Taller de Modelización Matemática y Computacional en Biociencias

Manual introductorio a Octave

I. Introducción

Octave o GNU Octave es un programa y lenguaje de alto nivel[1] orientado principalmente al cálculo numérico.

Es un software libre bajo la GNU General Public Licence (GPL), esto significa que el código fuente de Octave está disponible y se le pueden hacer todos los cambios que se desee, siempre que el nuevo software creado respete esa misma condición. Otra característica de poder acceder al código fuente es que se puede saber exactamente como se lleva a cabo una determinada computación.

Posee una alta compatibilidad con MATLAB, por lo que los programas escritos en uno pueden usarse con el otro, siempre que se tomen en cuenta algunas diferencias. Al igual que MATLAB la representación de datos en Octave es en forma de matrices, por lo que posee gran funcionalidad para trabajar con ellas.

Octave posee una interfase gráfica y es desde allí que realizaremos todo el trabajo del curso.

Algunos de los usos de Octave son:

- Cálculo numérico
- Desarrollo de algoritmos
- Modelado, simulación y desarrollo de prototipos
- Análisis y visualización de datos
- Construcción de gráficas

Así como MATLAB posee toolboxes, paquetes de funciones creados por los usuarios para objetivos específicos, Octave posee el Octave Forge, allí se pueden encontrar paquetes en áreas como procesamiento de señales e imágenes, Redes Neuronales, etc.

En este manual introductorio esperamos que el alumno se familiarice con los comandos básicos de Octave de forma de poder realizar el análisis de los diferentes modelos que se verán a lo largo del Taller. Recomendamos abrir Octave en otra ventana e ir practicando los distintos comandos a medida que se leen estas notas.

II. Comandos Básicos

Al iniciar Octave se abrirá una ventana con diferentes paneles (Figura 1).

[1] Un lenguaje de programación de alto nivel se caracteriza por ser más cercano al entendimiento humano, por lo tanto son más fáciles de escribir y leer. Deben ser compilados para poder ser interpretados por una máquina. Ejemplos de lenguajes de alto nivel son: Pascal, C, Python.

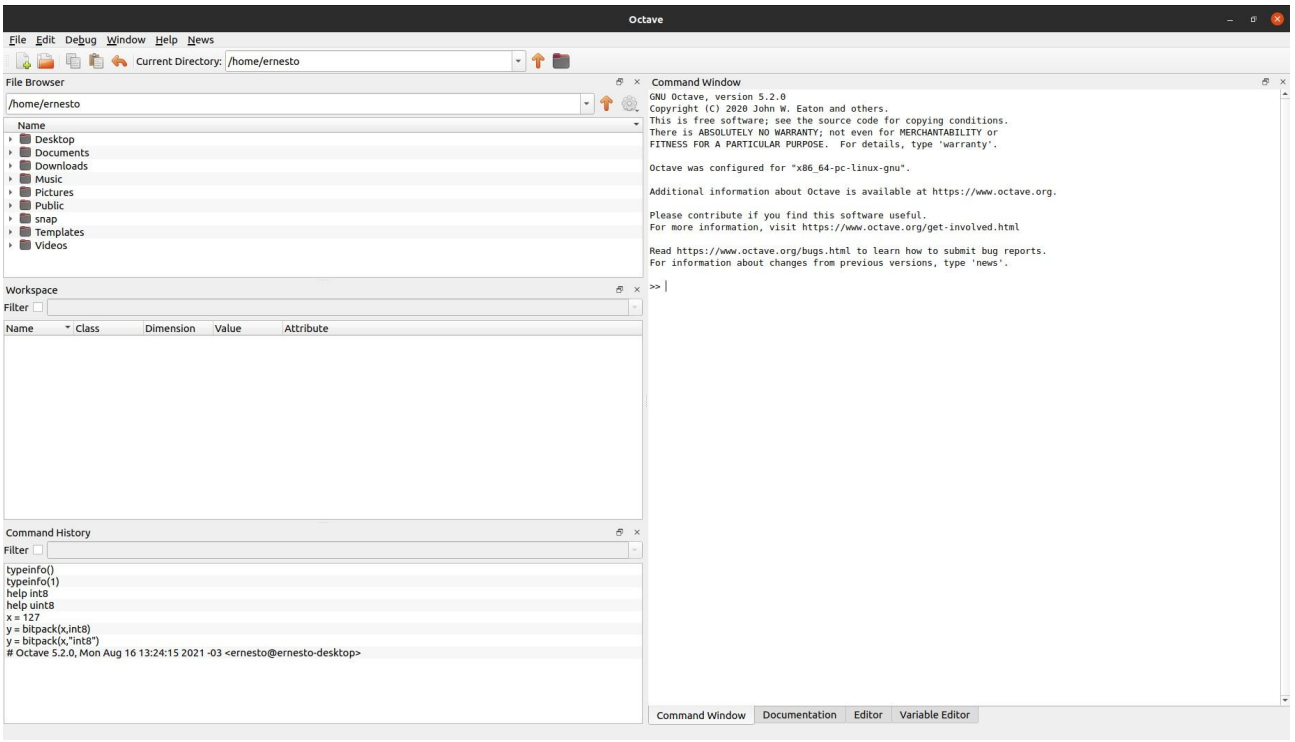

Figura 1: Ventana de inicio de Octave

- File Browser: Permite navegar por las carpetas y acceder a los archivos.
- Command Window: Ventana donde se introducen los comandos a ejecutar por el programa. Estos se escriben en la línea de comando indicada por el “prompt” (>>). 22
- Workspace: Permite observar los datos que se crean o se importan.
- Command History: Permite ver o volver a ejecutar los comandos introducidos en la línea de comando.

A. Navegar los directorios

Octave se inicia por defecto en el directorio indicado como “Current Directory”. Antes de comenzar a trabajar es conveniente cambiarse a una carpeta de trabajo. Esto se puede lograr utilizando las herramientas de la ventana “File Browser” o mediante comandos en la línea de comando.

Algunos comandos útiles para navegar los directorios desde la línea de comando:

cd (change directory): Cambia de directorio.

>>cd DIR

Cambia de directorio a DIR

mkdir: Crea un nuevo directorio.

>>mkdir DIR

Crea el directorio DIR.

Se recomienda a los estudiantes que experimenten con estos comandos para entender su lógica.

Un comando particularmente útil es *help*, que se utiliza seguido del comando sobre el que se quiera consultar información.

Por ejemplo:
>> help cd

Devuelve:

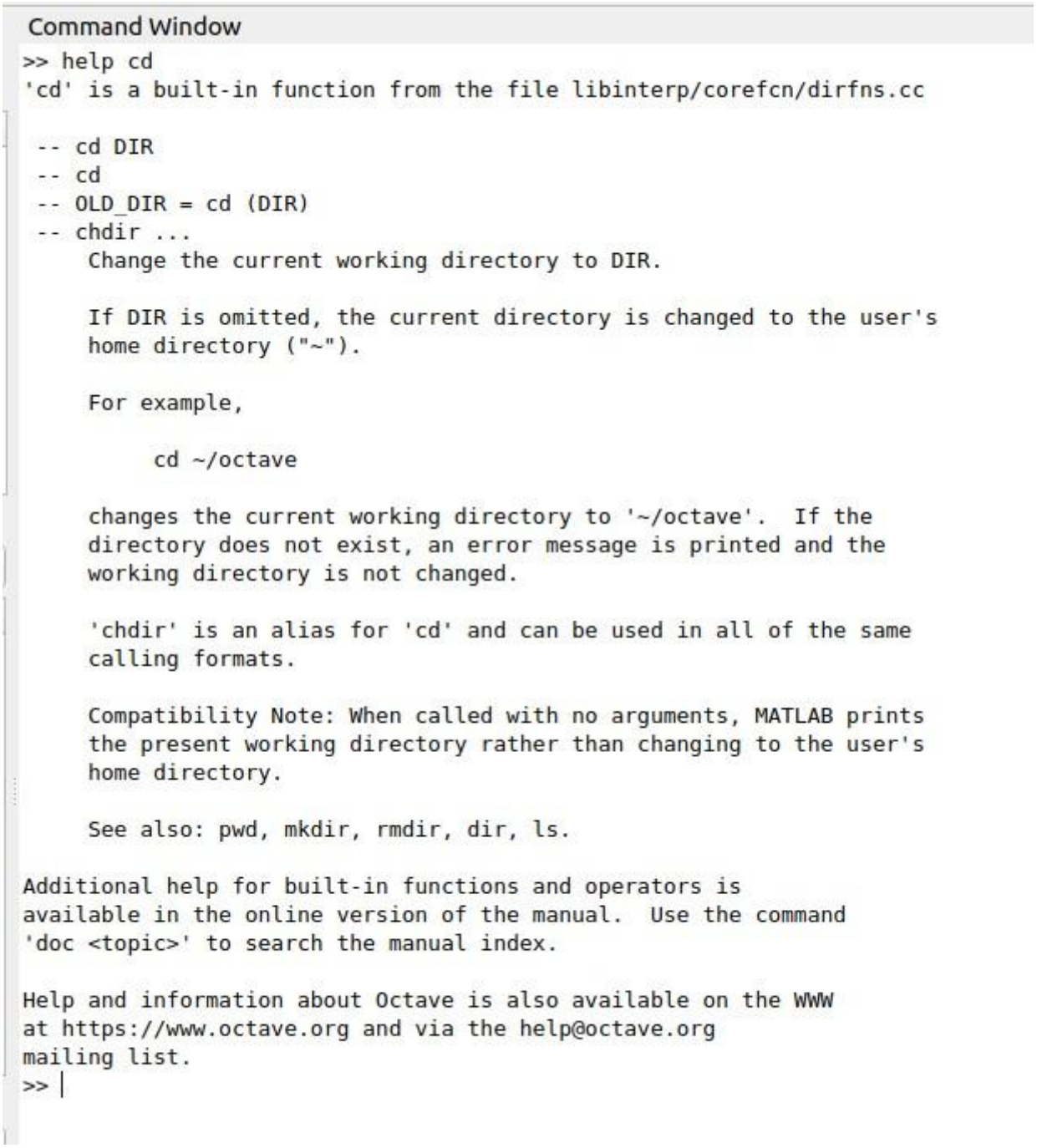

Figura 2: devolución del comando **help cd**.

B. Algunos comandos básicos

- **help**: Seguido por un comando da información sobre este.
- **doc**: Permite acceder al manual de Octave desde el promt.
- **whos**: Provee información sobre las variables definidas.
- **what**: Lista los archivos de Octave presentes en la carpeta actual.
- **close**: Cierra las ventanas gráficas.
- **clear**: Borra las variables de la memoria.
- **clc**: Limpia la ventana de comandos.
- **clf**: Limpia la ventana de gráficos.
- **Control + c**: Interrumpe el programa.

Todos estos comandos generarán efectos específicos con diferentes argumentos.

III. Tipos de Variables

En Octave es posible definir diferentes tipos de variables (y con el comando *whos* es posible visualizar qué tipo de variables están almacenadas en la memoria de trabajo).

Algunos ejemplos:

a. Variables numéricas: pueden ser reales o complejas.

   Por ejemplo:

   >> a = 25.4591
   >> b = 25.4591*i, donde i = √-1 es la unidad imaginaria

b. Variables string (caracter): Se asigna a la variable una serie de caracteres.

   Por ejemplo:

   >> a = ‘hola’

   Se asigna a la variable “a” los caracteres “hola”.

c. Variables simbólicas

   Para que Octave permita manipular variables simbólicas es necesario instalar el paquete “Symbolic”. En el curso no utilizaremos esta funcionalidad, por lo que no les exigiremos que tengan instalado este paquete.

Todos estos tipos de variables pueden definirse unitariamente o en forma de vectores o matrices del mismo tipo de variables. Existen otros formatos de variable que permiten mezclar variables de distinto tipo en una sola (por ej.: celdas). Se puede consultar el capítulo 6 del manual de Octave “Data Containers” por más información de este tipo.

IV. Funciones básicas

Octave tiene predefinidas un conjunto de funciones matemáticas básicas.

A continuación veremos algunos ejemplos de funciones:

A. Trigonométricas

Octave tiene la posibilidad de trabajar con las funciones trigonométricas en radianes o grados. Las funciones trigonométricas terminadas en “d“ indican su versión en grados. Por ejemplo:

>> a = sin(pi/3) = 0.8660
>> b = sind(60) = 0.8660
>> c = asin(1) = 1.5708
>> d = asind(1) = 90

B. Exponenciales

>> a = exp(5) = 148.4132
>> b = log(3) = 1.0986 Logaritmo natural (en base e)
>> c = log10(100) = 2 Logaritmo en base 10

C. Números Complejos

Consideremos el siguiente número complejo: y = 5 + 2.3i

>> a = real(y) = 5 (parte real)
>> b = imag(y) = 2.3 (parte imaginaria)
>> c = abs(y) = 5.5036 (valor absoluto)
>> d = conj(y) = 5.0000 - 2.3000i (conjugado)

D. Aproximaciones y Redondeo

x = [45.23 -23.79 0.3];

>>round(x) = [45 -24 0]; Redondea hacia el entero más próximo
>> fix(x) = [45 -23 0]; Redondea hacia cero
>> floor(x) = [45 -24 0]; Redondea hacia menos infinito
>> ceil(x) = [46 -23 1]; Redondea hacia más infinito

E. Polinomios

Un polinomio se puede definir mediante un vector de coeficientes.

Por ejemplo, el polinomio: $x^4 - 8x^2 + 6x - 10 = 0$ se puede representar mediante el vector [1, 0, -8, 6, -10].

Octave puede realizar diversas operaciones sobre él, como por ejemplo evaluarlo para un determinado valor de x (función polyval()) y calcular las raíces (función roots()):

```
>> pol=[1,0,-8,6,-10]

pol =
        1  0  -8  6  -10

>> ra = roots(pol)

ra =
        -3.2800
        2.6748
        0.3026 + 1.0238i
        0.3026 - 1.0238i
```

Aquí evaluamos el polinomio *pol*, con x = 1.

```
>> y = polyval(pol,1)

y =

        -11
```

V. Matrices y Vectores

A. Construcción de Matrices

Veamos ahora cómo construir vectores y matrices en Octave.

Esto es fundamental pues los datos deben ser presentados en forma de matrices para que el programa pueda procesarlos.

En el siguiente ejemplo definimos una matriz A de 2 filas y 3 columnas (es decir que tiene dimensiones 2×3), escribiendo los elementos que constituyen sus filas.

```
>> A = [3 4 5; 6 7 8]

A =

        3  4  5
        6  7  8
```

El comando size proporciona las dimensiones de la matriz (el primer número corresponde a la cantidad de filas y el segundo a la cantidad de columnas):

```
>> g = size(A)

g =
```

2 3

Si por ejemplo quisiéramos identificar el elemento ubicado en la fila 2 y la columna 1 de la matriz A, ejecutamos:

>> e21 = A(2,1)

e21 =

6

De esta forma podemos seleccionar un elemento dado de una matriz por la fila n y la columna m a la cual pertenece:

>> a = A(n,m);

De la misma manera, se puede seleccionar el n-ésimo elemento de un vector d como d(n).

Al seleccionar elementos, el parámetro end se puede usar para referirse al último elemento de un vector. Así d(end) selecciona el último elemento de d.

Cabe notar que entre los paréntesis pueden incluso referenciarse operaciones u otras variables.

B. Formas de Definir Vectores

Definamos un vector v que contenga los enteros 3, 4 y 5 en ese orden:

>> v = [3 4 5]

v =

3 4 5

Otra forma de definir el vector anterior sería de la siguiente forma:

>> v = [3:5];

En general se puede definir un vector cuyos elementos se obtienen a partir del anterior más un cierto incremento con la sintaxis. Por ejemplo, si queremos crear un vector que comienza en n_i, se incremente con d_n y termina en n_f :

>> v = [n_i : d_n: n_f]

En el caso que d_n = 1 alcanza con escribir

>> v = [n_i : n_f]

El número de elementos N del vector, se relaciona con n_i y n_f mediante la expresión:

$$N = \frac{n_f - n_i}{d_n} + 1$$

Si queremos crear un vector , que comience en n_i , termine en n_f , y tenga N elementos se debe usar el comando linspace y usar la siguiente sintaxis:

>> linspace(n_i , n_f, N)

o calcular el d_n correspondiente.

Un vector también puede definirse a partir de los elementos de una fila o columna de una matriz dada. En el siguiente ejemplo los elementos del vector v corresponden a la segunda fila de la matriz A, y los elementos del vector w corresponden a la primera columna de la misma matriz:

```
>> A = [3 4 5 ; 6 7 8 ; 1 2 3]

A =

     3 4 5
     6 7 8
     1 2 3

>> v = A(2 , :)

v =
     6
     7
     8

>> w = A(: , 1)

w =
     3
     6
     1
```

Ejecutando el comando length podemos obtener la longitud de un vector dado:

```
>> L = length(w)

L =

     3
```

C. Operaciones con matrices y vectores

Suma y resta

En este caso las matrices deben tener las mismas dimensiones. Por ejemplo, definamos dos matrices M1 y M2, y luego efectuemos la suma entre ellas; comprobaremos en el

siguiente ejemplo que cada elemento de la matriz suma es la suma de los elementos correspondientes de las matrices sumados (ídem para la resta):

```
>> M1 = [1 2 3; 4 5 6];

>> M2 = [0 1 0;-1 2 1];

>> s = M1 + M2

s =
        1  3  3
        3  7  7
```

Producto entre matrices

Sea P la matriz producto de las matrices A y B. En este caso el número de columnas de A debe ser igual al número de filas de B. Veamos el siguiente ejemplo:

```
>> A = [3 4 5; 6 7 8];

>> B = [3 2; 1 1; 0 -3];

>> p = A*B

p =
        13  -5
        25  -5
```

Todo lo anterior referido al producto entre matrices es válido igualmente para el cociente entre matrices (siempre que el determinante de la matriz divisor sea no nulo), y también para el producto y cociente entre vectores (por ser el vector un caso particular de matrices).
Potencia enésima de una matriz (sólo posible para matrices cuadradas):

```
>> A^n
```

Potencia enésima de una matriz elemento a elemento (en este caso la matriz NO tiene porque ser cuadrada): >> A.^n

Notar que en este caso hay que anteponer el punto (.) al símbolo de la potencia (^) para que realice la operación elemento a elemento.

Productos entre vectores

Sean los vectores x e y:

```
>> x= [1 2 3];
>> y= [2 -1 4];
```

Producto escalar: dot(x,y) el resultado es un escalar:

```
>> prodescal = dot(x,y)
```

prodescal = 12

Producto vectorial: cross(x,y) el resultado es vector:

>> prodvect = cross(x,y)

prodvect =

11 2 -5

VI. Lectura y almacenamiento de datos

A. Almacenamiento de Datos

Usualmente será necesario interrumpir el trabajo con Octave y poder continuarlo más tarde, recuperando la sesión en el mismo punto en el que se dejó (con las mismas variables definidas, con los mismos resultados intermedios, etc.). Hay que tener en cuenta que al salir del programa todo el contenido de la memoria se borra automáticamente.

Para guardar el estado de una sesión de trabajo existe el comando save. Si se teclea:

>> save <nombre del archivo>

se crea en el directorio actual un archivo de texto llamado <nombre del archivo> con el estado del workspace (esto no incluye, por ejemplo, los gráficos, que deben guardarse separadamente).

Dicho estado puede recuperarse la próxima vez que se arranque el programa con el comando:

>> load <nombre del archivo>

Hay que tener en cuenta que para cargar cualquier archivo, este debe encontrarse en nuestro Current Directory o debemos señalar la ruta hasta el directorio en el que se encuentra, por ejemplo:

>> load /home/Documentos/<nombre del archivo>

Este mismo tipo de sintaxis puede usarse también con el comando save.

El formato del archivo es, por defecto, un archivo de texto de datos de Octave. Este formato puede ser modificado por los argumentos del comando save. Veremos dos formatos alternativos para almacenar los datos.

Almacenamiento en Código ASCII

La sintaxis es: save -ascii <nombre del archivo> <variables>

Ejemplo:

```
>> a = [0:1:100];

>> save -ascii ejemplo1.dat a
```

Por defecto guarda solo la parte real de los números complejos, solo guarda matrices 2-D y utiliza el formato de precisión simple. Además, todas las variables deben tener la misma dimensión para ser almacenadas.

Esta forma de almacenamiento presenta la gran ventaja de que este archivo de datos puede ser leído por cualquier programa de manejo de texto y/o planillas. Por ejemplo: bloc de notas, Word, Excel, etc.

La terminación .dat no es obligatoria, pero se suele utilizar para identificar rápidamente el archivo como un archivo de datos en ascii.

Almacenamiento en Binario

La sintaxis es: save -binary <nombre del archivo> <variables>

Estos archivos no pueden ser leídos desde programas de procesamiento de texto, pero tienen algunas ventajas que se verán más adelante. En este caso las variables almacenar pueden tener diferente dimensión.Ejemplo:

```
>> a = [0:1:100];

>> b = sin(a);

>> save ejemplo2 a b
```

Creación de un Archivo desde el Bloc de Notas

Los resultados experimentales o de una simulación pueden ser procesados utilizando Octave.

Una opción para manejar estos datos desde el programa es generar desde el bloc de notas un archivo de datos. Este será un archivo ASCII. Para ello simplemente se debe abrir el block de notas e ingresar los datos en forma de columnas. Separar las columnas con un espacio es suficiente para que Octave las identifique como columnas distintas. Al almacenar tener la precaución de ponerle terminación .dat para identificarlos rápidamente como archivo de datos.

B. Lectura de Archivos de Datos

El comando para leer archivos de datos es load. Veremos a continuación como trabajar en los dos formatos vistos anteriormente: ASCII y BINARIO. En todos los casos, la sintaxis de lectura es: load <nombre del archivo> o alternativamente: load(‘nombre del archivo’)

En ambos casos, el comando puede ejecutarse directamente, o ser asignado a una variable:

```
>> a = load(“nombre del archivo‟)
```

Archivos ASCII

Cabe notar que al leer datos de un archivo ASCII, la variable que tenemos en el workspace de Octave tiene el mismo nombre que el archivo, pero sin la terminación. Por lo que suele ser de utilidad asignar un nuevo nombre a esta variable

>> b = load -ascii <nombre del archivo>

Archivos binarios

Los archivos binarios contienen un estado del workspace con las variables definidas y los valores dados, como fuera guardado en su momento. De esa manera, una vez leído el binario, las variables que tenemos en el workspace de Octave tiene el mismo nombre con que habían sido almacenadas, lo que representa una gran ventaja si se está realizando la lectura en el marco de un programa, puesto que se conoce a priori el nombre de las variables. Esto también sobreescribirá cualquier variable del mismo nombre que hubiera en el workspace antes de la lectura.

VII. Creación de M-files.

Octave permite ejecutar secuencias de comandos almacenados en un archivo. Estos archivos deben tener la extensión ".m", y por eso se denominan M-files. Existen básicamente dos tipos de M-files: los denominados script-files y function-files.

A. Script-files

Un script-file consiste de una sucesión de comandos Octave que se ejecutan sucesivamente cuando se teclea el nombre del fichero en la línea de comandos.

Por ejemplo, si el archivo tiene el nombre prueba.m, cuando se lo corre desde la pantalla de comandos (>> prueba), los comandos del archivo se ejecutan en el orden en que aparecen en el listado.
Las variables en el script-file son globales. Esto significa que si hay previamente en memoria alguna variable con el mismo nombre, su valor cambiará cuando se ejecute el programa. A su vez, todas las variables definidas dentro de un script quedarán en el workspace una vez terminado este.

El primer paso es abrir un editor de texto para escribir el programa, Octave incluye un editor para crear M - files, se encuentra una pestaña en la misma ventana que la command window, como se puede observar en la figura 3:

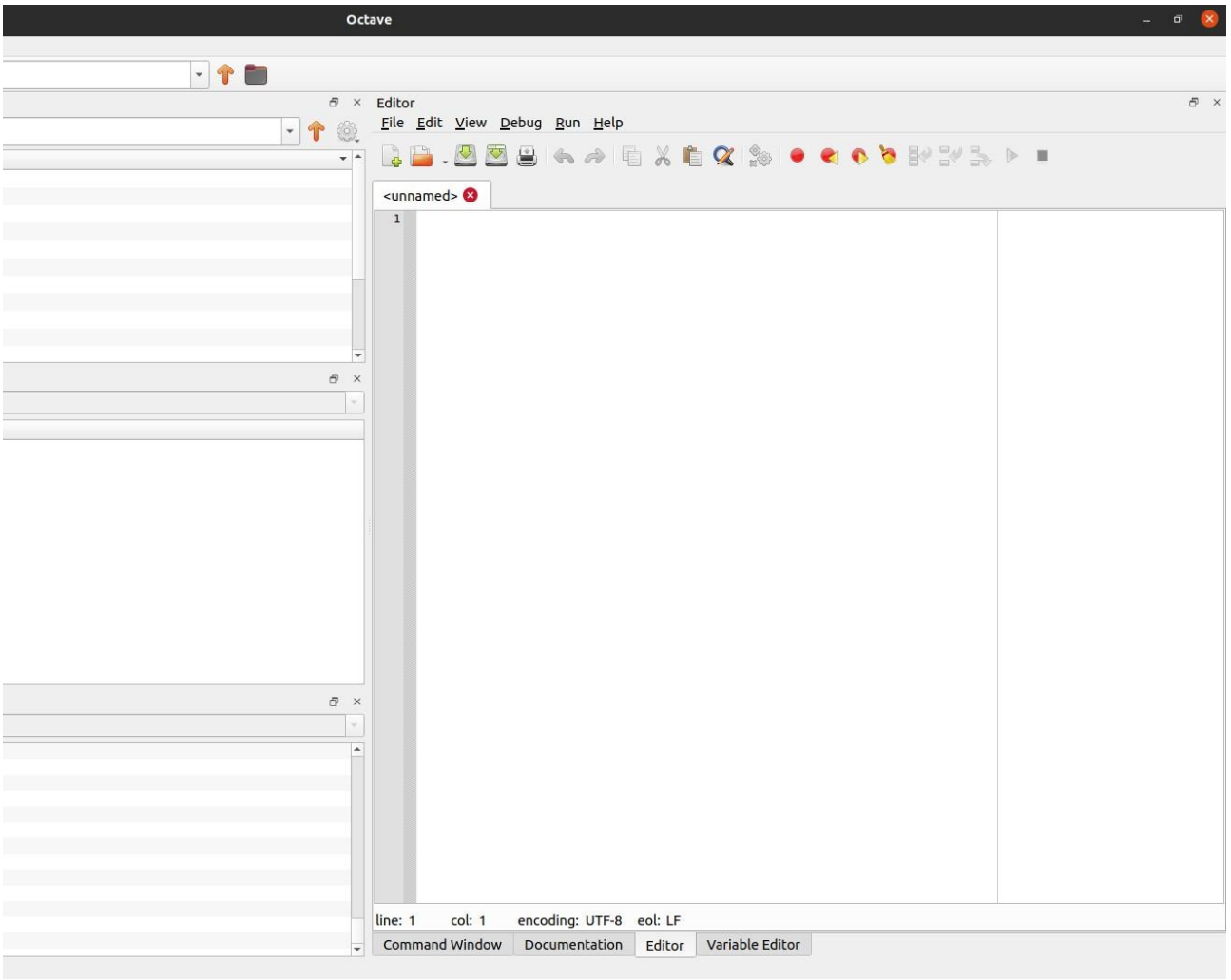

Figura 3: Editor desde el que se pueden crear M - files.

Una vez terminado de escribir el script, se debe salvar (haciendo click en save y luego haciendo click en save as... aparecerá otra ventana donde elegimos un nombre para el archivo del script (npar.m por ej.) y el directorio donde guardar el archivo.

Es importante tener en cuenta que los nombres de los scripts no pueden tener espacios (Por ejemplo, para guardar un script con el nombre programa 1, esto puede hacerse nombrando al mismo como programa1 o programa_1). Asimismo, recordar que los nombres de los programas en Octave deben tener extensión .m.

B. Function-files

Los function-files permiten extender la biblioteca de funciones Octave para el uso en aplicaciones específicas. A diferencia de los script-files, las variables en un function-file son locales, lo que significa que las funciones generan un workspace “virtual‟ con sus propias variables al ser llamadas. Por lo tanto, ninguna variable del mismo nombre que ya existiera en el workspace del usuario será modificada por la función y ninguna variable definida dentro de la función quedará en el workspace una vez que ésta termine.

En ambos casos, la excepción es la variable (o argumento) de salida de la función.

La primera línea de un function-file debe comenzar con la palabra function seguida de los argumentos de salida (si hay un solo argumento de salida esto puede ser omitido), una

expresión donde se declara el nombre de la función, que indica cómo la función debe ser llamada desde el espacio de trabajo de Octave, y los argumentos de entrada de la función. El nombre del function-file debe ser de la forma function-name.m.

En la figura 4 se muestra el ejemplo de crear una función que suma dos variables.

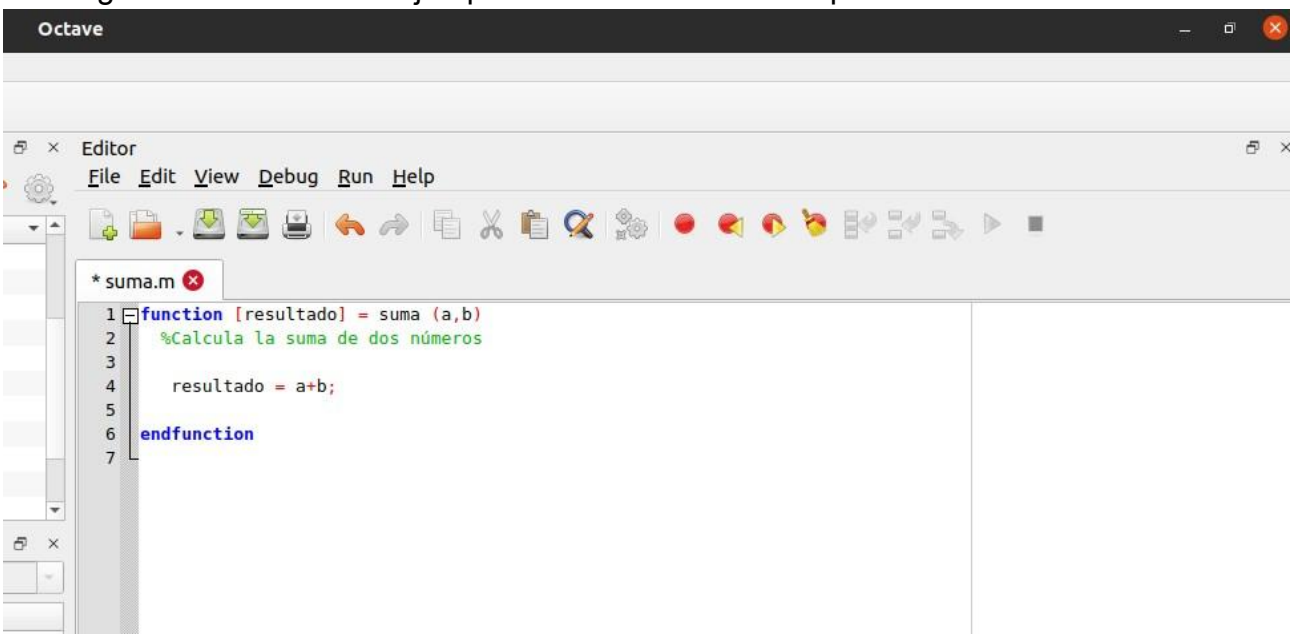

Figura 4: Función que suma dos variables.

En este caso, aún cuando hay un solo argumento de salida (“resultado”), elegimos ponerlo. “suma” es el nombre de la función y “a” y “b” los argumentos de entrada.

Abrimos un fichero M y lo guardamos en nuestra carpeta. Ahora se introduce:

```
>> s32 = suma(2,3)

s32 =

     5
```

Las líneas que comienzan con el carácter “%” corresponden a comentarios. Las líneas de comentarios que están inmediatamente debajo de la primera línea en un function file aparecerán como descripción cuando se llame a la función con el comando help.
Por lo tanto, es recomendable siempre incluir esta documentación en un function-file.

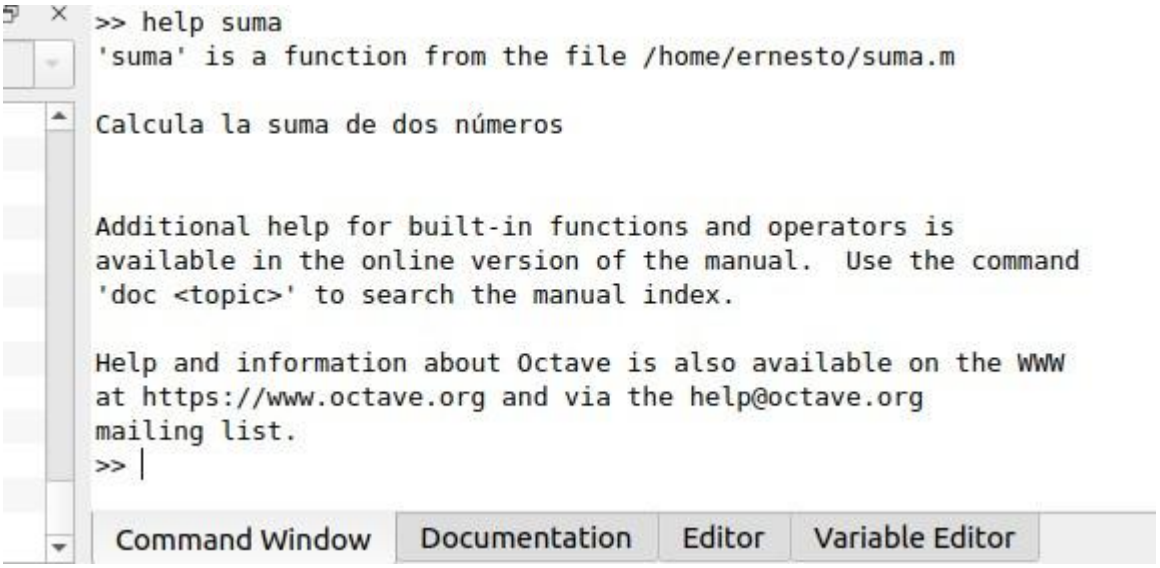

Figura 5: Ejemplo de la devolución de help para una función creada por nosotros.

Los comentarios pueden ser utilizados para explicar cualquier línea de un m-file.

VIII. Gráficas

Aquí veremos sólo algunos de los comandos básicos para la generación de gráficas en 2 dimensiones.

Lista de algunos comandos:

- plot: Crea el gráfico
- hold: Permite realizar la superposición de 2 o más gráficos en una misma pantalla. Es un comando on/off.
- figure: Genera una nueva pantalla de gráficos.

El comando más simple es plot(x, y), que utiliza dos vectores, x e y, de la misma longitud. Dibujará los puntos (x_i , y_i) y los unirá mediante rectas continuas.

Si no se le da ningún vector x, Octave asume que $x(i) = i$. A continuación plot(y) recibe el mismo espacio en el eje de las x: los puntos son (i , $y(i)$).

Se pueden cambiar el tipo y color de la línea que une los puntos mediante un tercer argumento. Si este argumento no existe, Octave dibuja por defecto una línea continua de color azul "-".

Introduciendo >> help plot se obtienen muchas opciones, aquí sólo indicamos unas pocas a modo de ejemplo:

>> plot(x, y, 'r+') dibuja los puntos en forma de +, y en rojo.

>> plot(x, y,' --') dibuja una línea discontinua.

>> plot(x, y,':') dibuja una línea de puntos.

Se pueden omitir las líneas y representar sólo los puntos discretos de distintas formas:

>> plot(x, y,'o') dibuja círculos.

Otras opciones son '+', 'x' o '*'.

Para obtener dos gráficas en los mismos ejes, utilizar

>> plot(x, y, X, Y)

Sustituyendo plot por semilogx, semilogy, o loglog se cambian uno o ambos ejes respectivamente a la escala logarítmica.

El comando

>> axis([a b c d]) especifica el valor de los ejes x e y según $a \leq x \leq b$, $c \leq y \leq d$.

Para dar título al gráfico o marcar los ejes de las x o de las y, se escribe entre comillas la etiqueta deseada, como en los ejemplos siguientes:

>> title ('altura Vs. tiempo')

>> xlabel ('tiempo en segundos')

>> ylabel ('altura en metros')

El comando hold conserva el gráfico anterior mientras se dibuja uno nuevo. Este comando alterna entre las opciones on y off. Para imprimir o guardar la pantalla de gráficos en un archivo, véase help print.

IX. Bifurcaciones y bucles

Aquí veremos sólo los primeros conceptos de programación. Para aquellos estudiantes que tengan interés, recomendamos profundizar en la bibliografía propuesta.

Octave, como cualquier otro lenguaje de programación, dispone de sentencias para realizar bifurcaciones y bucles. Las bifurcaciones permiten realizar una u otra operación según se cumpla o no una determinada condición. Los bucles permiten repetir las mismas o análogas operaciones sobre datos distintos un cierto número de veces.

Se utiliza la palabra end para indicar que finaliza el bucle.

Lista de operadores de comparación

== (igual) > (mayor) < (menor) >= (mayor o igual) <= (menor o igual)

~= (distinto) & (and) | (or)

A. Bifurcaciones: sentencia *if*

Para la bifurcación se utilizan las siguientes sintaxis:

```
if condición
        sentencia 1
```

```
        else
        sentencia 2
endif

sentencia 3
```

En la primera línea *if* indica que si (y sólo si) se cumple la condición dada, la segunda línea se va a realizar (sentencia 1). La tercera línea indica que si no se cumple la condición (*else*) se realiza la sentencia 2. Finalmente se cierra la bifurcación utilizando el comando *endif.*

En cualquiera de los casos, el programa sigue por la sentencia 3.

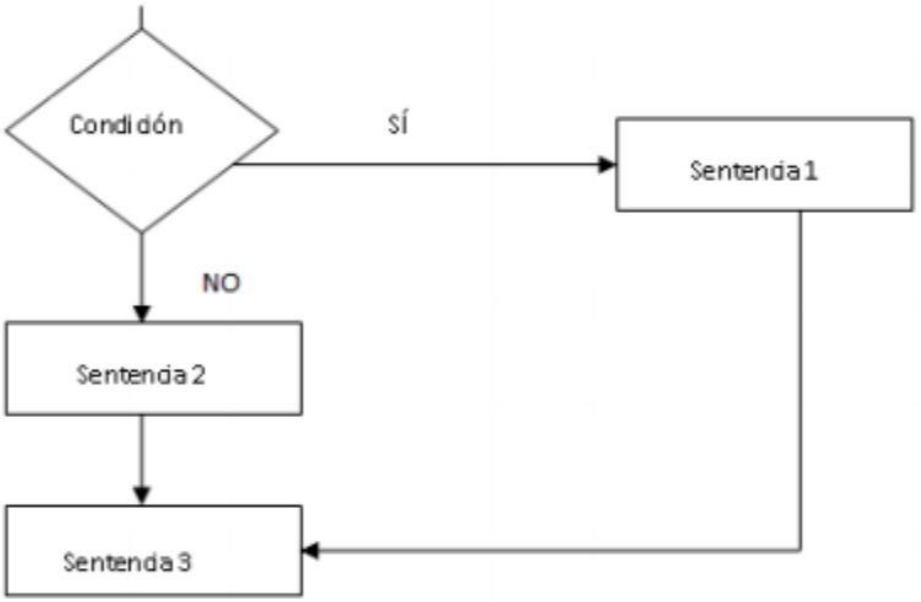

Figura 6: Diagrama de flujo para una bifurcación.

Vemos en la figura 7 un programa en el que se introducen dos números por el teclado y que nos dice cuál es el mayor:

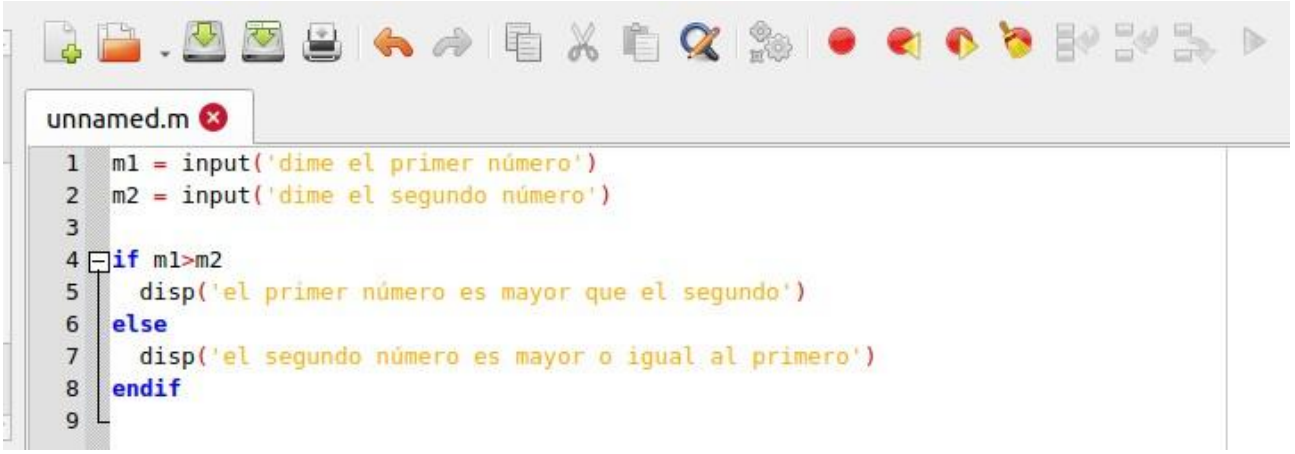

Figura 7:Programa para diferenciar el más grande de dos números.

Condiciones múltiples

Existe también la bifurcación múltiple, en la que pueden concatenarse tantas condiciones como se desee:

if condición1

bloque1

elseif condición2

bloque2

elseif condición3

bloque3

else (opción por defecto para cuando no se cumplan las condiciones 1,2,3)

bloque4

endif

Donde la opción por defecto *else* puede ser omitida: si no está presente no se hace nada en caso de que no se cumpla ninguna de las condiciones que se han chequeado. En todos los casos, las condiciones se chequean de arriba hacia abajo, teniendo prioridad las que fueron escritas primero.

En la figura 8 podemos observar el diagrama de flujo para condiciones múltiples.

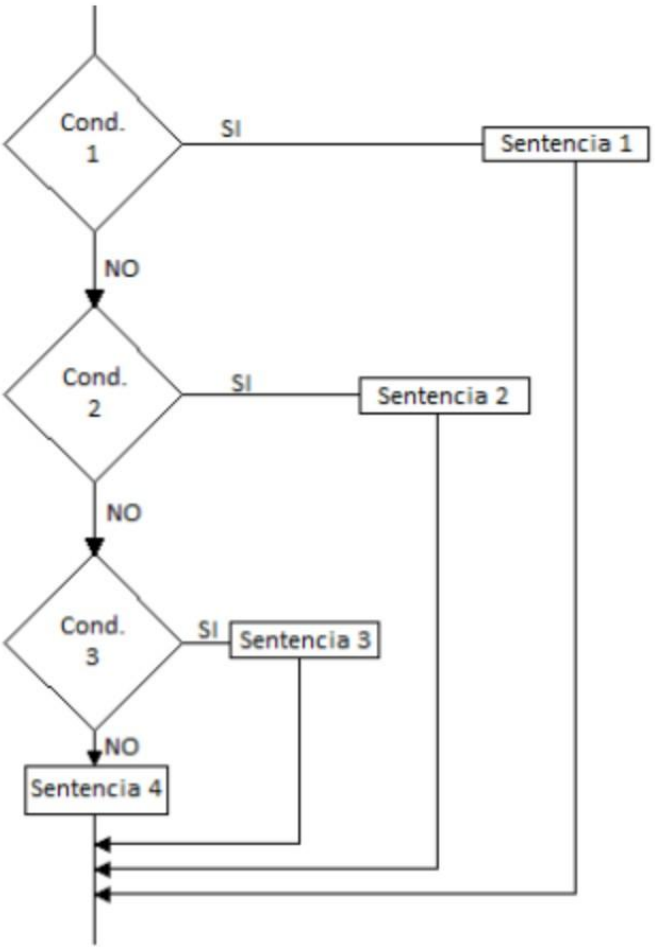

Figura 8: Diagrama de flujo de la bifurcación múltiple

B. Bucles: Sentencia *for*

Hay ocasiones en las que es necesario repetir el mismo conjunto de instrucciones muchas veces, tal vez cambiando algunos detalles.

La sentencia *for* repite un conjunto de sentencias un número predeterminado de veces. La siguiente construcción ejecuta sentencias con valores de i de 1 a n, variando el índice de uno en uno.

```
for i = 1 : 1 :n

        sentencias

endfor
```

Figura 9: Diagrama de flujo de un bucle *for*.

Las palabras claves son for y endfor. Este bucle pone en marcha una variable llamada contador (i) que va desde un valor de inicio (1, en este caso) hasta un valor final (n, en este caso) de paso en paso.

Cada vez que las sentencias se ejecutan, el contador aumenta en un valor paso (que si se omite, se le asigna automáticamente el valor 1). Cuando el contador llega al valor final, el bucle se acaba y el programa continúa con las sentencias que haya más allá de *endfor*.

En el siguiente ejemplo se presenta una estructura correspondiente a dos bucles anidados. La variable j es la que varía más rápidamente (por cada valor de i, j toma todos sus posibles valores):

```
for i = 1 : m
        for j = 1 : n
                sentencias
        endfor
endfor
```

C. Bucles: sentencia *while*

Una variante que combina la simplicidad de repetición del comando *for* con la flexibilidad de selección del *if* es el comando *while*. Éste funciona similar al bucle f*or*, a excepción de que en lugar de utilizar un contador que repita las sentencias dentro del bucle una cantidad dada de veces, las seguirá repitiendo siempre que se cumpla una condición dada.

Por ejemplo:

```
while a~=5
        sentencias
end
```

Repetirá el bloque “sentencias‟ mientras que la variable *a* no tome el valor 5. Al utilizar el bucle *while* debe tenerse precaución en no incluir una condición que nunca deje de cumplirse, a riesgo de caer en un “bucle infinito‟.

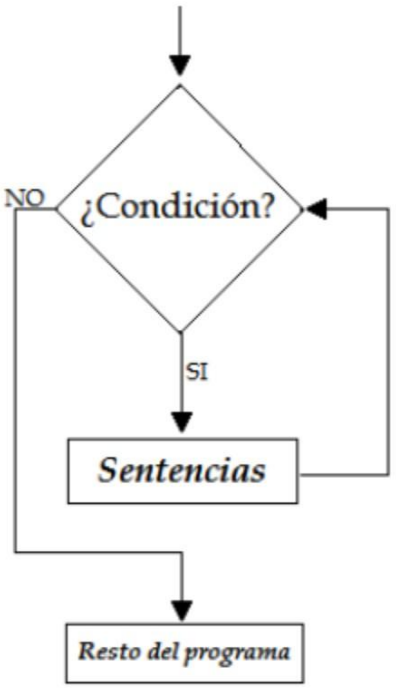

Figura 10: Diagrama de flujo de un bucle *while*.

X. Bibliografía y recursos

- https://www.gnu.org/software/octave/index
- Octave Manual: Free your Numbers - https://octave.org/octave.pdf
- https://wiki.octave.org/GNU_Octave_Wiki
- https://wiki.octave.org/Video_tutorials