Macroeconomía III

Práctica Dirigida #6

Laboratorio #1

Guía Rápida de Matlab*

Jason Cruz
jj.cruza@up.edu.pe

*Basada en la colaboración de (Mario Alloza, 2015)

Índice

1. Introducción

Matlab es un lenguaje de programación que permite una gran variedad de cálculos numéricos y es especialmente potente para realizar manipulaciones con matrices. La interfaz de usuario incluye las siguientes ventanas:

Command Window

Este espacio tiene una doble función:

1. Permite introducir cualquier entrada (comandos dentro de las instrucciones).
2. Muestra el resultado de cualquier operación solicitada.

Workspace

Aquí se almacenan todos los objetos creados, a los que se puede acceder haciendo doble clic en sus iconos o escribiendo el nombre abierto del objeto en la ventana de comandos. En ambos casos aparecerá una hoja de cálculo tipo Excel (la ventana del editor de variables) con los valores del objeto.

Current Folder

Muestra los archivos que contiene la carpeta actual. Define la carpeta de trabajo pues solo se ejecutarán los archivos que se encuentren en ella. Se debe cambiar la ruta cuando sea conveniente.

Editor

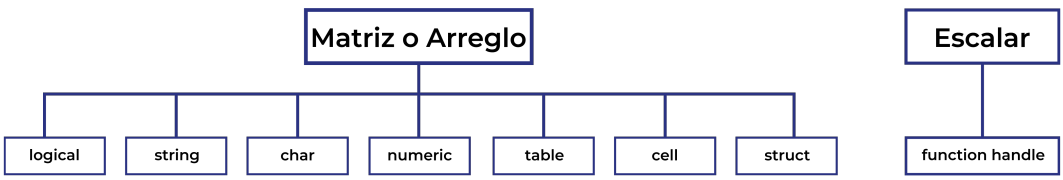
En la mayoría de los casos, en lugar de utilizar el prototipo de ventana de comandos de las instrucciones, podemos preferir crear un archivo que contenga una serie de comandos o un programa creado por nosotros mismos. Estos archivos tienen la extensión .m y en adelante los llamaremos ***scripts***. El código escrito en el editor puede ejecutarse pulsando la tecla `F5` o haciendo clic en el icono del triángulo verde `(Run)`.

2. Tipos de variables en Matlab

Son atributos de los datos que indica a la computadora acerca de la clase de información que va a procesar. Esto incluye imponer restricciones en los datos, como qué valores pueden tomar y qué operaciones se puede realizar.

En Matlab, los tipos de variables fundamentales están dadas en forma de arreglos (*arrays*). Un arreglo puede crecer de tamaño desde $0x0$ (matriz nula de dimensión 2) hasta otro de cualquier tamaño.

Figura 1: Tipos de variables en Matlab



Puede ver más sobre tipos de variables aquí.

3. Comandos básicos

3.1. Matlab como una calculadora

Para realizar operaciones básicas con escalares, podemos tipearlos directamente en la *Command Window* y la salida se mostrará inmediatamente después de nuestro comando, como en una calculadora; por ejemplo:

```
5+7
20*3
3/5
ans^2
```

Matlab almacenará los resultados de estas operaciones en un objeto llamado `ans` que se sobrescribirá cada vez que escribamos un comando. Alternativamente, podemos definir un nombre para el resultado de una operación, por ejemplo:

```
myresult = 5+7
X = 13-11
Y = 4^X
```

Tenga en cuenta que Matlab distingue entre mayúsculas y minúsculas, por lo que `X` $\neq$ `x`.

3.2. Matrices

Construcción de matrices. Una matriz se puede definir en Matlab escribiendo uno a uno sus elementos separando las columnas por una coma (o un espacio) y saltando a la siguiente fila usando un punto y coma (o pulsando enter). Las matrices van siempre entre corchetes `[]`. Entonces, tenemos:

```
X = [3,7;1,4]
Z = [3 7
     1 4]
```

Donde X = Z, o X − Z es la matriz nula.

Podemos transponer una matriz (cambiando filas por columnas) añadiendo el símbolo `'` después de una matriz, como en:

```
X'
```

Podemos crear nuevas matrices "apendeando" dos o más matrices, o concatenando vectores o escalares:

```
M = [X Z']
N = [X [3;4]]
```

Operaciones con matrices. Como en el caso de las operaciones escalares, podemos realizar operaciones algebraicas con matrices:

```
Y = [4 9; 12 0]
X*Y
```

Para invertir una matriz, podemos utilizar el comando `inv()` o el operador "división". Tenga siempre en cuenta que la multiplicación de matrices no son conmutativas, es decir, que el orden sí importa.

```
inv(Z)
X*inv(Z)
X/Z
```

Podemos estar interesados en calcular el producto de Hadamard (producto punto) entre dos matrices de las mismas dimensiones. Obtenemos entonces una nueva matriz, digamos A, con elementos $A_{i,j} = X_{i,j} \cdot Y_{i,j}$, donde i indexa las filas, y j indexa las columnas.

```
X.*Y
X./Y
```

Tip: *ahora podemos resolver sistemas de matrices con las herramientas que hemos utilizado hasta ahora. Reordenamos el sistema para que tenga la forma:* `A*B = C` *donde* `A` *son los coeficientes,* `B` *las incógnitas y* `C` *números reales. Podemos resolver las incógnitas escribiendo:* `B = inv(A)*C`.

Acceso a los elementos de una matriz. Podemos acceder a los elementos de una matriz, para editarlos, o para crear una nueva variable a partir de ellos. Seleccionamos un elemento de una matriz escribiendo el número de fila y columna (entre paréntesis) justo después del nombre de la matriz:

```
element_1_2 = X(1,2)
X(1,2) = 654
X(end,2) = 7
```

Podemos acceder a un grupo de elementos utilizando el símbolo `:`. Así, `Q(1: 4,3)` selecciona los elementos de la primera a la cuarta fila de la tercera columna de la matriz Q. Si escribimos simplemente `Q(:, 3)`, se seleccionarán todos los elementos de la tercera fila. En nuestro ejemplo:

```
X(1:2,1) = 3
X(:,2) = Y(:,2)
```

Algunas matrices especiales. Aquí enumeramos algunos comandos relativos a matrices interesantes como la matriz identidad, o matrices que contienen siempre el mismo elemento:

```
eye(3)       % una matriz diagonal 3x3
X*eye(2)     % devuelve la matriz X
5*eye(3)   % una matriz diagonal 3x3 con 5s
zeros(3)     % una matriz 3x3 de ceros
ones(4)      % una matriz 4x4 de unos
diag(X)      % vector con los elementos diagonales
```

Tip: *un vector de unos puede añadirse fácilmente a una matriz existente de regresores cuando queremos estimar un modelo con constante: si tenemos T observaciones en una matriz X, podemos escribir* `X = [ones(T, 1) X]`.

3.3. Funciones integradas útiles

Una lista con algunos comandos prácticos:

Limpieza. `clear` limpia todos los objetos definidos durante la sesión de trabajo. `clc` limpia todos los comandos (y su salida) tipeados en la *Command Window*. `close all` cierra todas las ventanas abiertas mostrando figuras.

Vectores. Podemos crear vectores de números consecutivos utilizando el símbolo `:`. Si escribimos `n:m`, Matlab producirá un vector fila que contiene una lista de números $n, n+1, n+2, \ldots, m-2, m-1, m$. Podemos crear un vector columna transponiendo el vector fila. Además, podemos crear una lista de números $n, n+k, n+2k, \ldots, m-2k, m-k, m$ con el comando `n:k:m`:

```
1:6
(1:6)'
0:2:10
```

Del mismo modo, el comando `linespace (x1, x2, n )` creará un vector de filas de n números igualmente separados de `x1` a `x2` .

```
linspace(1,10,10)'
```

Tip: *cualquiera de los comandos anteriores, puede utilizarse para incorporar una columna (o fila) adicional a una matriz de regresores que contenga una tendencia lineal. ¿Cómo podríamos añadir una tendencia cuadrática a una matriz?*

Funciones elementales. Commands as `sqrt()` , `std()` , `exp()` o `log()` calcular la raíz cuadrada, la desviación típica, la exponencial o el logaritmo del argumento entre paréntesis:

```
sqrt(144)
std(1:6)
exp(1)
log(exp(1))
```

Tenga en cuenta que existen un montón de funciones incorporadas en Matlab como `media` , `min` , `max` , `suma` , `round` , ... Utilice el comando `help` para ver cómo funcionan.

Funciones de series temporales. Cuando se trata de datos de series temporales, las últimas versiones de Matlab incluyen comandos como `lagmatrix(Y, n)` , que crea una nueva matriz como Y, pero con un retraso de n períodos. Tenga en cuenta que al estimar un modelo de series de tiempo, es posible que desee incluir un número de rezagos, entonces puede sustituir n por una secuencia de rezagos como `1: n` . El comando `diff(Y, n)` calculará la diferencia n-ésima de la matriz Y. Ejemplos:

```
R = [4 7; 5 8; 9 3; 12 0; 1 5];
lagmatrix(R,1:3)
diff(R,1)
```

Tip: *vea más sobre series de tiempo aquí.*

Generadores de números aleatorios. En ocasiones, puede interesarnos generar números aleatorios. Esta funcionalidad se implementa en Matlab mediante el uso de los comandos `rand(m,n)` o `randn(m, n)` para generar matrices m veces n de números aleatorios siguiendo una distribución uniforme o una distribución $\mathcal{N}(0,1)$, respectivamente.

```
rand
rand(10,2)
randn(5)
```

Tip: ¿y si queremos extraer números de una distribución $\mathcal{N}(\mu,\sigma)$? Podríamos implementarlo suponiendo, por ejemplo $\mu = 5$ y $\sigma = 2$ usando `data=randn(10000,2) * 2+5;` . Para comprobarlo, escriba `media(data)` y `std(data)` .

Cambiar la forma de las matrices. Si estamos interesados en replicar la misma matriz un número de veces, podemos hacer uso de la función de producto de Kronecker construido en Matlab. Recordemos que $A \otimes B$ multiplica cada elemento de la matriz A por la matriz entera B, por lo tanto, si sustituimos A por una matriz de unos de orden n, estaríamos replicando la matriz B n veces. Lo mismo se puede conseguir utilizando el comando `repmat` :

```
X = [3,7;1,4]
kron(ones(2,2),X)
repmat(X,2,2)
```

También podemos vectorizar una matriz en Matlab. Así, una matriz $n \times m$ se convertiría en un vector $nm \times 1$, apilando todas las columnas de la matriz. Inversamente, el comando `reshape(X, n, m)` puede cambiar la forma de una matriz X a una nueva con n filas y m columnas.

```
vec_X = X(:)
reshape(vec_X,2,2)
```

Podemos determinar el tamaño de una matriz mediante la instrucción `size()`. Esta instrucción en particular, como muchas otras funciones implementadas en Matlab, entrega dos objetos diferentes como salida: un objeto que contiene el número de filas, y un objeto que contiene el número de columnas. Por lo tanto, tenemos que definir la salida (lo que está a la izquierda del símbolo =) en consecuencia:

```
[rows columns] = size(X)
```

Tip: el comando `length(X)` puede utilizarse alternativamente para calcular el tamaño máximo de la matriz X (es decir, en una matriz n veces m, devuelve n si $n > m$ o m en caso contrario).

4. Cargar y guardar

4.1. Importar datos

Matlab puede cargar datos en formato excel (*.xls* o *.xlsx* extensiones) mediante el comando `xlsread`. Tendremos que especificar primero el nombre del archivo (incluyendo su extensión) y la hoja donde se encuentran los datos (si sólo hay una hoja, se puede omitir este paso). Ambos argumentos deben ir encerrados entre `‘’`:

```
mydata = xlsread('GDPdata.xls','dataMatlab');
TIME = mydata(:,1);
GDP = mydata(:,2);
```

Es importante tener en cuenta que nuestra *Current Folder* (vea la sección 1) debe ser la que contiene el *script* que estamos intentando cargar. Cuando queramos importar datos en otro formato que no sea una hoja de cálculo, o cuando estemos utilizando una Mac (no podemos utilizar el comando anterior con ordenadores que no utilicen Microsoft Windows como sistema operativo) podemos utilizar el comando `importdata`:

```
data_imp = importdata('GDPdata.csv');
mydata = data_imp.data;
TIME = mydata(:,1);
GDP = mydata(:,2);
```

En este ejemplo hemos utilizado un archivo separado por comas (*.csv*), pero podríamos haber utilizado alternativamente otros tipos de archivos de texto. Observe que, cuando usamos el comando `importdata`, el objeto que se crea (`data_imp` en el ejemplo anterior) es lo que Matlab llama una “estructura”, que incluye tanto datos como texto. La segunda línea del código anterior extrae sólo el componente de datos de esta estructura.

4.2. Exportación de datos

La forma más sencilla de exportar datos es abrir la ventana del editor de variables (véase la descripción del *Work Space* en la sección 1) y, a continuación, copiar los valores y pegarlos en Excel (o en cualquier otro lugar). Esto puede hacerse de una forma más elegante utilizando los comandos `xlswrite` o `export`.

4.3. Cargar y guardar matrices

Los comandos `load` y `save` nos permiten crear o abrir un archivo que contenga todas o algunas de las matrices de nuestro Workspace. El archivo resultante sólo podrá ser leído por Matlab. Esto podría ser una solución potencial cuando se utilizan matrices muy grandes que consumen la memoria del ordenador; en ese caso, dividiendo una matriz grande en matrices más pequeñas que se guardarán y cargarán secuencialmente, podríamos realizar ciertas operaciones de forma más rápida.

5. Gráficos

Matlab es una herramienta especialmente adecuada para plotear datos. El comando básico para plotear un vector es `plot(Y)`. Sin embargo, también podemos escribir `plot(X,Y)` para plotear datos en el vector Y sobre el vector X:

```
X = (1:100)';
Y_1 = sin(X)
Y_2 = cos(X)+2
plot(X,Y_1)
```

Cuando ejecute un código/programa que trace múltiples figuras, tenga en cuenta que por defecto Matlab hará las figuras en la misma ventana, sobreescribiendo la figura anterior. Para aclarar las cosas, puede usar el comando `close all` para cerrar el archivo anterior o usar el comando `figure` antes del código relacionado al ploteo para pedir a Matlab que dibuje la nueva figura en una ventana separada. Estas ventanas pueden estar numeradas: `figure(1)`, `figure(2)` ...

```
figure
plot(X,[Y1 Y2])
```

Matlab implementa una amplia gama de tipos de gráfico, una característica que puede ser útil dependiendo de la naturaleza de los datos que queramos plotear:

```
close all
a = randn(1000,1);
hist(a)
```

Se pueden manipular las opciones de un gráfico: fuente, tamaño, colores, etc. A continuación mostramos un ejemplo de código para plotear un gráfico utilizando algunas de las características disponibles.

```
figure
plot(log(1:50),'--k','LineWidth',2)
hold on;
plot(log(1:50)+3,'-ro','MarkerEdgeColor','k', ...
'MarkerFaceColor','g')
plot(log(1:50)+1,'-b','MarkerEdgeColor','k', ...
'LineWidth',4)
title('Different Logarithmic Functions')
legend('log(x)','log(x)+3','log(x)+1')
xlabel('X')
ylabel('logarithmic function')
axis([0 45 0 10])
```

Guardando gráficos. Puede guardar cualquier gráfico que haya calculado. En la ventana de la figura, haga clic en `File` y, a continuación, en `Save as`. Podrás guardar el gráfico en una gran variedad de formatos: *.pdf*, *.eps*, *.png*, *.jpg*. Para hacerlo orgánicamente es necesario recurrir al código. Como ejemplo

tomemos el caso en donde asignamos el nombre de `figura_100` a nuestra gráfica y deseamos guardarla en formato `pdf`, en este caso el código sería `saveas(gcf, 'densidad_pbi_pc', 'pdf');`.

También puede utilizar el formato *.fig*, la extensión para figuras de Matlab, que le permitirá reabrir la figura guardada en Matlab y realizar operaciones con ella (por ejemplo: cambiar el aspecto de la figura).

6. *Loops*

Cuando se requiere repetir la misma instrucción varias veces, podemos utilizar un bucle para pedir a Matlab que ejecute esta instrucción repetida. El uso de bucles es muy común en programación, y puede simplificar enormemente el propio código.

6.1. *"For"*

Un bucle `for` simple. El bucle for repite la misma instrucción tantas veces como la definamos. La sintaxis es muy sencilla: la instrucción a repetir va precedida del comando `for index = first_iteracion :last_iteration` y va seguida de `end`. Por ejemplo:

```
for n=1:3
    n
end
```

6.2. *"While"*

Bucles anidados. Podemos anidar dos o más bucles `for` para ejecutar diferentes operaciones. En el siguiente ejemplo, estamos definiendo cada elemento de la matriz $W_{n,m}$ en la fila n y la columna m como $W_{n,m} = n^m$. Por lo tanto, utilizamos dos bucles `for` anidados para "escanear" cada elemento de la matriz W y calcular la operación (habríamos utilizado tres bucles para una matriz tridimensional):

```
W = zeros(5,4)
[rows columns] = size(W)
for m=1:columns
    for n=1:rows
        W(n,m) = n. m
    end
end
```

Ejemplo: generando un *Random Walk (RW)*. Los datos de series temporales artificiales (o cualquier operación que requiera un cálculo recursivo) pueden generarse utilizando un bucle `for`. En este ejemplo generamos la siguiente serie temporal: $y_t = y_{t-1} + \varepsilon_t$, donde $\varepsilon_t \sim \mathcal{N}(0,1)$ e $y_0 = 0$.

```
y = zeros(100,1);
eps = randn(100,1);
for i=2:100
    y(i,1) = y(i-1,1) + eps(i,1),
end
plot(y)
```

Un simple bucle `for` con sentencias condicionales. A menudo queremos calcular una instrucción condicional a alguna sentencia. En esta situación, podemos utilizar el comando

Este bucle, ejecutará la instrucción durante un número indefinido de veces hasta que se cumpla alguna condición. La sintaxis es muy similar a la del bucle `for`. Debes tener en cuenta algunas observaciones

cuando utilices este tipo de bucles. Primero, antes de que el bucle termine (es decir, antes de escribir `end`), tenemos que escribir una instrucción para decirle a Matlab que el bucle debe seguir ejecutándose antes de que no se cumpla la condición. Segundo, el bucle `while` puede estar ejecutándose un número infinito de veces si no se cumple la condición; si esto se debe a un error de programación hay que detener Matlab con `Ctrl+C`.

```
draw = rand
while draw <0.9
    display('The draw was below 0.9, try again.')
    draw = rand
end
```

7. Funciones

Comandos como `mean()` o `rand()` llaman a funciones ya incorporadas en Matlab. Sin embargo, podemos crear nuestras propias funciones para realizar algunas tareas específicas. De esta forma, simplificaremos mucho nuestro código al no tener que replicar código (en lugar de escribir dos veces el mismo conjunto de instrucciones, podemos simplemente llamar a nuestra función) ya que nuestro programa será más legible al tener menos líneas en el *script* principal (o *master script*), creando *scripts* separados con funciones para realizar tareas específicas.

Al escribir una función utilizaremos la siguiente sintaxis:

1. Comience el *script* con `function[output_1, output_2 ...] = function_name(input_1, input_2...)`.
2. Escriba todas las instrucciones que hacen uso de `input_1`, `input_2` ... especificadas por el usuario, las cuales producirán `output_1`, `output_2` ... Obviamente, debemos utilizar los mismos nombres para definir las variables de *output* que los que definimos en el punto anterior.
3. finalice el *script* de funciones con el comando `end`.

Al utilizar nuestra propia función, es importante tener en cuenta lo siguiente:

- El `function_name` especificado anteriormente, debe ser el mismo nombre que hemos utilizado para guardar el nuevo archivo `.m`. Si nuestra función se llama `compute_std_errors`, el nuevo *script* que contiene la función se guardará como `compute_std_errors.m`.
- Podemos llamar a la función desde nuestro *master script* escribiendo `[result_1, result_2...] = function_name (argument_1, argument_2...)`. Tenga en cuenta que escribimos `result` y `argument` en lugar de `output` y `input` para enfatizar que los nombres dados a estas variables no tienen por qué ser los mismos.

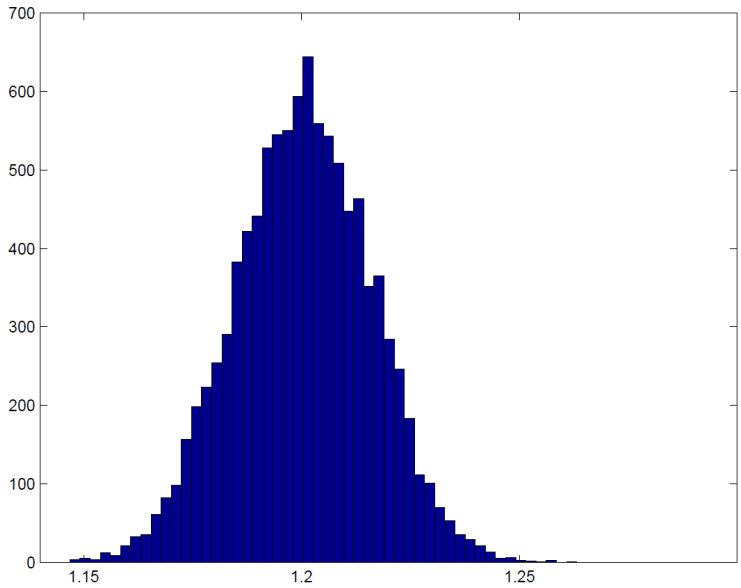
Para ilustrar el uso de funciones, el siguiente ejemplo crea datos artificiales y ejecuta una estimación OLS del modelo lineal clásico mediante una función definida por nosotros mismos, llamada `ols_estim`. Este ejemplo también pretende resumir algunos de los conceptos destacados mostrados en este breve tutorial. El *master script* luce así:

```
% OLS estimation of artificially generated data
% We assume we know the true DGP
T = 50000; % number of observations
```

```
eps = randn(T,1)*7; % errors
X = randn(T,1)*2 + 3; % regressors
X = [ones(T,1) X]; %include a constant
beta_true = [0.7 1.2]'; % true parameter
Y = X*beta_true + eps;
% Run the estimation
[beta_estim,sigma_estim] = OLSestim(X,Y);
display('The true parameters are:');
beta_true
display('The estimated parameters are:');
beta_estim
% Repeat the estimation B times
% for different draws of epsilon
B = 1000;
beta_mat = zeros(B,2);
for i=1:B
eps = randn(T,1)*7;
Y = X*beta_true + eps;
[beta_mat(i,:), sigma_mat(i,:)] = OLSestim(X,Y);
end

display('Mean estimation of the slope coeff.:')
mean(beta_mat(:,2))
hist(beta_mat(:,2),50)
```

Figura 2: Distribución del estimador (pendiente) en diferentes muestras.



El archivo `.m` que contiene la función `ols_estim` es:

```
function [beta_hat,sigma_hat] = OLSestim(XX,YY)
beta_hat = inv(XX'*XX)*XX'*YY;
% Alternatively, a more efficient code would be:
% beta_estimt = XnY;
residuals = YY - XX*beta_hat;
df = length(XX)-length(beta_hat);
sigma_sq = (residuals'*residuals)/df;
sigma_hat = sqrt(sigma_sq);
end
```

8. *Debugging*, cálculo eficiente y buenas prácticas

Desgraciadamente, nuestros propios programas no siempre funcionan bien y hay que detectar algunos errores de programación. A pesar de que Matlab ofrece algunas herramientas para hacer esta tarea menos dolorosa, escribir código "limpio" y rápido, puede ahorrarnos horas (¡o incluso días!). He aquí algunos consejos:

- Intente utilizar funciones para separar del código principal aquellas tareas que deban repetirse varias veces o que contengan código específico que pueda aislarse fácilmente frente al núcleo de nuestro programa. Idealmente, un proyecto Matlab incluirá un montón de programas/funciones que se llaman desde un master script *.m*.
- Sé generoso escribiendo comentarios (utilizando el símbolo %). A pesar de que es una cosa tediosa de hacer cuando se escribe código, resultará ser una práctica muy útil cuando vuelvas a tus proyectos después de algún tiempo. Este es el significado más básico de "documentar un código"[2]. Esto último se resume en: "mientras 'codeas', piensa que otros usuarios accederán a tu código y deben entenderlo al pie de la letra, como si se tratase de un recetario".
- Hazlo sencillo. Evita nombres complejos para las nuevas variables e intenta encontrar un método que te funcione.
- Intenta ser *"pythonic"*. Cuando defina variables o recurra a diversas etiquetas trate de ser usar solo minúsculas y, para diferenciar entre palabras, use guión bajo. Esto ayudará a que sus definiciones sean mucho más legibles y universales. El código de un *"pythonic"* luce como: `nombre_atributo`. Por supuesto, el nombre y atributo deben describir perfectamente a su variable.
- Hazlo de forma elegante. Hay muchas formas de abordar la misma tarea, sin embargo, las formas más elegantes tienden a hacer tu código más comprensible para los demás e incluso para ti mismo (por ejemplo, copiar y pegar varias veces algunas líneas de código pueden ser sustituidas por un bucle).
- Cálculo eficiente. Entre otras características de su ordenador, el rendimiento de Matlab está fuertemente correlacionado con el procesador de su ordenador. Sin embargo, los programas pueden ejecutarse mucho más rápido cuando evitamos tareas que son particularmente lentas para Matlab:
  - Matlab ama las matrices: cuando sea posible, utilice operaciones matriciales en lugar de bucles. Ejemplo: usar un bucle para ejecutar la misma instrucción para cada elemento de una matriz puede ser más lento que hacer una vez la instrucción para toda la matriz.
  - Cuando se ejecuta un bucle que almacena los resultados en una matriz, es más rápido definir que esta contenga sólo ceros antes de que comience el bucle, en lugar de dejar que el tamaño de la matriz crezca en cada iteración.
  - Algunos comandos son más rápidos que otros y producen los mismos resultados: `Y/X` es más rápido que `X*inv(Y)`.
  - Para comprobar el tiempo que Matlab tarda en formar una operación, puede utilizar `tic` y `toc` como en `tic; randn(10000,1); toc`. Este es el **tic-toc** que realmente importa a los pogramadores/economistas.

[2] Encuentre una guía super útil sobre buenas prácticas en el *coding* aquí. Este recurso es especialmente útil para la **organización de código** y **documentación**, habilidades que son esenciales en la investigación (posiciones de *Research Assistant*). Este recurso es versatil, de manera que le servirá para otros lenguajes de programación y *softwares*.

- Aprovecha las posibilidades de Matlab. Puede dividir su código en celdas (vea el menú de Matlab llamado `Celdas`) escribiendo dos símbolos de comentario %% para cada celda. Matlab le permite evaluar estas celdas en lugar de todo el programa, algo que puede ser útil en las primeras etapas de su proyecto.
- Al depurar, a veces es muy conveniente comprobar que Matlab está haciendo lo que usted cree que debería estar haciendo. Establezca puntos de interrupción (vea el menú de Matlab `Debugging`) o haga clic en la derecha de un número de línea en el código hasta que aparezca una bola roja) mientras que Matlab ejecuta todo el código hasta este punto y le permite avanzar paso a paso desde ese punto en adelante. Esto es particularmente interesante cuando se ejecuta un código que involucra funciones, ya que cuando Matlab está ejecutando una función, los objetos creados por esta no se almacenan a menos que se definan como *output*.
- También puede comparar que dos procedimientos alternativos producen los mismos resultados utilizando sentencias condicionales: sentencias verdaderas como `3==3` devolverán `1` como respuesta, mientras que sentencias falsas como `3==5` devolverán `0` como respuesta. Si quieres comparar dos vectores de resultados que parecen tener resultados similares pero no puedes comprobarlo debido a su gran dimensión, `min(my_result_1 == my_result_2)` debe ser `1`, de lo contrario, los dos métodos no son equivalentes para algunos casos.

9. Aprendiendo Matlab

Matlab implementa una gran variedad de funciones. Desde funciones, Matlab incorpora una documentación detallada de cada comando. Puede acceder a esta información escribiendo el comando `help unknown_command`.

Adicionalmente, use el menú `Help` para navegar y buscar todas las funciones implementadas para comprobar si Matlab ya incorpora lo que está buscando. Por ejemplo, no es necesario programar una descomposición de Choleski cuando el comando `chol()` ya lo hace.

Una de las mayores ventajas de Matlab es que es un *software* ampliamente utilizado. Eso significa que existe mucho código ya escrito. Algunos economistas comparten libremente su código lo que facilita mucho el aprendizaje. Visite, por ejemplo, las páginas web de Chris Sims, Larry Christiano o Martín Uribe para ver algunos ejemplos. Algunas revistas como **Review of Economic Dynamics** o algunos artículos de **American Economic Review** incluyen código que replica los resultados de los artículos publicados. Esa es una oportunidad perfecta para ver cómo luce un código profesional.