1)
K>> 
```

MATLAB R2018b - academic use
Editor - /Users/guilhermesalome/linreg_ols.m
Step
Continue
Breakpoints
Function Call Stack: linreg_ols
Paused in debugger

Figure 5: Debugging: Execution Paused at Breakpoint.

Notice that even though we called `linreg_ols` without supplying `add_intercept`, the function still executes. You can continue the execution by clicking on the button "Step", see Figure 6.

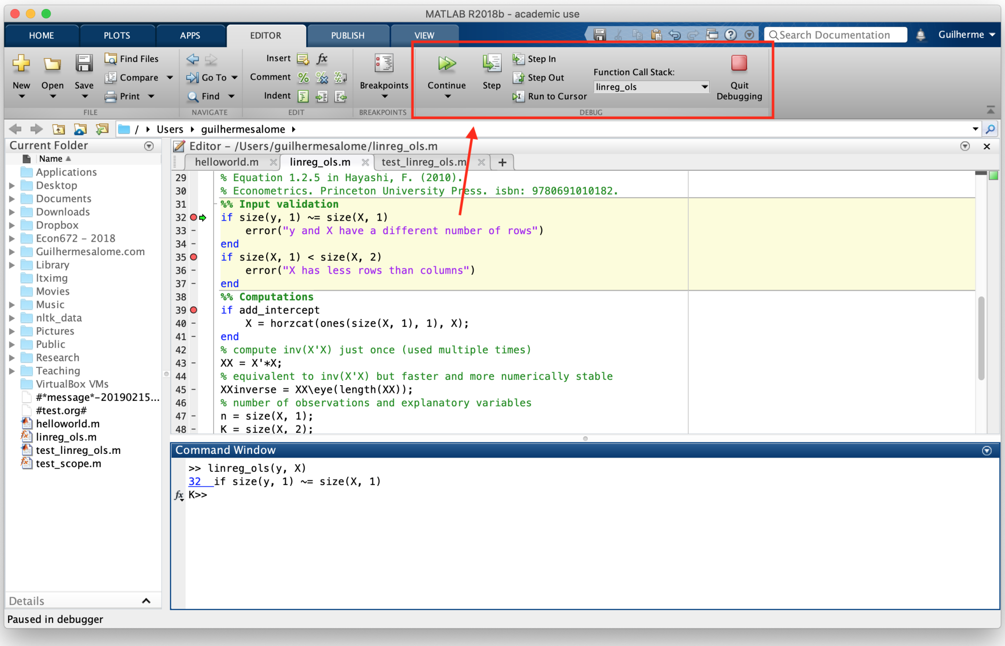
Keep clicking on "Step" until the code is paused at the third breakpoint. At this time, the variable `add_intercept` is not defined, and if you continue the execution you will get an error message.

Debugging is useful when dealing with large amounts of code and provides a way to ascertain the value of variables in your program. The code for `linreg_ols` will run even if the `add_intercept` variable is not passed, but will raise an error as soon as this variable is required. We can modify our code to allow `add_intercept` to be truly an optional variable, so that if it is omitted some default behavior will occur.

6.3 Optional Variables and Default Values

In Matlab, we can add optional variables to a function by passing the keyword `varargin` as an input. Then, inside the function, a variable named `varargin` is available and contains all optional arguments passed to it. Since optional arguments could be of any type, `varargin` must be able to hold different types of data.

In Matlab, a `cell array` is a data type that can hold data of various data types, and is the data type used for `varargin`. A `cell array` is similar to a number array and is indexed in the same way, but each element of a `cell array` is a `cell`, and a `cell` can have any data type. That is, a single `cell` can hold a number, a character, a string, or even other arrays. We can create a `cell array` with braces `{ }`:



```
% Equation 1.2.5 in Hayashi, F. (2010).
% Econometrics. Princeton University Press. isbn: 9780691010182.
%% Input validation
if size(y, 1) ~= size(X, 1)
    error("y and X have a different number of rows")
end
if size(X, 1) < size(X, 2)
    error("X has less rows than columns")
end
%% Computations
if add_intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
```

```
>> linreg_ols(y, X)
32  if size(y, 1) ~= size(X, 1)
K>>
```

Figure 6: Debugging: Continue Execution.

```matlab
% create a cell array
data = {"Parameters", [1, 2, 3]}
% use () to refer to cells, not their contents
data(1, 1)
data(1, 2)
% to access the actual contents of a cell index with {}
data{1, 1}
data{1, 2}
```

We can modify `linreg_ols` to accept optional arguments using `varargin`. Inside `linreg_ols`, the variable `varargin` is a cell array holding any extra inputs passed to the function. If no extra input was passed, then `varargin` is an empty cell array. We can check whether `varargin` is empty with the function `isempty`. To let the default behavior of the function be adding the intercept, we can verify if `varargin` is empty, and if it is we add the intercept.

```matlab
% linreg_ols.m
function results = linreg_ols(y, X, varargin)
%% Input validation
if size(y, 1) ~= size(X, 1)
    error("y and X have a different number of rows")
end
if size(X, 1) < size(X, 2)
    error("X has less rows than columns")
end
%% Add Intercept if Required
if isempty(varargin) || (varargin{1} == true):
    X = horzcat(ones(size(X, 1), 1), X);
end
%% Computations
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically
   stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
beta = XXinverse*X'*y;
% estimate Var(beta_hat|X)
e = y - X*beta;
s2 = (e'*e)/(n - K);
stderr = diag((s2*XXinverse)^0.5);
% t-statistics
t_stat = beta./stderr;
p_value = (1 - tcdf(abs(t_stat), n - K))*2;
% create results struct
results = struct('b', beta, 's2', s2, 'stderr', stderr, ...
                 't_stat', t_stat, 'p_value', p_value);
end
```

It is possible to tie together optional inputs and input validation.

6.4 Parsing Inputs

Matlab has a built-in way to do input parsing and allowing optional inputs: `inputParser`. The `inputParser` is an object created inside a function, which takes care of parsing inputs.

We use the `inputParser` by first creating a `parser object`. Then, we tell this `parser object` what are the inputs the function should receive. For example, we tell the `parser object` it must receive a variable named y, it must receive a variable named X, and that it could receive a variable named `intercept`. We can also tell the `parser object` what conditions each variable must satisfy for it to be accepted and the function to run normally. We then tell the `parser object` to parse the inputs, and the parsed inputs will be available in the `parser object` as a `struct`. Let's do an example:

```
% example_inputparser.m
function example_inputparser(arg1, arg2, varargin)
% create the parser object
parser = inputParser;
% add a required argument to the parser: arg1
addRequired(parser, 'arg1');
% add a required argument to the parser: arg2
addRequired(parser, 'arg2');
% add an optional argument to the parser
addOptional(parser, 'intercept', true);
```

```
% actually parse the inputs
parse(parser, arg1, arg2, varargin{:});
inputs = parser.Results;
% display the inputs
disp({inputs.arg1, inputs.arg2, inputs.intercept});
end
```

The `inputParser` code above is quite more extensive than what we had before, but the code itself is also very clear on what it is doing (it is self-documenting). We create the `parser` object, and add two required arguments with the `addRequired` function. The `addRequired` function informs the `parser` object that it must receive two variables, the first variable should receive the name `arg1` and the second variable should receive the name `arg2`. Then, we call `addOptional` to inform the `parser` object that it can receive a third variable, and if it does, this variable should be named `intercept`. If no third variable is received, then the variable `intercept` should be created and its value should be `true`. Test the function by calling:

```
example_inputparser(1, 2)
example_inputparser(1, 2, false)
example_inputparser()
```

The functions `addRequired` and `addOptional` define inputs that must appear in the correct order (positional arguments). While `addRequired` defines required inputs, `addOptional` defines optional inputs, but both inputs are positional arguments. However, we can also define inputs that can come in different order, but to do so these inputs must also be named (name-value pair arguments). We can add these name-value pair inputs with the `addParameter` function:

```
% example_inputparser.m
function example_inputparser(arg1, arg2, varargin)
% create the parser object
parser = inputParser;
% add a required argument to the parser: arg1
addRequired(parser, 'arg1');
% add a required argument to the parser: arg2
addRequired(parser, 'arg2');
% add an optional argument to the parser
addOptional(parser, 'intercept', true);
% add a name-value pair input to the parser
addParameter(parser, 'cov_type', 'standard');
% actually parse the inputs
parse(parser, arg1, arg2, varargin{:});
inputs = parser.Results;
% display the inputs
disp({inputs.arg1, inputs.arg2, inputs.intercept, inputs.
   cov_type});
end
```

The order for inputs is always the following: required arguments, optional arguments, name-value pairs.

The advantage of using `inputParser` is that we can also do input validation at the same time. Each of the functions `addRequired`, `addOptional` and `addParameter` can take a validation function that is used to check whether the inputs are valid. Let's add input validation to `example_inputparser`:

```matlab
% example_inputparser.m
function example_inputparser(arg1, arg2, varargin)
% create the parser object
parser = inputParser;
% add a required argument to the parser: arg1
addRequired(parser, 'arg1', @validate_arg1);
% add a required argument to the parser: arg2
addRequired(parser, 'arg2', @validate_arg2);
% add an optional argument to the parser
addOptional(parser, 'intercept', true, @validate_intercept);
% add a name-value pair input to the parser
addParameter(parser, 'cov_type', 'standard', @validate_cov_type);
% actually parse the inputs
parse(parser, arg1, arg2, varargin{:});
inputs = parser.Results;
% display the inputs
disp({inputs.arg1, inputs.arg2, inputs.intercept, inputs.cov_type});
end

%% Functions for input validation
% these functions must return a boolean value or throw an error
function bool = validate_arg1(arg1)
    bool = isnumeric(arg1);
end

function bool = validate_arg2(arg2)
    bool = isnumeric(arg2) && size(arg2, 1) >= size(arg2, 2);
end

function bool = validate_intercept(intercept)
    bool = islogical(intercept) && length(intercept) == 1;
end

function bool = validate_cov_type(cov_type)
    covs = ["standard", "White", "HAC"];
    bool = ischar(cov_type) && ismember(cov_type, covs);
end
```

The functions declared in the section "Functions for input validation" are passed as arguments to the functions `addRequired`, `addOptional` and `addParameter`. To pass a function as an argument to another function we use the `@` operator in front of the function name. When we use the `@` syntax with the function name we are creating a function handle. A function handle allows a function to be passed around as any other variable, and calling a function handle is the same as calling the function itself.

Notice that most of the functions above have just one line of code. We can simplify the code by using anonymous functions.

6.5 Anonymous Functions

An anonymous function is a function that is written in a single line and is itself a function handle. We can create an anonymous function using the syntax `@(inputs) code;`. For example:

```matlab
% create an anonymous function and assign to a variable
square_root = @(x) x.^0.5;
% call the anonymous function
square_root(4)
% since the anonymous function is a function handle
% we can pass it as inputs to other functions
integral(square_root, 0, 1)
% we can create an anonymous within an anonymous function
poly_integral = @(a) integral(@(x) (a*x.^2 + x + 2), 0, 1);
```

The validating functions in `example_inputparser` can be substituted by anonymous functions, which makes the code clearer. Let's modify `example_inputparser` to make use of anonymous functions:

```matlab
% example_inputparser.m
function example_inputparser(arg1, arg2, varargin)
% create the parser object
parser = inputParser;
% add a required argument to the parser: arg1
addRequired(parser, 'arg1', @isnumeric);
% add a required argument to the parser: arg2
addRequired(parser, 'arg2', @(x) (isnumeric(x) && size(x, 1)
    >= size(x, 2)));
% add an optional argument to the parser
addOptional(parser, 'intercept', true, @(x) (islogical(x) &&
    length(x) == 1));
% add a name-value pair input to the parser
addParameter(parser, 'cov_type', 'standard', @
    validate_cov_type);
% actually parse the inputs
parse(parser, arg1, arg2, varargin{:});
inputs = parser.Results;
% display the inputs
disp({inputs.arg1, inputs.arg2, inputs.intercept, inputs.
    cov_type});
end

%% Functions for input validation
```

```
% these functions must return a boolean value or throw an
   error
function bool = validate_cov_type(cov_type)
    covs = ["standard", "White", "HAC"];
    bool = ischar(cov_type) && ismember(cov_type, covs);
end
```

We can make use of these features to validate the input for `linreg_ols`:

```
% linreg_ols.m
function results = linreg_ols(y, X, varargin)
%% Input validation
parser = inputParser;
addRequired(parser, 'y', @isnumeric)
addRequired(parser, 'X', @validate_X);
addParameter(parser, 'intercept', true, @islogical);
parse(parser, y, X, varargin{:});
y = parser.Results.y;
X = parser.Results.X;
if size(y, 1) ~= size(X, 1)
    error("y and X have a different number of rows")
end
%% Add Intercept if Required
if parser.Results.intercept
    X = horzcat(ones(size(X, 1), 1), X);
end
%% Computations
% compute inv(X'X) just once (used multiple times)
XX = X'*X;
% equivalent to inv(X'X) but faster and more numerically
   stable
XXinverse = XX\eye(length(XX));
% number of observations and explanatory variables
n = size(X, 1);
K = size(X, 2);
beta = XXinverse*X'*y;
% estimate Var(beta_hat|X)
e = y - X*beta;
s2 = (e'*e)/(n - K);
stderr = diag((s2*XXinverse)^0.5);
% t-statistics
t_stat = beta./stderr;
p_value = (1 - tcdf(abs(t_stat), n - K))*2;
% create results struct
results = struct('b', beta, 's2', s2, 'stderr', stderr, ...
                 't_stat', t_stat, 'p_value', p_value);
end

%% Functions for Input Validation
```

```
function bool = validate_X(X)
    bool = isnumeric(X);
    if size(X, 1) < size(X, 2)
        error("X has less rows than columns");
    end
end
```

We can now call `linreg_ols` with or without a name-value pair specifying the inclusion of an intercept:

```
linreg_ols(y, X)
linreg_ols(y, X, 'intercept', true)
```

7 Importing Data and Saving Results

We will now use our `linreg_ols` function with a real data set. Download this data set on housing prices in California, and save the file in your working folder. This data set was first used in Pace and Barry (1997), and was later modified for use in Géron (2017). The data contains information on household prices in California in 1990 (based on the census at that time). The first line of the file contains the name of the explanatory variables.

7.1 Importing Data

Matlab has various built-in functions for importing data of different types. We can import `.csv` files to Matlab with the `readmatrix` function[4].

```
data = readmatrix('housing.csv');
```

The default behavior of `readmatrix` is to infer the type of the file from its extension, in this case `.csv`. Knowing the file extension, the function knows that the values are comma-separated, and the function also attempts to detect if there is a header line and the number of variables (columns) to be imported. The `housing.csv` data is mostly numeric, with the exception of the last column. The function `readmatrix` imports the data into a numeric matrix, which works well for the first 9 columns of `housing.csv`. However, the last column of the file contains categorical data, which is not automatically converted to numbers, and so Matlab treats all its values as `NaN`. If we are not interested in using the last column we could work with `readmatrix`.

The resulting `data` matrix has all the information from `housing.csv`, with the exception of the names of the variables and the last column of the file. We can keep all the columns of the data and the names of the variables if we import the data to a `table` instead of a numeric array. To do so, we use the function `readtable`. As with `readmatrix`, the function `readtable` can figure out the delimiter, header and number of variables from the file extension, but instead of loading the data into a numeric array, it loads the data into a `table`.

[4]There is also a GUI interface for importing data under the menu tab `Home`, via the button `Import Data`. Older versions of Matlab have different functions for importing data. Before the R2019a release, for example, `.csv` files could be imported with the function `csvread`. You can still use `csvread`, but `readmatrix` should be preferred since it is an improved version.

```matlab
% import housing data
data = readtable('housing.csv');
% verify that everything was imported correctly
data(1:5, :)
% notice that the names of the variables are also
% displayed, and that the last is correctly imported

% data in a table can be accessed by normal indexing as in a
% numeric array
data(1:10, 1)
data(1:10, 2)
data(1:10, 10)
% but we can also access the data like in a struct, where the
    field
% names are the variable names
data.longitude
data.longitude(1:5)
data.ocean_proximity(1:5)
% notice that the ocean_proximity data is stored as character
% arrays in cells
% we can convert it to a string array:
data.ocean_proximity = string(data.ocean_proximity);
data.ocean_proximity(1:10)
% we can see all the variable names in a table via its
   properties
data.Properties.VariableNames
% we can access the columns of the table by index with the
   names
data(1:5, 'longitude')
% for multiple columns we pass a string array
data(1:5, ["population", "households", "median_income"])
% similar to cells, when we index with parentheses we do not
   get the
% actual values stored in a table, but we get a subtable
% to get the actual values we use the curly brackets notation
data{1:5, ["population", "households", "median_income"]}
```

The data has a few missing values (added on purpose). We can count how many lines have missing values with the functions `isnan` and `sum`. The function `isnan` takes a numeric array and finds all `NaN`. The function `sum` can be used to sum across values in rows or columns (or both) of a matrix.

```matlab
% dimensions of table
size(data)
% find missing data no the first column
sum(isnan(data{:, 1}))
% find missing data on all columns, except the last one
sum(isnan(data{:, 1:9}))
```

```
% we see that there are 207 missing data points for the
% total_bedrooms variable
% indeed:
sum(isnan(data.total_bedrooms))
% let's remove these rows from the table
% first we need to find the indices for the rows with missing
    values
missing_values = isnan(data.total_bedrooms);
% missing_values is a boolean array, it has 1's where there
   are
% NaNs in the data.total_bedrooms array
% we can use this logical (boolean) array to index the
   original
% table and select only the rows without NaN's
data_no_missing = data(~missing_values, :);
% this gives us a new table, from which we can extract data
   for the regression
```

We can also remove rows with missing values (represented by `NaN`) automatically using the function `rmmissing`:

```
size(data)
data_no_missing = rmmissing(data);
size(data_no_missing)
```

Let's run a regression of the logarithm of the median house value on the logarithm of the number of bedrooms in the house.

```
results = linreg_ols(log(data.median_house_value), ...
                     log(data.total_bedrooms));
disp(results.b);
```

7.2 Saving Results, Cleaning the Workspace and Loading Variables

Now that we can generate results from data, we need to save these results. Saving and loading variables in Matlab can be done with the functions `save` and `load`. There are several ways of using the function `save`:

```
% Save ALL variables in the workspace
save('all_workspace_variables');
% The above creates a file named all_workspace_variables.mat
% .mat files can be easily imported with the load function

% Save only ONE variable
save('housing_clean', 'data_no_missing');
% saves the variable data_no_missing on the file
   housing_clean.mat

% Save MULTIPLE variables
```

```
save('data_and_results', 'data_no_missing', 'results');
% saves the variables data_no_missing and results in
% the file data_and_results.mat
```

The files created with the function `save` have the `.mat` extension and are binary files.

Let's now clear our workspace to and reload the variables. First, clear the command window by executing `clc`. This command simply clears the `Command Window` so that you have a clear screen to type commands. You can still use the arrow keys to execute previous commands. Second, use the special command `clear` to delete all variables from the workspace. After executing this command the variables previously created are gone. It is also possible to clear specific variables by executing `clear variable_name`. Third, use `load` to load the `.mat` files, which will take the variables stored in those files and make them available in the workspace.

```
% Load ALL variables from before
load('all_workspace_variables.mat');
% Load only the HOUSING variable from before
clear;
load('all_workspace_variables.mat', 'housing');
% Notice we pass the name of the variable to load so that
    only that
% variable is loaded
% Load the cleaned housing data
clear;
load('housing_clean.mat');
% Load the cleaned housing data and the regression results
clear;
load('data_and_results.mat');
% Load only the regression results
clear;
load('data_and_results.mat', 'results');
```

Notice that the variables are loaded with their original names.

8 Summary

We have covered the basics of the Matlab programming language. You should be able to:

- Create `.m` script files to run code
- Work with vectors and matrices
- Do basic computations and linear algebra with numeric arrays
- Create function files
- Use conditionals
- Use for-loops and while-loops
- Generate data from statistical distributions

- Create arrays for text data
- Present data in a neat manner using `table`
- Organize function results in `struct`
- Validate input variables with conditionals
- Debug code with the built-in debugger
- Let functions accept any number of inputs with `varargin`
- Use `inputParser` to parse input variables and specify default values
- Pass functions as arguments with function handles
- Simplify small functions with anonymous functions
- Import `.csv` data
- Clean rows from a table with missing values
- Save variables to `.mat` files
- Clear the workspace
- Load variables stored in `.mat` files

9 Assignment

All assignments should be submitted to the Github repository you have been assigned to. The deadline is July 26th by midnight. You should write a report in Latex with the solutions to the problems below. If the problem requires you to code, then the code should also be included in the report. For a quick guide on how to add Matlab code to your Latex files, refer to the Section "Adding Matlab Code" on this tutorial.

Problem 1 *Update the code for generating data to function without loops (vectorization). To do so, you will need to read the documentation for the random number generator functions we have used: `rand` and `randn`. After vectorizing the problem, verify you get the same estimates for the parameters as before. For this part, you will need to fix the generation of random numbers, so that you generate the same random numbers when you run your code without vectorization and with vectorization. Read the documentation for the `rng` function, and if you need more details read about managing the global stream.*

Problem 2 *Extend `linreg_ols` to also output the R^2 of the regression (see Equation 1.2.18 in Hayashi (2000).*

Problem 3 *Run the linear regression suggested in Equation (8) of Pace and Barry (1997).*

Problem 4 *Matlab has been around since 1984 and has many packages for common problems. Figure out if Matlab has a function for estimating the parameters of a linear regression using OLS. If it does, what is the name of the function? How does the function actually estimate the coefficients? To answer this, you need to look at the code of the function. You can do so with the special command* `type`*. For example, if the name of the function is* `ols`*, then you can execute* `type ols` *to see the source code.*

Problem 5 *When the classical assumption of homoskedasticity fails, we need a different estimators for the standard errors of the OLS estimates. White (1980) proposes a heteroskedasticity-robust estimator for the standard errors of the OLS estimates, which is now known as White's standard error. Equation 2.4.1 in Hayashi (2000) shows White's standard error:*

$$\widehat{SE(\hat{\beta}_i)} \equiv \sqrt{\frac{1}{n}[S_{xx}^{-1}\hat{S}S_{xx}^{-1}]_{[i,i]}}$$

There is a slight change of notation in this part of the Hayashi (2000), and x_i' is the ith row of X (x_i is a column vector with the explanatory variables in the ith row of X). The term S_{xx} is the sample mean of $x_i x_i'$: $S_{xx} = \frac{1}{n}\sum_{i=1}^{n} x_i x_i'$ (Equation 2.3.6 in Hayashi (2000)). The term $\hat{S}$ is an estimator for a matrix of fourth moments, and it is defined as $\hat{S} = \frac{1}{n}\sum_{i=1}^{n} e_i^2 x_i x_i'$ (Equation 2.5.1 in Hayashi (2000)), and e_i is as before (residual for the ith observation). Implement this estimator in the `linreg_ols` *function. You could write a local function to estimate White's standard error in the* `linreg_ols.m` *file.*

Problem 6 *(Optional) Implement the t-test based on White's standard error. Also, compute the p-value from given the statistic.*

Problem 7 *Newey and West (1987) proposes another estimator to the standard errors for the OLS estimates under weaker assumptions. The estimator the authors propose is robust not only to heteroskedasticity, but also autocorrelation, and is known as the HAC standard errors. Does Matlab have a function to compute the HAC standard errors? If so, extend* `linreg_ols` *to also compute the HAC standard errors.*

Problem 8 *(Optional) Implement the t-test based on the HAC standard error. Also, compute the p-value from given the statistic.*

Problem 9 *Extend* `linreg_ols` *to accept an optional input named* `cov_type`*, which specifies the type of standard errors estimator to report. Use* `inputParser` *to do so.*

References

Géron, Aurélien (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.". URL: `https://isbnsearch.org/isbn/9781491962299`.

Hayashi, F. (2000). *Econometrics.* Princeton University Press. ISBN: 9780691010182. URL: `https://books.google.com/books?id=QyIW8WUIyzcC`.

Newey, Whitney K. and Kenneth D. West (1987). "A simple, positive semi-definite, heteroskedasticity and autocorrelation consistent covariance matrix". In: *Econometrica* 55.3, p. 703. URL: `https://login.proxy.lib.duke.edu/login?url=https://search-proquest-com.proxy.lib.duke.edu/docview/214867593?accountid=10598`.

Pace, R Kelley and Ronald Barry (1997). "Sparse spatial autoregressions". In: *Statistics & Probability Letters* 33.3, pp. 291–297. URL: `https://doi.org/10.1016/S0167-7152(96)00140-X`.

White, Halbert (1980). "A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity". In: *Econometrica* 48.4, pp. 817–838. URL: `https://www.jstor.org/stable/1912934`.