Introducción a `matlab/octave`

Prácticas de Cálculo Numérico I (doble grado)

9 de febrero de 2021

1. Datos generales

Nombre de la asignatura: Prácticas de Cálculo Numérico I.
Profesor: Fernando Chamizo.
Horario: Martes 11:30–13:30.
Modalidad de docencia: En *streaming* via TEAMS. Se intentará grabar todas las clases, las cuales quedarán almacenadas en Microsoft Stream hasta la fecha de caducidad impuesta por esta plataforma.
Página web del curso: `http://matematicas.uam.es/~fernando.chamizo/` (seleccionando "Prácticas de Cálculo Numérico I"). Moodle se usará para las calificaciones.

2. Instalación y primer uso de `matlab/octave`

Entre el *software* empleado en cálculo numérico con fines académicos, sobre todo dentro del orientado a ingeniería, el que ha alcanzado una mayor difusión es `matlab` cuyo nombre abrevia *matrix* y *laboratory*. El nombre da una idea de que, al menos originariamente (ya tiene casi 40 años), estaba orientado a cálculos con matrices. Actualmente su ámbito es más amplio, especialmente cuando se instalan paquetes adicionales, sin embargo las matrices se siguen reflejando en la propia sintaxis y en mi experiencia (quizá sesgada) son los cálculos con ellas lo que mejor hace.

A pesar de que internamente esos cálculos se llevan a cabo con una adaptación de rutinas que hoy son de dominio público, `matlab` es *software* comercial propietario. Un punto a favor es que las licencias para estudiantes no son astronómicas y muchas universidades, como la UAM, tienen licencias generales. De todas formas, una alternativa libre y gratuita es (GNU) `octave`. Los comandos y programas básicos son intercambiables hasta el punto de que fuera de un ámbito relativamente avanzado, te parecerá que

`octave` es un emulador perfecto de `matlab` (salvo alguna cuestión fina en las representaciones gráficas). En particular, es de prever que todo lo que veremos en estas prácticas no establecerá ninguna diferencia. Más adelante habrá algunos comentarios dedicados `octave`, a pesar de que con seguridad la mayor parte de vosotros utilizaréis `matlab`. Un uso más avanzado muestra ventajas a favor de `matlab`, sobre todo con el uso de librerías, llamadas *toolboxes*, para temas especiales.

Para instalar `matlab` necesitamos una licencia. La UAM tiene una de campus disponible para todos los estudiantes. Las instrucciones para emplearla, por supuesto teniendo una dirección de correo institucional, están descritas siguiendo el enlace correspondiente en:

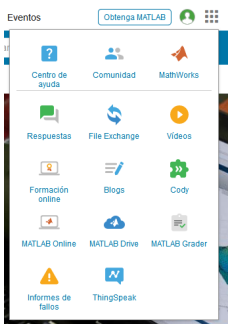
`https://faq.uam.es/index.php?action=show&cat=38`

Es bastante posible que muchos no deseéis ocupar disco duro en vuestro ordenador con programas grandes si hay alternativa. En este caso sí la hay, y es emplear `matlab` *online*. En principio es el formato que pienso utilizar en las clases porque reflejará la situación de la mayoría y además me permitirá no tener que transportar el ordenador si cambio de lugar en el campus.

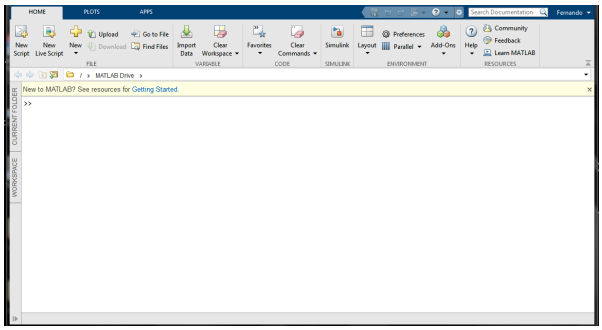
En realidad dispones de dos formas de usar `matlab` a través de la red. Una es con el servidor jupyter del departamento de matemáticas (`https://jupyter.mat.uam.es/hub/login`) pero esta no es la mejor idea porque requiere que sepas algo de jupyter y mi experiencia es que hay algunos problemas (sobre todo con los gráficos). Más conveniente es utilizar la web oficial de `matlab`. Una vez registrados, accederemos a través de

`https://matlab.mathworks.com/`

y tras entrar en nuestra cuenta, en el menú de la derecha, elegimos MATLAB Online que nos lleva al interfaz principal.



MATLAB Online



Aspecto inicial de la interfaz

Si deseamos instalar `octave`, la página del proyecto es:

`https://www.gnu.org/software/octave/`

Allí están los instaladores para diferentes sistemas operativos. La instalación básica es más ligera que la de de `matlab`, la última versión para Windows ocupa 1.8 G en el disco duro. Al parecer, `octave` también se puede usar *online* a través de `https://octave-online.net/`, pero no tengo experiencia al respecto.

Al ejecutar `octave` obtenemos una interfaz parecida a la de `matlab` pero más espartana. Insisto, no obstante, que es difícil notar diferencias moviéndose en un nivel básico.

3. Uso como una calculadora avanzada

El *prompt* está pidiendo que escribamos algo y ahí podemos teclear operaciones como en una calculadora. Ni que decir tiene que el producto se indica con `*` y, por otro lado, las potencias se indican con `^`, como en `sagemath` de la asignatura de Laboratorio. Por ejemplo, tecleando `1+1` y pulsando *enter*, obtendremos nuestra primera sesión

```
>> 1+1
ans = 2
```

Aquí `ans` es la abreviatura de *answer* (para decir toda la verdad, en realidad en la versión *online* la respuesta sale menos compacta, ocupando varias líneas). Por supuesto, existen todas las funciones que habitualmente pueblan las teclas de nuestras calculadoras y que incorporan los principales lenguajes de programación. Sin ánimo de ser exhaustivo, se tiene `sqrt` (raíz cuadrada), `exp`, `log` (logaritmo neperiano), las funciones trigonométricas `sin`, `cos`, `tan` y sus inversas `asin`, `acos`, `atan`. Además tenemos el número π denotado por `pi`. Por ejemplo:

```
>> ( log(pi) + exp(1) )*( (sin(1))^2 + sqrt(6) )
ans = 12.1977
```

Los espacios son irrelevantes, solo se han introducido para dar legibilidad al código.

Si te sientes defraudado por este ejemplo, ya que tu calculadora te da mayor precisión, es el momento que conozcas que hay varios formatos posibles y los dos más usados son `format short` y `format long`. Por defecto, estamos en el primero que da solo precisión simple. Si tecleamos `format long` y después

nuestro ejemplo anterior, el resultado pasa a ser `12.197703479831025` que posiblemente mejore a tu calculadora de bolsillo. Estos dos formatos admiten algunas variantes que no examinaremos aquí. Solo mencionaré `format rat` que aproxima por una fracción y en el caso anterior daría `5306/435`.

Una constante que no está en las calculadoras más básicas es `i`, la unidad imaginaria. Sin nada especial, podemos hacer operaciones con números imaginarios, incluso con funciones trascendentes. Por ejemplo, la famosísima fórmula de Euler $e^{i\pi} + 1 = 0$, es coherente con que `exp(i*pi)` resulte `-1.0000+0.000i` y con que `log(-1)`, que no nos dejaban hacer en secundaria, resulte `0.0000+3.1416i`. Por cierto, dependiendo de la versión de `matlab/octave` puede que la parte imaginaria de `exp(i*pi)` sea un número del orden de 10^{-16} en vez de ver `0.000`. De hecho, en la versión actual de `matlab` (*online*), el resultado antes mencionado contrasta con que `exp(pi*i) + 1` dé `0.0000e+00 + 1.2246e-16i` (recuerda que la notación científica *a*e*b* abrevia $a \cdot 10^b$). Seguro que estás sospechando, acertadamente, que esto se debe a que `matlab` (a diferencia de `sagemath`) no está haciendo cálculo simbólico. Hay cierto error interno con números que no son "exactos". La falta de precisión se cuantifica con el llamado *épsilon máquina*. Este es el menor error relativo que es posible distinguir. En `matlab` el épsilon máquina se obtiene con el comando `eps`. Si lo tecleamos, obtendremos `2.2204e-16`. Parece un número que no tiene nada de particular pero si calculas `2^-52` quizá cambies de opinión. La inmensa mayoría de los instrumentos digitales de cálculo que hayas utilizado en tu vida están afectados exactamente por este épsilon máquina. La razón es que el formato empleado habitualmente para almacenar números con 64 bits no permite un error relativo menor.

Si todo esto del error relativo más pequeño te resulta incomprensible, el siguiente ejemplo te iluminará:

```
>> (0.1+eps)-0.1
ans = 2.2204e-16

>> (10+eps)-10
ans = 0
```

La moraleja es que aunque nuestra calculadora u ordenador sea capaz de tratar con números tan pequeños como 10^{-100} y distinguirlos de cero, eso no significa que sea capaz de discriminar cualquier par de números que difieran en 10^{-100}. El límite real es del orden de 10^{-16} y este problema es bastante universal, no una manía de `matlab`.

Volviendo a la comparación con la calculadora, algo en lo que `matlab` va más allá es que está permitido el uso de variables como en cualquier lenguaje de programación típico. Además, al escribir la variable recuperamos su valor.

Por ejemplo, digamos que queremos saber cuántos minutos ha tenido 2020. Para ello creamos las variables `d` de días, `hd` de horas por día y `mh` de minutos por hora. La cuenta sería

```
>> d = 366
d =  366
>> hd = 24
hd =  24
>> mh = 60
mh =  60
>> d*hd*mh
ans =  527040
```

Es un poco rollo ver el mismo resultado que tecleamos. Una solución, que a la hora de programar será casi necesidad, es añadir al final de la línea un punto y coma, lo cual omite la salida. Es decir, teclearíamos `d = 366;` etc. excepto en la última línea de la que queremos ver el resultado. Digamos que ahora queremos hacer el cálculo para 2021, entonces `d` se reduce en uno porque no es bisiesto y podríamos proceder con

```
>> d = d - 1;
>> d*hd*mh
ans =  525600
```

4. Organizando los cálculos en ficheros

En cuanto queramos hacer algo más que un cálculo inmediato, es mejor utilizar un fichero en vez de teclear separadamente cada cálculo. Esto nos permitirá hacer modificaciones de manera más eficiente y se convertirá en una necesidad para programar.

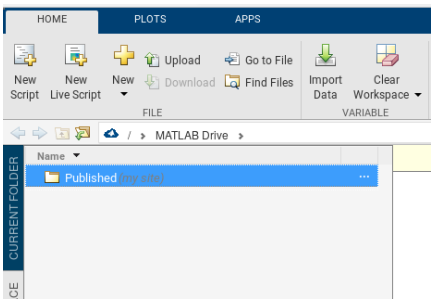
Los ficheros con instrucciones matlab tienen la extensión `.m` y esencialmente son de dos tipos: *scripts* y funciones. En realidad, la terminología varía según los autores. En un contexto informático general, los primeros son lo que llamaríamos programas y las segundas subrutinas a las que pueden llamar los programas.

Por ahora nos centramos en los *scripts*. Imaginemos que queremos crear uno llamado `ex01.m`. Distingamos dos situaciones, en la primera teniendo `matlab/octave` instalado y en la segunda con `matlab` *online*.

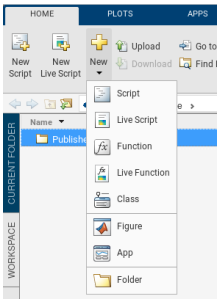
En el primer caso, las versiones modernas de `matlab/octave` llevan incorporado un editor pero si nos gusta más el nuestro (esto se aplica sobre todo a los maniáticos de Linux), nada impide que usemos el que queramos (que es lo que hago yo, sobre todo con `octave`). Así que basta crear con nuestro editor o con el editor incorporado, un fichero de texto `ex01.m` de la manera habitual (posiblemente File, New,...). Para que `matlab/octave`

tenga acceso al fichero debe estar en el `path` correspondiente. Existen los comandos de Linux `cd`, `ls`, `pwd` que nos permitirán saber dónde estamos y movernos donde esté el fichero. Soy consciente de que la explicación es un poco críptica pero al buen entendedor... y prefiero centrarme en el la segunda posibilidad, que posiblemente sea la mayoritaria.

Veamos ahora con todo detalle cómo crear un *script* en `matlab` *online*, que quedará almacenado en la nube. Para hacerlo un poco más completo vamos a meterlo dentro de un directorio llamado `week01`. Si desplegamos la pestaña `Current folder` veremos que `MATLAB Drive` solo contiene una carpeta llamada `Published`. Para crear `week01` pulsamos en `New` y en el desplegable escogemos `Folder`. Nos pedirá el nombre, `week01` en nuestro caso.

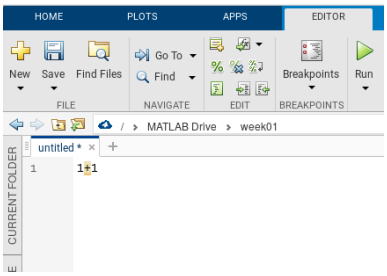


Pestaña Current folder

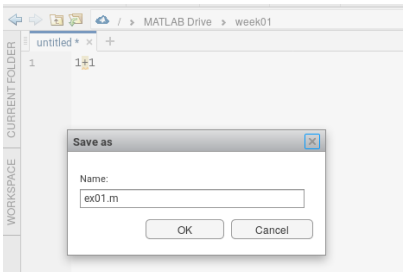


Desplegable de New

Con ello, en `Current folder` ya tenemos el nuevo directorio y nos metemos en él simplemente pinchando encima. Notemos que en la línea de encima, `>MATLAB Drive` se ha completado con un `>week01` a la derecha. Podemos usar esa línea con el ratón para navegar entre los directorios. O bien pulsando de nuevo `new`, pero ahora escogiendo `script`, o bien haciendo lo mismo pulsando el botón derecho del ratón dentro de la pestaña `Current folder`, creamos un *script*. De la primera forma podremos ajustar directamente el nombre, de la segunda, cuando demos a `Save` nos pedirá un nuevo nombre (por defecto es `untitled.m`.



Nuevo script



Grabar el nuevo script

En las imágenes, `ex01.m` tiene un contenido bastante anodino, solo `1+1`. Si lo ejecutamos pulsando el botón `Run` (el triángulo verde) o escribiendo en la ventana de comandos `ex01` (sin el `.m`), obtendremos la consabida respuesta `ans = 2`.

Iremos viendo a lo largo del curso muchos comandos. Por ahora, juguemos a hacer nuestro uso tipo calculadora pero con más comodidad. Los comentarios se escriben precediendo su contenido con `%`.

```
1  % Aproximando el número e
2  n = 50;
3  (1+1/n)^n
4  % Con un n doble del anterior
5  n = 2*n;
6  (1+1/n)^n
7  % sin(1) y su aproximación de Taylor de orden 7
8  sin(1)
9  1-1/factorial(3)+1/factorial(5)-1/factorial(7)
10 % Diferencia de sin(1) y Taylor de orden 7
11 sin(1) - (1-1/factorial(3)+1/factorial(5)-1/factorial(7) )
```

Los puntos y comas 2 y 11 impiden que se vean los valores 50 y 100 de `n`. Por cierto, indico los números de línea para hacer referencia a ellos y facilitar la copia, por supuesto no hay que teclearlos.

5. Matrices, la gran fortaleza de `matlab`

Cada lenguaje de programación tiene sus peculiaridades: en C los punteros, en C++ y Java las clases y en python las listas. Como indica su nombre, en `matlab` la estructura de datos estrella son las matrices. En muchas ocasiones, sustituirán a los bucles que utilizaríamos en los lenguajes mencionados. La manera de indicar una matriz es limitándola con corchetes e indicando los cambios de fila con punto y coma. Los elementos de una fila se pueden separar con comas pero esto no es obligatorio, con espacios es suficiente. La suma y el producto de matrices no requieren comandos, bastan `+` y `*`. La traspuesta se indica con el apóstrofo. Para ser del todo rigurosos, lo que se consigue con él es la traspuesta conjugada pero para matrices reales no hay diferencia. La matriz nula `m` $\times$ `n` se indica con `zeros(m,n)` y la matriz identidad de dimensión `n` mediante `eye(n)`. También existe `ones(m,n)` que general una matriz `m` $\times$ `n` con todos sus elementos iguales a uno.

Como ejemplo, escribamos en un *script* algunas operaciones con las matrices

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 2 & -1 & 1 \end{pmatrix} \qquad \text{y} \qquad B = \begin{pmatrix} -2 & 1 & 0 \\ 3 & 1 & 1 \end{pmatrix}$$

que también involucren matrices especiales.

```
% Definiendo dos matrices
A = [1, 0, 1; 2, -1, 1];
B = [-2, 1, 0; 3, 1, 1];
% Calcula la suma de matrices
A + B
% La traspuesta de B es
B'
% El producto de A por la traspuesta de B es
A*B'
% Esto da A
A + zeros(2,3)
% y esto vuelve a dar A*B'
A*B'*eye(2)
```

Un comentario para los que uséis `octave` en vez de `matlab` *online*, es que cuando la salida ocupa más de una "página" debemos pasar la salida línea a línea. Para evitar esto, basta teclear en la ventana de comandos `more off`. Si queremos restablecer este comportamiento, usaremos `more on`.

El elemento en la fila `m` y columna `n` de una matriz `A` se indica con `A(m,n)`. Así, en el ejemplo anterior `A(2,1)` resultaría 2 y `B(1,1)` sería -2. A diferencia de lo que ocurre en `sagemath` y en los principales lenguajes de programación, los índices no empiezan en cero sino en uno.

Las matrices admiten ser construidas por bloques conservando la misma sintaxis que si los bloques fueran números. Por ejemplo, en física la matriz de Dirac γ_2 es una matriz que se construye a partir de la matriz de Pauli σ_2 por medio de

$$\gamma_2 = \begin{pmatrix} O & -\sigma_2 \\ \sigma_2 & O \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & -i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix} \qquad \text{con} \qquad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

Esta matriz es unitaria, es decir, al multiplicarla por su traspuesta conjugada da la identidad. Escribamos su construcción y la comprobación de que es unitaria en un *script*.

```
1 % Matriz de Pauli
2 s_2 = [0 -i; i 0]
3 % Matriz de Dirac
4 g_2 = [zeros(2) -s_2; s_2 zeros(2) ]
5 % Comprobamos que es unitaria
6 g_2*g_2'
```

La construcción por bloques se produce en la línea 4 y de paso aprendemos que `zero(n)` es una abreviatura de `zero(n,n)` (lo mismo funciona para `ones`). La matriz γ_2 es simétrica, por tanto la traspuesta conjugada es lo mismo que la conjugada. La conjugación se indica mediante `conj`, entonces sustituyendo la línea 6 por `g_2*conj(g_2)`, seguimos obteniendo la identidad.

Los vectores (fila o columna) no son otra cosa que matrices que tienen una de sus dimensiones iguales a uno. Sin embargo hay dos constructores de vectores fila que merecen una consideración especial.

Generalizando ligeramente lo que ocurre en python, con `[`a`:`b`]`, o simplemente con a`:`b, obtenemos un vector fila cuyos elementos son a, $a+1$, etc. hasta el último valor que no supera a b. De esta forma $2:4$ es el vector `[2, 3, 4]` y $1{,}7:4$ es `[2.7, 3.7]`. Si necesitamos un incremento diferente de la unidad, basta indicarlo entre los dos puntos. Así podemos generar cualquier vector cuyos elementos estén en progresión aritmética. Por ejemplo:

```
[2:3:15]      →   [2, 5, 8, 11, 14]
```

donde los corchetes son superfluos.

Necesitaremos, a veces, definir estos vectores con elementos en progresión aritmética especificando el origen, el final y el número de elementos. Por supuesto lo podemos conseguir con los dos puntos haciendo un pequeño cálculo en sucio, pero es más directo emplear el comando `linspace(`a`,`b`,`n`)` donde n es el número de elementos. Por ejemplo, `linspace(2,5,8)` resultaría

```
2.0000 2.4286 2.8571 3.2857 3.7143 4.1429 4.5714 5.0000
```

(así, sin corchetes ni comas es como aparece en la salida). El incremento es de $3/7$ y por tanto equivale a `2:3/7:5`.

Por cierto, la longitud de un vector se obtiene con el comando `length` y las dimensiones de una matriz con `size`.

La notación de los dos puntos sirve también para establecer rangos de índices de vectores o matrices. Veámoslo con un ejemplo.

```
% Una matriz y un vector fila
A = [1, 0, 1; 2, -1, 1];
v = [2, 3, 5, 7];
% La matriz formada por las dos últimas columnas
A(1:2,2:3)
% Una alternativa
A(:,2:3)
% La sección central de v
v(2:3)
% Muestra las dimensiones de A y v
size(A)
size(v)
length(v)
% Anulamos las dos últimas columnas de A
A(:,2:3) = zeros(2)
```

De hecho la última instrucción se podría escribir como `A(:,2:3) = 0` porque cuando no hay ambigüedad, los números se identifican con matrices de las dimensiones adecuadas. Por ejemplo, `12+A` es válido y `12` se identifica con la matriz 2×3 que tiene todas sus elementos iguales a 12.

Veamos otro ejemplo para practicar. Se sabe que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \qquad \text{para} \quad n \geq 1,$$

donde F_n son los números de Fibonacci. Utilizando esta fórmula, vamos a hacer un programilla, poco eficiente, que dado n nos diga cuánto vale F_n. Utilizaremos los comandos de entrada/salida `input`, que pide un dato al usuario y `disp` que muestra el valor de una variable sin que aparezca `ans =`.

```
% Pedimos la n
n = input('Introduce n');
% Elevamos a n
F = [1,1;1,0]^n;
% El resultado es el elemento 1,2 (o el 2,1)
disp(F(1,2))
```

Como cabe esperar, elevar a un n natural una matriz en `matlab` es la abreviatura de multiplicarla por sí misma n veces. Esto funciona incluso con números enteros si la matriz es invertible.

Las funciones trascendentes se aplican elemento a elemento a una matriz. De esta forma `exp(A)` no tiene nada que ver con el e^A que te explica(rá)n en los cursos de ecuaciones diferenciales sino que es algo tan prosaico como calcular e elevado a cada elemento. De esta forma

```
exp([0,1]) → 1.0000 2.7183,     sin(ones(2)) → 0.84147 0.84147
                                               0.84147 0.84147
```

donde, recuérdese, que `ones(2)` abrevia `ones(2,2)`.

A veces nos gustaría tener un producto de matrices, o potencias naturales, término a término. En principio parece un antojo raro pero en breve veremos que es fundamental, por ejemplo a la hora de representar gráficas. La manera de indicar que `*` o `^` se deben hacer término a término es precediéndolos de un punto. Por ejemplo, `[1:10].^2` es un vector fila con los 10 primeros cuadrados y `[1,2].*[6,3]` resulta `[6,6]`. Con la multiplicación usual `*` no tiene sentido multiplicar dos matrices 2×3 pero sí lo tiene empleando `.*` para obtener otra matriz de las mismas dimensiones. Análogamente, `./` define una división término a término sin problemas siempre que los denominadores no se anulen.

Referencias

[Gil06] A. Gilat. *Matlab: una introducción con ejemplos prácticos.* Reverté, 2006.

[Gri15] D. F. Griffiths. An Introduction to Matlab. `http://www.maths.dundee.ac.uk/ftp/na-reports/MatlabNotes.pdf`, 2015.

[HVZ12] E. Hernández, M. J. Vázquez, and M. A. Zurro. *Álgebra lineal y geometría*. Pearson, Madrid, 2012. 3a ed.

[Pra02] R. Pratap. *Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers*. Oxford University Press, 2002.

[QS07] A. Quarteroni and F. Saleri. *Cálculo Científico con MATLAB y Octave*. Springer, Milan, 2007.

[QSG14] A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific computing with MATLAB and Octave*, volume 2 of *Texts in Computational Science and Engineering*. Springer, Heidelberg, 2014. Fourth edition [of MR2253397].