CAPÍTULO 1

Introducción a las funciones y los archivos `.m`

En este curso vamos a utilizar con frecuencia la manera habitual de "programar" en Matlab/Octave, que es mediante *archivos* `.m` y funciones definidas en ellos.

Un archivo `.m` no es más que un archivo ordinario cuyo nombre termina en `.m` y que contiene una lista de instrucciones de Matlab/Octave con una estructura precisa. El código del listado 1.1 se puede incluir en un archivo `.m` que simplemente realizará una secuencia de instrucciones:

```
% Un archivo elemental
e = exp(1);
b = linspace(-2,2,1000);
plot(b, e.\^b)
```

Listado 1.1. Un archivo `.m` simple

La potencia de los archivos `.m` se debe a dos propiedades:

- Si se graban en un directorio directamente accesible a Matlab/Octave (por ejemplo, `Mis Documentos/Matlab`), se puede ejecutar un comando cuyo nombre sea el mismo que el de un fichero de ese directorio (sin la extensión `.m`) y Matlab/Octave hará las instrucciones pertinentes. Por ejemplo, si se ha grabado el archivo `ejemplo1.m` del código de 1.1, entonces puede ejecutarse la línea

  ```
  > trial1
  ```

  que dibujará la función e^x para $x \in [-2, 2]$ usando 1000 puntos.
- Se pueden usar para definir funciones complicadas.

En lugar de una mera secuencia de comandos, se pueden definir un programa ("una función", suele decirse) dentro de un archivo `.m`. Considérese el listado 1.2, que define una función que devuelve los dos máximos valores de una lista, en orden creciente.

La estructura es la siguiente:

(1) Varias líneas de comentarios (las que comienzan con %). Se utilizan para describir el programa definido después.

(2) Una línea como

```
function [y1, y2,...] = nombreFun(p1, p2,...)
```

donde la palabra `function` aparece justo al principio, después una lista de nombres (entre *corchetes*, `[]`), que designan las *variables de salida*, un signo igual =, después el *nombre de la función*, en este caso `nombreFun`, y una lista de nombres, que son los *parámetros de entrada*, entre paréntesis.

(3) Luego viene una sucesión de líneas de comandos de Matlab, que implementan justamente lo que hace la función (esto sería el programa en sí).

(4) La palabra clave `end` al final.

(5) Por último, es esencial que el nombre del archivo sea el mismo que el de la función junto con la extensión `.m`. Por ejemplo, para el listado 1.2, el nombre del archivo *debe ser* `max2.m`.

Un archivo que siga todas estas reglas definirá un comando de Matlab/Octave nuevo que se puede utilizar como cualquier otro.

```
% max2(x)
% devuelve los 2 valores máximos de x, en orden creciente
% si solo hay un valor (length(x) == 1), se devuelve [-inf, x].

function [y] = max2(x)

  % inicializar: el primer elemento es siempre -inf
  y = [-inf, x(1)];

  % si solo hay un número, ya hemos terminado
  if(length(x) == 1)
    return;
  end

  % para cada elemento, solo hay que hacer algo si es mayor que y(1)
  for X=x(2:end)
    if(X > y(1))
      if(X > y(2))
        y(1) = y(2);
        y(2) = X;
      else
        y(1) = X;
      end
    end
  end
```

```
end
```

Listado 1.2. Un primer fichero .m que define una función. Grábese como max2.

Por ejemplo, si se guarda el código del listado 1.2 en un archivo de nombre max2.m dentro del directorio Mis Documentos/MATLAB, entonces se puede ejecutar la siguiente cadena de comandos:

```
> x = [-1 2 3 4 -6 9 -8 11]
x =

   -1    2    3    4   -6    9   -8   11

> max2(x)
ans =

    9  11
```

que muestra cómo se ha definido una nueva función llamada max2 que, cuando se corre, ejecuta las instrucciones del archivo max2.m.

Como se van a utilizar archivos .m y funciones definidas en ellos con bastante frecuencia, se sugiere al estudiante que haga tantos ejemplos como le sea posible. La programación en Matlab/Octave no es para nada más compleja que la de Python (y en muchos casos, es más sencilla, por el carácter vectorial del lenguaje).

Nota 1. Los principales elementos de programación que se requerirán están incluidos en la hoja de consulta rápida y son los siguientes:

- El enunciado if...else. Tiene la sintaxis siguiente:

  ```
  if CONDICIÓN
  ... % sucesión de comandos si se cumple CONDICIÓN
  else
  ... % sucesión de comandos si NO se cumple CONDICIÓN
  end
  ```

  Hay más posibilidades (con elseif) pero no vamos a entrar en demasiados detalles.
- El bucle while. Sintaxis:

  ```
  while CONDICIÓN
  ... % sucesión de comandos si se cumple CONDICIÓN
  end
  ```

  que ejecutará los comandos internos mientras (while) se cumpla la condición.
- El bucle for. Sintaxis:

```
for var=LIST
... % sucesión de comandos
end
```

asignará en la variable `var` secuencialmente cada elemento de la lista `LIST` y ejecutará los comandos internos del bucle hasta que se termine la lista.

- Expresiones lógicas. Las `CONDICIÓNes` (de `if` y `while`) pueden ser expresiones simples (como `x<3`) o complejas (construidas utilizando los operadores `and`, `or` y `not`, que se escriben como `&&`, `||` y ` `).

Ejercicio 1: Impleméntese una función `min3` que, dada una lista de números como entrada, devuelva los *tres* elementos mínimos. Úsese la función `max2` definida arriba como guía. —

Ejercicio 2: Impleméntese una función `es_creciente` que, dada una lista como entrada, devuelve un `1` si la lista está en orden no decreciente y un `0` si no. ¿Cómo lo harías? —

Ejercicio 3: Mejórese la función del Ejercicio 2 para que devuelva dos valores: primero, el mismo que la función `es_creciente`, y como segundo valor, la longitud de la "secuencia creciente" más larga que haya al principio de los valores de entrada. Llámese a la función `es_creciente2`. Por ejemplo, podría usarse así:

```
> [a,b] = es_creciente2(7, -1, 2, 3 4)
  a = 0
  b = 1
> [u,v] = es_creciente2(-1, 2, 5, 8, 9)
  u = 1
  v = 5
> [a,b] = es_creciente2(3, 4, 5, -6, 8, 10)
  a = 0
  b = 3
```

—

Ejercicio 4: Defínase una función llamada `positiva` que, dada como entrada una lista de números, devuelve dos valores: el número de elementos positivos de la lista (estrictamente mayores que 0) y la lista de dichos elementos como segundo valor de salida. Ejemplos de uso:

```
> [a,b] = positiva(-2, 3, 4, -5, 6, 7, 0)
  a = 4
  b = [3 4 6 7]
```

```
> [a,b] = positiva(-1, -2, -3, -exp(1), -pi)
  a = 0
  b = []
> [a,b] = positiva(4, 2, 1, 3 2)
  a = 5
  b = [4 2 1 3 2]
```

¿Usarías un bucle `while` o un `for`? ¿Por qué?

Obsérvese que para las funciones que devuelven varios valores de salida, si se quiere que devuelvan más de uno, hay que pedirlo explícitamente asignando una lista de variables, como en el Ejercicio 4. Si solo se quiere un valor (el primero de la lista de salida), se llama con normalidad:

```
> positiva(1, 2, -2, 0, 4)
  3
> x = positiva(-2, 1, 3, 7, 2)
  x = 4
```

pero si se quiere que devuelva los dos, hay que hacer una asignación con corchetes:

```
> [n x] = positiva(pi, -2, 3, -5, 0)
  n = 2
  x = [pi 3]
```

Si la función se llama sin asignar el valor de salida, solo devuelve el primero. Este valor es habitualmente el más importante, pero hay ocasiones en que uno requiere acceder a más parámetros de salida.

Ejercicio 5: Defínase una función `secante` que reciba como entrada una función `f`, un número real `x0` y un número real positivo `epsilon`. Debe devolver dos números `a` y `b`: la pendiente `a` y la altura `b` del corte con el eje OY de la recta secante a la gráfica de `f` que pasa por $f(x_0 - \epsilon)$ y $f(x_0 + \epsilon)$.

Ejercicio 6: Defínase una función `mav` que transforme una matriz `A` de cualquier tamaño en un vector `v`, de la siguiente manera: si `A` tiene n filas y m columnas, el vector `v` tendrá $n \times m$ componentes: las primeras m componentes serán la primera fila A, las siguientes m la segunda fila, etc…como en el ejemplo.

$$A = \begin{pmatrix} 1 & 0 & -3 \\ 2 & 4 & 7 \\ 3 & 2 & 1 \\ -2 & 0 & 6 \end{pmatrix} \longrightarrow v = (1, 0, -3, 2, 4, 7, 3, 2, 1, -2, 0, 6).$$

Ejercicio 7: Defínase una función `vam` que realice la operación inversa de `mav` (ver ejercicio 6): dado un vector `v` y dos dimensiones `n` y `m` (número de filas y número de columnas), se ha de devolver una matriz de tamamño $n \times m$ cuya primera fila sean los primeros m elementos del vector, cuya segunda fila sean los m siguientes, etc. Como en el ejemplo:

$$v = (1,2,0,4,5,7,-2,-3,1,0,2,-6), n = 3, m = 4$$
$$\longrightarrow A = \begin{pmatrix} 1 & 2 & 0 & 4 \\ 5 & 7 & -2 & -3 \\ 1 & 0 & 2 & -6 \end{pmatrix}.$$

Si $n \times m$ es mayor que la longitud de v, se ha de devolver una matriz vacía (como erro). Si es menor, los elementos que sobren han de ovlidarse.

Ejercicio 8: Defínase una función `elementosM` que reciba como entrada una matriz `A` de cualquier tamaño y una matriz `P` de dos columnas (y cualquier número de filas). La función debe devolver los elementos de `A` que corresponden a las filas de `P` como si fueran coordenadas, en un vector. Por ejemplo:

$$A = \begin{pmatrix} 1 & 2 & 0 & 4 \\ 5 & 7 & -2 & -3 \\ 1 & 0 & 2 & -6 \end{pmatrix}, P = \begin{pmatrix} 2 & 3 \\ 1 & 4 \\ 1 & 1 \\ 3 & 2 \end{pmatrix} \longrightarrow (-2,4,1,0).$$

Si `P` no tiene dos columnas o alguna de sus filas es una coordenada que no existe en `A`, se debe devolver un vector vacío (como un error).

Ejercicio 9: Defínase una función `darvalores` que reciba como entrada una matriz `A`, una matriz de dos columnas `P` y un vector `v` con tantas columnas como filas de `P`. La función debe devolver la matriz `A` en la que cada posición correspondiente a las filas de `P` tenga el valor

que corresponde de v, como en el ejemplo.

$$A = \begin{pmatrix} 1 & 2 & 0 & 4 \\ 5 & 7 & -2 & -3 \\ 1 & 0 & 2 & -6 \end{pmatrix}, P = \begin{pmatrix} 2 & 3 \\ 1 & 4 \\ 1 & 1 \\ 3 & 2 \end{pmatrix}, v = (3, 9, 2, 1)$$

$$\longrightarrow A = \begin{pmatrix} 2 & 2 & 0 & 9 \\ 5 & 7 & 3 & -3 \\ 1 & 1 & 2 & -6 \end{pmatrix}$$

En caso de que haya algún lugar inaccesible o de que v tenga un número distinto de elementos que las filas de P, se ha de devolver (a modo de error) una matriz vacía.

CAPÍTULO 2

Soluciones aproximadas de ecuaciones no lineales

Este capítulo estudia los métodos elementales para resolver ecuaciones no lineales de una variable:

$$f(x) = 0 \tag{1}$$

donde f es una función "razonable".

1. Bisección

El teorema de Bolzano dice que una función continua que cambia de signo en un intervalo cerrado tiene automáticamente un cero en dicho intervalo. Este resultado (que es uno de los más importantes del Cálculo) sirve para diseñar un procedimiento que aproxime una raíz dividiendo el intervalo en segmentos cada vez más pequeños. En concreto: dado un intervalo $[a, b]$ y una función $f(x)$ continua sobre él, si $f(a)f(b) < 0$ entonces se sabe que hay una raíz de f en (a, b). Tómese $c = (b + a)/2$ (el punto medio). Si $f(a)f(c) < 0$ entonces hay una raíz en $[a, c]$. Si no, tiene que haberla en $[c, b]$. En cualquiera de los dos casos, ahora se tiene un intervalo de longitud la mitad y puede repetirse el argumento. Esto se formaliza en el Algoritmo 1.

Ejercicio 10: Impleméntese el algoritmo de bisección en un fichero `.m` llamado `biseccion.m`. —

Ejercicio 11: Utilícese el algoritmo de bisección para calcular raíces aproximadas de las funciones que se indican en los intervalos correspondientes. Téngase en cuenta que el algoritmo usado "sin pensar" puede no funcionar para algunos casos, incluso aunque las funciones tengan raíces en dichos intervalos.

$$\begin{aligned}
f(x) &= \cos(e^x), x \in [0, 2] \\
g(x) &= x^3 - 2, x \in [0, 3] \\
h(x) &= e^x - 2\cos(x), x \in [0, \pi] \\
r(t) &= t^2 - 1, x \in [-2, 2] \\
s(z) &= \sin(z) + \cos(z), z \in [0, 2\pi] \\
u(t) &= t^2 - 3, t \in [-2, 2] \\
f(t) &= \operatorname{atan}(t - 2), t \in [-3, 3]
\end{aligned}$$

Algoritmo 1 Algoritmo de bisección.

Input: una función $f(x)$, un par de números reales a, b con $f(a)f(b) < 0$, una tolerancia $\epsilon > 0$ y un límite de iteraciones $N > 0$.
Output: bien un mensaje de error o un número real c entre a y b tal que $|f(c)| < \epsilon$ (es decir, una raíz aproximada).

⋆Condición previa
if $f(a) \notin \mathbb{R}$ **ó** $f(b) \notin \mathbb{R}$ **then**
  return ERROR
end if
⋆Inicio
$i \leftarrow 0$
$c \leftarrow \dfrac{a+b}{2}$
while $|f(c)| \geq \epsilon$ **y** $i \leq N$ **do**
  if $f(a)f(c) < 0$ **then**
    $b \leftarrow c$ [`intervalo` $[a, c]$]
  else
    $a \leftarrow c$ [`intervalo` $[c, b]$]
  end if
  $i \leftarrow i + 1$
  $c \leftarrow \dfrac{a+b}{2}$ [`punto medio`]
end while
if $i > N$ **then**
  return ERROR
end if
return c

Explíquese cómo puede usarse para las funciones que tienen raíces pero para las cuales el algoritmo "sin pensar" no da resultado. —

2. El método de Newton-Raphson

El algoritmo de Newton-Raphson para encontrar raíces de una función $f(x)$ requiere que esta sea al menos derivable (y, por tanto, continua). Se puede describir formalmente como en el Algoritmo 2.

Ejemplo 1 Para implementar el método de Newton-Raphson *sin utilizar operaciones simbólicas* en Matlab, hace falta programar una función cuya entrada incluya la semilla `x0`, la función `f` (como una función anónima), su derivada `fp` (pues no se puede utilizar derivación

Algoritmo 2 Algoritmo de Newton-Raphson.

Input: una función diferenciable $f(x)$, una *semilla* $x_0 \in \mathbb{R}$, una tolerancia $\epsilon > 0$ y un límite para el número de iteraciones $N > 0$.
Output: Bien un error bien un número real c tal que $|f(c)| < \epsilon$ (es decir, una raíz aproximada).

⋆Inicio
  $i \leftarrow 0$
  while $|f(x_i)| \geq \epsilon$ **y** $i \leq N$ **do**
    $x_{i+1} \leftarrow x_i - \dfrac{f(x_i)}{f'(x_i)}$ [puede dar un NaN ó ∞]
    $i \leftarrow i + 1$
  end while
  if $i > N$ **then**
    return ERROR
  end if
  $c \leftarrow x_i$
  return c

simbólica) y tanto la tolerancia `epsilon` como el límite para el número de iteraciones `N`. El listado 2.1 muestra una tal implementación.

```
% Newton-Raphson implementation.
% Returns both the approximate root and the number of
% iterations performed.
% Input:
% f, fp (derivative), x0, epsilon, N
function [z n] = newtonraphson(f, fp, x0, epsilon = eps, N = 50)
  n = 0;
  xn = x0;
  % initialize z to a NaN (i.e. error by default)
  z = NaN;
  % Both f and fp are anonymous functions
  fn = f(xn);
  while(abs(fn) >= epsilon & n <= N)
    n = n + 1;
    fn = f(xn); % memorize to prevent recomputing
    % next iteration
    xn = xn - fn/fp(xn); % an exception might take place here
  end
  z = xn;
  if(n == N)
    warning('Tolerance not reached.');
  end
```

```
end
```

Listado 2.1. Una implementación (simple) del método de Newton-Raphson.

Obsérvese que, puesto que la función se llama `newtonraphson`, el archivo debe llamarse `newtonraphson.m`.

Ejercicio 12: Utilícese la función `newtonraphson` definida en el Ejemplo 1 para calcular raíces aproximadas de las funciones del Ejercicio 11. Utilícense diferentes valores para la semilla, el épsilon y el límite de iteraciones.

¿Converge siempre el algoritmo?

Ejercicio 13: Modifíquese el código de `newtonraphson` (escribiendo una nueva función llamada `newtonraphson_plot`, con su correspondiente fichero) de manera que, en cada paso, se marque el punto $(x_i, 0)$ en una gráfica, con símbolo suficientemente grande. Úsese con las funciones del Ejercicio 11.

Ejercicio 14: **Hágase este ejercicio sólo si se tiene tiempo suficiente.** Como en el Ejercicio 13, créese una función `newtonraphson_graph` que dibuje, no solo los puntos $(x_i, 0)$ sino también la recta tangente a $f(x)$ desde $(x_{i-1}, f(x_{i-1}))$ hasta $(x_i, 0)$, para visualizar el proceso. Utilícese el comando `pause` entre cada dibujo para esperar a que el usuario presione una tecla (así, el proceso puede verse paso a paso). Úsese esta función como en el Ejercicio 11.

3. El método de la secante

El método de la secante consiste en modificar del de Newton-Raphson que se utiliza cuando la derivada de f es, bien desconocida, bien computablemente pesada de calcular. En lugar de utilizarse $f'(x_i)$ en cada paso, este algoritmo aproxima dicho valor como

$$f'(x_i) \simeq \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}},$$

lo cual requiere llevar cuenta no solo de x_i sino también de x_{i-1} en cada paso.

Ejercicio 15: Escríbase una función `secant` en un archivo llamado `secant.m` que implemente el método de la secante. Téngase en cuenta que, a diferencia del método de Newton-Raphson, esta requiere como entrada solo una función anónima `f` pero *dos* semillas, `x0` y `x1`.

Utilícese esta función `secant` para calcular raíces aproximadas de las funciones del Ejercicio 11. —

4. El método del punto fijo

El método del punto fijo se basa en el siguiente resultado:

Theorem 1. *Sea $g : [a, b] \to [a, b]$ una función de $[a, b]$ en sí mismo, continua y derivable en $[a, b]$. Si hay un número real positivo $\lambda < 1$ tal que para cada $x \in [a, b]$, $|g'(x)| \leq \lambda$, entonces existe un y solo un $\alpha \in [a, b]$ para el que $g(\alpha) = \alpha$.*

El resultado requiere verificar una buena colección de condiciones antes de que pueda ser utilizado para resolver una ecuación (de hecho, lo primero que hace falta habitualmente es transformar esa ecuación en una de punto fijo). Esencialmente, si uno comienza con la ecuación

$$f(x) = 0,$$

antes de utilizar el método del punto fijo, uno tiene que transformarla en un "problema de punto fijo", es decir, en una ecuación de la forma

$$\tilde{f}(x) = x$$

lo que significa que uno busca un valor c que queda fijo por $\tilde{f}$ (es decir, $\tilde{f}(c) = c$, de ahí el nombre). La transformación habitual es sencilla:

$$f(x) = 0 \Leftrightarrow f(x) + x = x.$$

Obviamente, $f(c) = 0$ si y solo si $f(c) + c = c$. Así, por lo general $\tilde{f}(x) = f(x) + c$.

En lugar de intentar verificar todas las condiciones del teorema 1, uno habitualmente trata de encontrar una solución de $f(x) + x = x$ directamente y, *a posteriori*, verificar que se ha encontrado

Ejercicio 16: Escríbase una función `fixed_point` que implemente el método del punto fijo. La entrada debería ser: una función anónima `f`, una semilla `x0`, una tolerancia `epsilon` y un límite para el número de iteraciones `N`. La salida debería ser bien un mensaje de error o bien un número `c` tal que `abs(f(c))<epsilon`.

Nota: La iteración para este problema no es $x_{i+1} = f(x_i)$.

Utilícese esta función `fixed_point` para calcular (si se puede) raíces aproximadas de las funciones del Ejercicio 11. ¿Funciona siempre? ¿Cuándo lo hace? ¿Por qué? —

APÉNDICE A

Estructura de un programa en Matlab

Estructura básica de un programa de Matlab que implementa un algoritmo.

Los parámetros son:

- Entrada: `p1`, `p2`, `p3`, …
- Salida: `s1`, `s2`, …

P.ej: el algoritmo de bisección requiere 5 parámetros de entrada:

f: La función cuya raíz se busca.
a: Extremo izquierdo del intervalo.
b: Extremo derecho del intervalo.
N: Número máximo de iteraciones.
e: Tolerancia (precisión deseada).

y los datos de salida pueden ser dos:

r: La raíz aproximada.
k: El número de iteraciones realizado.

Un programa que implementara una función sería como en el listado A.1 y *debe ser grabado en un archivo que se llame como la función, en el ejemplo* `NombreF.m`.

```
function [s1, s2, ..., sn] = NombreF(p1, p2, ...., pn)
    s1 = valor razonable por defecto
    s2 = valor razonable por defecto
    % ...
    % y así con todos los valores de salida

    % si hace falta un contador, se genera aquí
    k = 1

    % es posible que haga falta calcular algunas cosas antes
    % del bucle principal

    % AHORA VENDRÁ UN BUCLE while() o for().
    % Por lo general sera un while salvo que se sepa
    % exactamente el numero de veces que se va a hacer algo
    % la última condicion siempre será (si hay contador)
    %       que k <= número maximo de iteraciones,
    %       REPITO: si hubiera contador
```

```
    while( CONDICION1 && CONDICION2 ... && CONDICIONn )

         % aquí se escribe el "cuerpo del algoritmo",
         % "lo que es el propio algoritmo en sí"
         % (bisección, Newton-Raphson...)
         % aquí habrá ifs, cálculos, asignaciones, todo
         % tipo de cosas

    end

    % Antes de terminar, hay que verificar que la respuesta
    % es correcta. Por ejemplo, si hay un contador, hay que
    % verificar si se ha pasado el límite. Si esto ha ocurrido,
    % es IMPRESCINDIBILE avisar de que el resultado que se
    % devuelve es incorrecto. Por ejemplo:
    if( k > N) % asumiendo el límite es N
        warning('Numero maximo de iteraciones superado')
    end

    % fin del programa: no olvidar el 'end' de aquí abajo

end
```

Listado A.1. Estructura básica de un programa elemental de Matlab.