Análisis económico con MatLab (II)

Gonzalo Fernández de Córdoba Martos

Primero de diciembre de dos mil

1 Introducción

La primera parte del curso de MatLab tenía como objetivo estudiar las operaciones básicas de MatLab y en el esfuerzo hicimos algunos ejercicios de análisis económico que utilizaban los comandos esenciales de MatLab. Esta segunda parte va a estar más dirigida al análisis económico, aunque como veremos tendremos que hacer una pequeña inversión en MatLab. Vamos a ver fundamentalmente los métodos numéricos para resolver sistemas de ecuaciones no lineales, pero antes de empezar es esencial que sepamos cómo construír nuestras propias funciones en MatLab ya que su uso va a ser muy necesario de ahora en adelante.

2 Construcción de funciones

Una función en MatLab es un programa con extensión **.m** que sirve para ejecutar una rutina en la que dadas unas variables que entran como argumento, se obtiene el valor de otras variables como retorno. Por ejemplo cuando tenemos definido un vector $x = (x_1, x_2, \cdots, x_n)$ y queremos calcular la suma de las componentes podemos hacer dos cosas. Una es escribir en nuestro propio programa unas lineas que digan

```
suma=0;
for i=1:n
    suma=suma+x(i);
end
```

En esta pequeña parte de nuestro programa se irían sumando los elementos del vector x en una variable que se inicializa en 0 y que va acumulando las componentes del vector x. La otra cosa que podemos hacer es utilizar el comando **sum** y hacer

```
suma=sum(x);
```

Como se puede ver, esta segunda forma de calcular la suma es mucho más compacta y elegante y para usarla basta con saber que la función **sum.m** existe y saber qué hace. Existe en la estructura de directorios de MatLab un archivo llamado **sum.m** que hace este trabajo. Ese archivo con extensión **.m** es una función de MatLab. MatLab dispone de un archivo de inicio llamado **matlabrc.m** en el subdirectorio principal de MatLab. En este archivo se especifica la ruta de acceso de MatLab a cada una de las funciones que vienen incorporadas en el paquete. Cuando MatLab está ejecutando un programa

tienen prioridad de acceso los archivos situados en vuestro directorio de trabajo, si allí no encuentra la función o una variable escrita en el programa que se ejecuta, entonces acude a la ruta de acceso de funciones especificado en el archivo **matlabrc.m**. Por esta razón es importante que las funciones que vamos a construír estén accesibles con toda rapidez a MatLab situándolas en el mismo subdirectorio en el que estamos trabajando.

Existen dos tipos de archivo en MatLab que tienen extesión **.m,** unos son los que hemos visto hasta ahora, que son llamados archivos de tipo *script* y que contienen colecciones de comandos ejecutables por MatLab, y los otros son los que vamos a ver ahora, son los archivos del tipo *function.* Se llaman archivos de tipo *function* porque la primera palabra que MatLab tiene que encontrar cuando los ejecuta, es precisamente el comando **function**. Para ver cómo funcionan vamos a escribir un ejemplo y luego voy a comentar con detalle los elementos del ejemplo. Queremos construír una función que cuando la apliquemos sobre un vector nos dé la media de los elementos del vector. El código de MatLab que haría esto por nosotros es:

```
function y=media(x)
%Este programa calcula la media de los elementos del vector x.
n=length(x);
y=sum(x)/n;
```

Esta nueva función construida por nosostros se llamará **media.m** (nótese la correspondencia entre el nombre del archivo y la línea de comando **function y=media(x)**, que si bien no es necesaria es conveniente que así sea) y la conservaremos en el directorio en el que estamos trabajando con la ventana de comandos de MatLab, de modo que si estamos trabajando en el directorio c:\docs\user\alberto\xfiles el programa **media.m** lo guardaremos en ese directorio para que pueda ser usado desde ese espacio de trabajo. Note que los comentarios al programa están situados debajo de la línea **function y=media(x)**, ya que los archivos del tipo *function* deben comenzar, como ya he dicho, con el comando **function**. La línea que tiene definida a la operación principal de nuestro programa es **y=sum(x)/n;** y observe su correspondencia con la línea de comando **function y=media(x)**. Teniendo en cuenta estas reglas el programa será un éxito. Para comprobarlo basta con irnos a la ventana de comandos y allí crear un vector **a** que sea $a = (1, 2)$.

```
»a=[1 2];
```

y una vez que lo tengamos creado, como es el caso, llamar a nuestra función **media.m** para calcular la media de **a**.

```
»media(a)
```

```
ans =
1.5000
»
```

Como vemos ha funcionado correctamente y con esto hemos construido nuestra propia función en MatLab. Ahora fíjense en algo de extrema importancia que acaba de suceder. Si tecleamos sobre la ventana de comandos el ya conocido comando **whos** nos encontramos con algo interesantísimo:

```
» whos
  Name      Size          Elements    Bytes    Density    Complex
  a         1 by 2        2           16       Full       No
  ans       1 by 1        1           8        Full       No
Grand total is 3 elements using 24 bytes
»
```

¿Cuál es la sorpresa?. Las variables **y**, **x** y **n**, del archivo al que hemos llamado **media.m** no están en el listado de variables y si no están es porque no existen. Las variables que aparecen dentro del archivo del tipo *function* **media.m** son todas variables locales y sólo existen temporalmente en el periodo de ejecución pero nunca antes y nunca después. Para comprobarlo vamos a hacer una pequeña versión de **media.m** totalmente inútil pero muy ilustrativa del hecho.

```
function y=media(x)
%Este programa calcula la media de los elementos del vector x.
n=length(x);
y=sum(x)/n;
whos
```

Ahora al ejecutar el programa nos vamos a encontrar con lo siguiente:

```
» media(a)
  Name      Size          Elements    Bytes    Density    Complex
  n         1 by 1        1           8        Full       No
  x         1 by 2        2           16       Full       No
  y         1 by 1        1           8        Full       No
Grand total is 4 elements using 32 bytes
ans =
1.5000
»
```

El comando whos ha actuado en nuestra función reconociendo SÓLO las variables que están definidas en la función pero ninguna otra, ni siquiera la variable **a**. Es importante entender esto. Imaginen que en nuestro programa

del tipo *function* hubiera una variable llamada **a** (igual que el vector del que queríamos calcular la media pero con otro valor u otros valores distintos) e imaginen que MatLab, ejecutando **media.m**, se acordara de que **a** existe fuera de la función, entonces habría una colisión y MatLab no podría distinguir cuál de las dos **a**es tiene que usar. Para evitar este problema MatLab separa las variables GLOBALES que están definidas en el archivo del tipo *script* de las variables LOCALES definidas en los archivos de tipo *function* y una vez que sale de la ejecución del programa de tipo *function* olvida las variables que allí fueron definidas. Si no lo hiciera, esto sería un desastre.

Una forma de hacer "visible" una variable que fue definida en un archivo de tipo *script* en una función es escribir el comando **global var1 var2 ... varn**, donde var1 es el nombre de la primera variable que queremos hacer "visible" por todas las funciones. En este caso las variables "pasan" de los archivos del tipo *script* a los archivos del tipo *function*. Si bien esto es una posibilidad no está recomendado usarla, y la razón es la ya comentada: es posible que se produzcan colisiones entre variables de fuera de la función con variables de la función.

Ejercicio: Hacer una función que calcule varianza de un vector cualquiera. Las dos funciones que hemos construido hasta ahora son funciones que aceptan un argumento (el vector del que queríamos calcular la media o la varianza) y devuelven un número (la media o la varianza). Ahora queremos hacer una función que tome un argumento (que será un vector) y que nos devuelva la media y la varianza, es decir, dos números. Un programa que funciona es este:

```
function [y, z]=varme(x)
%Este programa calcula la media y la varianza de
%los elementos del vector x. Uso: [media,varianza]=varme(z)
%donde z es un vector cualquiera
n = length(x);
y = sum(x)/n;
z = sum((x-y).^2)/n;
```

Como podemos ver la diferencia con respecto al los dos programa que hemos hecho es que la primera línea de comandos **function [y, z]=varme(x)** recoge explícitamente el hecho de que queremos dos resultados, el **y** y el **z**. Para hacer funcionar este programa debemos también ser explícitos en el momento de pedir los resultados escribiendo sobre la línea de comandos

```
[c, d]=varme(a)
c =
```

1.5000
d =
0.2500
»

Como las variables del archivo de tipo *function* son variables locales podemos hacer uso de la función **varme** llamándola con cualquier vector **[nombre_1, nombre_2]=varme(nombre_3);** y, por tanto, no es necesario que el vector de llamada **[c, d]** coincida en nombre con el archivo **[y, z]**.

Ejercicio: Hacer un programa llamado **stad.m** que al introducir un vector nos devuelva el valor máximo, el valor mínimo, la media y la desviación típica. Ya hemos construído funciones que introducen un argumento y devuelven un número, funciones que introducen un argumento y devuelven uno o varios números. Ahora queremos hacer una función que introduce varios argumentos y devuelve varios números. Para ver cómo funciona voy a mostrarles un ejemplo. La función que voy a construír toma como argumentos los parámetros (a, b) de una función de demanda lineal $p = a - bQ_d$ y (c, d) de una función de oferta lineal $p = c + dQ_s$ además de un impuesto. La función nos devuelve la cantidad en transacción bajo el impuesto $Q(t)$, el precio pagado por los consumidores y el precio percibido por los productores. El programa llamado **taxes.m** podría ser así:

```
function [Qt, pc, pv]=taxes(a, b, c, d, t)
%Esta función devuelve las cantidades y los precios bajo un
%régimen fiscal dado por t.
Qt = (a-(c+t))/(b+d);
pc = (a*d+b*(c+t))/(b+d);
pv = ((a-t)*d+b*c)/(b+d);
```

Como podemos ver este programa tiene 5 argumentos y 3 variables de retorno. Debes pensar en esta función como en una función $taxes : \Re_+^5 \to \Re_+^3$ que sería una función de variable real que toma valores de $\Re_+^5$ y devuelve valores reales en $\Re_+^3$. Por esta analogía este tipo de archivos se llaman archivos del tipo *function*. Para hacerlo funcionar bastaría con escribir en la ventana de comandos algo cómo:

```
=taxes(10, 0.1, 2, 0.5, 0)
Qt =
13.3333
pc =
8.6667
pv =
```

8.6667
»

O quizá, y aunque esto sea reiterativo, esto otro:

=taxes(10, 0.1, 2, 0.5, 0)
a =
13.3333
b =
8.6667
c =
8.6667
»

Con este último ejemplo quiero poner de manifiesto una vez más la independencia entre los nombres de lo que escribís en la ventana de comandos y el funcionamiento de la función **taxes.m** resaltando el caracter local de las variables de dentro de la función y el caracter global de las variables de fuera de la función (y dentro del *script*).

Es posible que deseemos en algún momento hacer una función que tome valores de $\Re^{20}$ y devuelva valores de $\Re^{520}$, para lo cual tener que escribir sobre la ventana de comandos algo así como
$[x_1, x_2, \cdots, x_{520}] = nombre\_de\_la\_función(y_1, y_2, \cdots, y_{20})$ resultaría o muy pesado o imposible. Para resolver este pequeño problema podemos compactar los argumentos de la función y las variables de retorno usando matrices o vectores. Esto es decir que la naturaleza de los argumentos y de las variables de retorno no está restringida a que sean números; pueden ser también matrices de cualquier dimensión. Veámoslo con un ejemplo en el que además de los precios se nos devuelven los excedentes de todos y cada uno de los agentes introduciendo un vector de parámetros y el impuesto.

```
function res=taxes1(param)
%Esta función devuelve las cantidades y los precios bajo un
%régimen fiscal dado por t. El vector param contiene el intercepto
%de la demanda, y su pendiente, el intercepto de la oferta y su
%pendiente y finalmente el impuesto
a = param(1);
b = param(2);
c = param(3);
d = param(4);
t = param(5);
Qstar = (a-c)/(b+d);
```

```
pstar = (a*d+b*c)/(b+d);
Qt = (a-(c+t))/(b+d);
pc = (a*d+b*(c+t))/(b+d);
pv = ((a-t)*d+b*c)/(b+d);
EE = t*Qt;
EC = (a-pc)*Qt/2;
EP = (pv-c)*Qt/2;
PIE = t*(Qstar-Qt)/2;
ET = EE+EC+EP;
res = [Qstar pstar Qt pc pv EE EC EP PIE ET];
```

Esta función toma un único argumento (que es **param**)y devuelve un único elemento (que es **res**). Su funcionamiento sería igual que el de cualquiera de las funciones que hemos escrito hasta ahora.

Para apreciar la potencia de una función bien construida vamos a repetir un programa ya realizado, y que conocéis, y transformarlo en una función. Será una variante de **laffer.m** y la llamaremos **autolaff.m**. Esta función tomará como argumentos los parámetros de las funciones de oferta y demanda y nos devolverá todo lo que el archivo **laffer.m** era capaz de hacer. El programa es:

```
function res=autolaff(param)
%Funcion function res=autolaff(param);
%Este programa muestra la curva de Laffer
%para una colección de impuestos
%El vector param debe contener los parámetro [a, b, c, d] de las funciones
%de demanda y oferta donde:
%a es el intercepto de la función de demanda
%b es la pendiente de la función de demanda
%c es el intercepto de la función de oferta
%d es la pendiente de la función de oferta
%El resultado es una matriz res en la que
%La primera columna es el precio pagado por los consumidores
%La segunda columna es el precio percibido por los productores
%La tercera columna en la cantidad en transacción
%La cuarta columna es el excedente del estado
%La quinta columna es el excedente del consumidor
%La sexta columna es el excedente del productor
%La séptima es la pérdida irrecuperable de eficiencia
%La octava columna es el excedente total
```

```
a = param(1);
b = param(2);
c = param(3);
d = param(4);
%Cálculo del equilibrio sin impuestos
Qstar = (a-c)/(b+d);
pstar = (a*d+b*c)/(b+d);
%Máximo número de iteraciones, incremento y tasa inicial
maxit = 1000;
inc = 0.01;
tasa(1) = 0;
for i=1:maxit
    pc(i) = (a*d+b*(c+tasa(i)))/(b+d);
    pv(i) = ((a-tasa(i))*d+b*c)/(b+d);
    Qt(i) = (a-(c+tasa(i)))/(b+d);
    EE(i) = tasa(i)*Qt(i);
    EC(i) = (a-pc(i))*Qt(i)/2;
    EP(i) = (pv(i)-c)*Qt(i)/2;
    PIE(i) = tasa(i)*(Qstar-Qt(i))/2;
    ET(i) = EE(i)+EC(i)+EP(i);
    if Qt(i)<=0; break; end
    tasa(i+1)= tasa(i)+inc;
end
T = length(Qt)-1;
res=[ pc(1:T);
    pv(1:T);
    Qt(1:T);
    EE(1:T);
    EC(1:T);
    EP(1:T);
    PIE(1:T);
    ET(1:T)]';
```

Esta función devuelve una matriz que contiene toda la información sobre precios, cantidades y análisis del bienestar que se pueden extraer del ejemplo. Para hacerla funcionar bastaría con escribir en la ventana de comandos:

```
» res=autolaff(param);
»
```

y ya tendríamos toda la información pedida y guardada en la variable **res**. Para ello es necesario naturamente, que el vector **param** haya sido definido. Vamos a ver un ejemplo de uso de la función **autolaff.m** para ello vamos a hacer un programa del tipo *script* que nos muestre distintas curvas de laffer para distintos valores de la pendiente de la función de demanda. Para conseguirlo hacemos uso del archivo del tipo *function* **autolaff.m**.

```
%Este programa muestra distintas curvas de laffer para distintos valores
%del parámetro b (pendiente de la demanda).
clear
b = [0.1 2 6];
for i=1:length(b)
    param = [10, b(i), 2, 0.5];
    res = autolaff(param);
    plot(res(:,4)); pause(0.1);
    hold on
end
```

Lo primero que llama la atención es lo corto que es el código de un programa que hace tantas cosas. Ahora resulta muy fácil escribir distintos programas en los que cambiamos cosas como el parámetro de la pendiente de la curva de oferta o cualquiera que se nos antoje ya que tenemos una función en la que con sólo definir las variables de entrada o argumentos realiza operaciones tan complejas como queramos. Hay, no obstante, un pequeño problema con el programa anterior. Cada vez que hacemos una pasada por el bucle **for**, la matriz de resultados, **res,** queda sobreescrita, de modo que cuando el programa termina sólo nos queda el resultado de la última iteración, es decir, sólo tenemos los precios, las cantidades y los excedentes cuando la pendiente de la función de demanda es igual a **b(3)**. Para resolver este problema debemos hacer que en cada itearción la matriz de resultados, **res,** cambie de nombre. Por ejemplo, podríamos hacer que la matriz de resultados de la primera iteración sea **res1**, el de la segunda sea **res2** y el de la tercera sea **res3**. Para conseguir esto necesitamos usar el comando **eval**, (ver »**help eval**). Su sintaxis es sencilla si la miramos con atención. Un programa alternativo a **autob.m,** que consiga este resultado de cambiar el nombre en cada iteración, lo tenemos en el programa **autob1.m**, y es así:

```
%Este programa muestra distintas curvas de laffer para distintos valores
%del parámetro b (pendiente de la demanda).
clear
b = [0.1 2 6];
```

```
for i=1:length(b)
    param = [10, b(i), 2, 0.5];
    eval(['res' num2str(i) ' = autolaff(param);']);
end
plot(res1(:,4))
hold on
plot(res2(:,4))
plot(res3(:,4))
```

La función **eval** evalua cadenas de texto y en la forma en la que lo usamos en este programa lo que hace es 'pegar' el resultado de **num2str(i)** a la cadena de texto '**res**' e igualar al resultado de nuestra función **autolaff**. Si depués de la ejecución tecleamos el comando **whos** nos encontramos con el siguiente resultado:

```
» whos
```

Name	Size	Elements	Bytes	Density	Complex
b	1 by 3	3	24	Full	No
i	1 by 1	1	8	Full	No
param	1 by 4	4	32	Full	No
res1	801 by 8	6408	51264	Full	No
res2	801 by 8	6408	51264	Full	No
res3	801 by 8	6408	51264	Full	No

```
Grand total is 19232 elements using 153856 bytes
»
```

Como se puede ver existen tres matrices de la misma dimensión que contienen los resultados de cada una de las tres iteraciones que hemos realizado.

3 Cálculo

En esta sección vamos a emplear códigos de MatLab para aplicarlos a problemas de cálculo. El cálculo se ha mostrado demasido beneficioso para la teoría económica como para no tratarlo en primer lugar. Dentro de esta sección vamos a ver cosas como expansiones de Taylor, cálculo integral, el método de Newton-Raphson para la resolución de sistemas de ecuaciones no lineales, y finalmente algunos ejemplos económicos en los que los métodos de cálculo se aplican.

3.1 Expansión de Taylor

El teorema de Taylor dice que dada una función arbitraria $\phi(x)$, si nosotros conocemos o podemos calcular el valor de la función en $x = x_0$ (es decir, $\phi(x_0)$) y los valores de las derivadas en x_0 (es decir, $\phi'(x_0), \phi''(x_0), \phi'''(x_0)$, etc.), entonces esta función puede ser ser expandida entorno al punto x_0 de la siguiente manera:

$$\phi(x) = \frac{\phi(x_0)}{0!} + \frac{\phi'(x_0)}{1!}(x - x_0) + \frac{\phi''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{\phi^{(n)}(x_0)}{n!}(x - x_0)^n + R_n$$

donde n es un número arbitrario y R_n se intrepreta como un residuo, error, o resto. Para ver los detalles de este interesante teorema ver el libro de Alpha C. Chiang titulado Fundamental Methods of Mathematical Economics, editado por McGraw-Hill, 1984, del que existe una traducción en español y que es un libro muy interesante para aprender matemáticas y muy entretenido por estar lleno de buenos ejemplos económicos.

Veamos un ejemplo: queremos expandir la función $\phi(x) = \frac{1}{1+x}$ entorno al punto $x_0 = 1$, con $n = 4$. Para ello, de acuerdo con el teorema de Taylor, necesitaremos calcular las cuatro primeras derivadas de la función $\phi(x)$ y evaluarla en el punto que deseemos. Hacemos la siguiente tabla:

Derivada	*Evaluación*
$\phi'(x) = -(1+x)^2$	$\phi'(1) = -\frac{1}{4}$
$\phi''(x) = 2(1+x)^{-3}$	$\phi''(1) = \frac{1}{4}$
$\phi'''(x) = -6(1+x)^{-4}$	$\phi'''(1) = \frac{-3}{8}$
$\phi^{(4)}(x) = 24(1+x)^{-5}$	$\phi^{(4)}(1) = \frac{3}{4}$

También podemos ver que $\phi(1) = 1/2$. Así, fijando $x_0 = 1$, usando la información de la tabla y sustituyendo en la expresión del teorema de Taylor nos encontramos:

$$\phi(x) = \frac{1}{2} - \frac{1}{4}(x-1) + \frac{1}{8}(x-1)^2 - \frac{1}{16}(x-1)^3 + \frac{1}{32}(x-1)^4 + R_4$$

Ahora construímos un pequeño programa que represente los resultados para que podamos entender mejor qué quiere decir el teorema de Taylor.

```
%Este programa contruye una expansión de Taylor de órdenes 1, 2, 3 y 4
%entorno al punto x0=1, de la función f(x)=1/(1+x) y hace
%gráficas de dichas expansiones para mostrar el ajuste.
```

```
clear
x = [0.01:0.1:2]; %Abscisa
y = 1./(1+x); %Función a expandir
t1 = (1/2)-(1/4)*(x-1); %Primera expansión
t2 = t1+(1/8)*(x-1).^2; %Segunda expansión
t3 = t2-(1/16)*(x-1).^3; %Tercera expansión
t4 = t3+(1/32)*(x-1).^4; %Cuarta expansión
plot(x, y, x, t1, x, t2, x, t3, x, t4)
```

Es interesante comprobar que si fijamos $n = 1$ estamos haciendo una aproximación lineal a la función que estamos expandiendo. Para verlo en otro ejemplo propongo el siguiente ejercicio.

Ejercicio: Expandir la función cuadrática $\phi(x) = 5 + 2x + x^2$ entorno al punto $x_0 = 1$ fijando $n = 1$.

3.2 Integral definida

Dada una función continua $f(x)$ y dados dos valores del dominio de la variable a y b con $a < b$ se define la integral definida de f en el intervalo como

$$\int_b^a f(x)dx = F(x)|_a^b = F(b) - F(a)$$

Por ejemplo queremos evaluar

$$\int_1^5 3x^2 dx$$

Dado que el valor de la integral indefinida es $F(x) = x^3 + c$ el valor es

$$\int_1^5 3x^2 dx = x^3\Big|_a^b = 125 - 1 = 124.$$

Nosotros podemos interpretar el valor de la integral definida como el área que queda por debajo de una curva y entre los límites de integración, por tanto podemos escribir esta área como

$$A = \sum_{i=1}^{n} f(x_i)\Delta x_i$$

y aproximar el valor de la integral a través de esta expresión. Un programa que haga este trabajo es tan sencillo como escribir

```
%Este programa aproxima el valor de la integral definida de la
%función f(x)=3*x^2 entre los límites de integración a=1 y b=5.
clear
linf = 1; %Límite inferior
lsup = 5; %Límite superior
h = 0.0001; %Incremento
x = [linf:h:lsup];
y = 3*x.^2;
int = sum(y*h);
```

la ejecución de este programa con distintos valores de h daría lugar a la siguiente tabla

$h = 1$	$int = 165$
$h = 0.1$	$int = 127.92$
$h = 0.01$	$int = 124.3902$
$h = 0.001$	$int = 124.0390$
$h = 0.0001$	$int = 124.0039$

El programa sorprende por su sencillez. Para hacerlo mejor aún, podríamos hacer una función que nos dé el valor de la integral con sólo especificar en los argumentos de la función el límite inferior de integración, el superior, el incremento y una cadena de texto que contenga a la función. A esta función la vamos a llamar **autoint.m** y su funcionamiento sería:

```
function int=autoint(a, b, h, s)
%Esta función evalua la integral definida por la cadena de texto
% s entre los límites a y b con incremento h.
%Uso: int=autoint(linf,lsup,h,'expresión f(x)')
%Es importante en este programa que la función s tenga como
%argumento a x
x = [linf:h:lsup];
int = sum(eval(s)*h);
```

Si escribimos sobre la ventana de comandos **int=autoint(1,5,0.1,'3*x.^2')** obtenemos el siguiente resultado

```
» int=autoint(1,5,0.1,'3*x.^2')
int =
127.9200
```

»

que es el resultado esperado. Del mismo modo podríamos ahora escribir por ejemplo int=autoint(-1,1,0.001, 'x. ^2') para obtener el valor de la integral de una parabola entre los valores -1 y 1. El elemento más novedoso de este programa es el uso de la función construida por MatLab llamada **eval** (ver »**help eval**) con la que podemos evaluar una función dada por una cadena de texto.

3.2.1 Ejemplo 1

Dada la función de utilidad $u(c_1, c_2) = c_1^{\alpha} c_2^{1-\alpha}$, y la restricción presupuestaria $p_1 c_1 + p_2 c_2 = M$, deseamos saber cuánto varía el excedente del consumidor cuando los precios del bien c_2 disminuyen. Para averiguarlo primero planteamos el problema

$$\underset{c_1,c_2}{Max}\ \alpha \log c_1 + (1-\alpha)\log c_2$$
$$s.a.\ \ p_1 c_1 + p_2 c_2 = M$$

y la función auxiliar de Lagrange $L(c_1, c_2, \lambda) = \alpha \log c_1 + (1-\alpha)\log c_2 - \lambda\left[M - p_1 c_1 - p_2 c_2\right]$. Haciendo las derivadas parciales encontramos el siguiente sistema:

$$\begin{aligned}
\frac{\partial L}{\partial c_1} &= \alpha \frac{1}{c_1} - \lambda p_1 = 0 \\
\frac{\partial L}{\partial c_2} &= (1-\alpha)\frac{1}{c_2} - \lambda p_2 = 0 \\
\frac{\partial L}{\partial \lambda} &= M - p_1 c_1 - p_2 c_2 = 0
\end{aligned}$$

que al resolverlo para las variables c_1, c_2 y λ obtenemos

$$\begin{aligned}
c_1 &= \frac{\alpha M}{p_1} \\
c_2 &= \frac{(1-\alpha)M}{p_2} \\
\lambda &= \frac{1}{M}
\end{aligned}$$

La función inversa de demanda del bien 2 viene dada por $p_2 = \frac{(1-\alpha)M}{c_2}$. Para calcular el área entre dos consumos dados basta con determinar los límites de integración, el valor de α y el valor de la renta M, y luego hacer uso de nuestra función **autoint.m**.

3.3 El método de Newton-Raphson

3.3.1 Los ceros de una función

Encontrar los ceros de una función de variable real va a ser para nosotros una tarea muy interesante ya que en el análisis económico nos vamos a encontrar con que todos los modelos al final se reducen a hallar los ceros de una función. Desafortunadamente sólo en ocasiones será una tarea muy sencilla. Por ejemplo, encontrar los ceros de la función $y = x - 2$ es fácil ya que el cero de la función $0 = x - 2$ se encuentra en $x = 2$. Pese a lo sencilla que ha resultado la operación os voy a mostrar cómo se hace un programa para resolver este problema.

```
%Este programa encuentra el cero de la función y=x-2
clear
x = -0.3;
h = 0.01;
maxit = 500;
for i=1:maxit
     if sign(x-2)==1
          x = x-h;
     else
          x = x+h;
     end
end
```

Este programa empieza con un valor arbitrario de **x** fijado en $x = -0.3$, se fija un incremento y un número máximo de iteraciones. Después entra en un bucle **for** en el que vemos la sentencia condicional que pregunta por el signo de la expresión **(x-2)**. Si ésta es positiva entonces de **sign(x-2)** sale el valor 1 y en consecuencia tratamos de encontrar un nuevo valor de **x** más pequeño con la expresión **x=x-h**, en caso contrario incrementamos el valor de **x**. Si (ineficientemente) hacemos esto muchas veces acabaremos llegando al punto que buscábamos. Este programa sólo tiene interés en su lógica más básica y es que, si estamos buscando un cero de una función, en realidad estamos buscando el punto en el que a derecha e izquierda del punto, la función cambia de signo.

El problema anterior era tan fácil que en caso de encontrarlo en un problema real no haremos uso del ordenador para resolverlo. Si nos encontramos con otro tipo de problemas en los que por ejemplo tenemos que encontrar

la solución a un sistema de ecuaciones no lineal tendremos que utilizar otro tipo de técnicas. El algoritmo más comunmente utilizado es el algoritmo de Newton-Raphson que veremos a continuación.

3.3.2 La lógica del algoritmo de Newton-Raphson

Podemos describir cualquier sistema de n ecuaciones en n incognitas como $F : R^n \longrightarrow R^n$. Nuestro problema será enconces encontar un vector $\widehat{x} = (\widehat{x}_1, \cdots, \widehat{x}_n)$ de R^n tal que su imagen por $F : R^n \longrightarrow R^n$ sea $F(\widehat{x}) = 0$. El algoritmo basado en el método de Newton-Raphson pretende encontrar una solución a este sistema de la forma $F(\widehat{x}) = 0$. Para encontrar el $\widehat{x}$ solución del sistema se aproxima la función a través de la primera expansión de Taylor a la función F.

$$F(x) = F(\overline{x}) + J(\overline{x})(x - \overline{x}), \tag{1}$$

donde $J(\overline{x})$ es la matriz jacobiana de F evaluada en $\overline{x}$, es decir,

$$J(\overline{x}) = \begin{bmatrix} F_{11}(\overline{x}) & F_{12}(\overline{x}) & F_{1n}(\overline{x}) \\ F_{21}(\overline{x}) & F_{22}(\overline{x}) & F_{2n}(\overline{x}) \\ & & \\ F_{n1}(\overline{x}) & F_{n2}(\overline{x}) & F_{nn}(\overline{x}) \end{bmatrix},$$

donde $F_{ij}(\overline{x}) = \frac{\partial F_i(\overline{x})}{\partial x_j}$.

Dado que estamos buscando un cero de la ecuación $F(x)$, la ecuación 1 podemos evaluarla en $\widehat{x}$ y escribirla como,

$$\widehat{x} = \overline{x} - J(\overline{x})^{-1} F(\overline{x}).$$

El algoritmo funcionaría de la siguiente manera:

1. Proponemos una semilla x_1 y evaluamos la función $F(x_1)$ y $J^{-1}(x_1)$, para calcular

$$x_2 = x_1 - J(x_1)^{-1} F(x_1)$$

2. Fijamos un nivel de tolerancia ε y calculamos alguna distancia entre x_2 y x_1. Si la distancia es inferior a ε, entonces nos quedamos con x_2 como solución. En caso contrario volvemos al paso 1 y evaluamos la función $F(x_2)$ y $J^{-1}(x_2)$, para calcular

$$x_3 = x_2 - J(x_2)^{-1} F(x_2)$$

Realizamos estas operaciones tantas veces como sea necesario hasta que encontremos un x_t y x_{t+1} tales que la distancia sea menor que el nivel de tolerancia.

Ejemplo 1 Veamos ahora cómo funciona con un ejemplo sencillo. Queremos encontrar los ceros de la función

$$y = (x - 4)(x + 4).$$

La simple inspección visual muestra que los ceros de la función se encontrarán en $x = 4$ y $x = -4$. Ahora vamos a construír nuestro primer programa de MatLab con el método de Newton-Raphson para encontrar esas soluciones.

```
%ej1.m
%Este programa resuelve la ecuación sencilla del ejemplo 1
%La ecuación es: y = (x - 4)(x + 4)
clear
%Punto inicial
x(1) = -10;
%Máximo número de iteraciones
maxit=1000;
for s=1:maxit
    J(s) = 2*x(s);
    x(s+1) = x(s)-J(s)^(-1)*(x(s)-4)*(x(s)+4);
    if abs(x(s+1)-x(s))<1e-20; break; end
end
if s>=maxit
sprintf('Atención: Número máximo de %g iteraciones alcanzado', maxit)
end
sprintf('La solución es %g', x(s))
```

El programa ha empezado con una semilla $x_1 = -10$, y por tanto la solución que el programa nos dará es $x^* = -4$. Si hubieramos introducido como semilla el valor $x_1 = 10$, el programa habría dado como solución $x^* = 4$. En este simple ejemplo vemos algo esencial del algoritmo de Newton: las semillas son de una importancia capital para encontrar la solución a un problema. La localización de la semilla en relación a la solución tiene un impacto en la respuesta del programa. Cuanto más cerca esté la semilla a la solución que buscamos tanto mejor.

Ejercicio: Hallar los ceros de la función $y = x^3 + 2x^2 - 24x + 2$.

En este sencillo problema de hallar los ceros de una función de R en R hemos podido computar el jacobiano sin ningún problema, pero si la función hubiera sido de R^n en R^n, con n muy grande, entonces computar el jacobiano se puede convertir en un problema tan complicado como el de hallar los ceros

del sistema. Para resolver este problema podemos computar las derivadas numéricas de la función. La definición de derivada nos dice que

$$\begin{aligned} J_1(x) &= \lim_{h_1 \to 0} \frac{F(x) - F(x_1 + h, x_2, \cdots, x_n)}{h_1}, \\ J_2(x) &= \lim_{h_2 \to 0} \frac{F(x) - F(x_1, x_2 + h, \cdots, x_n)}{h_2}, \\ \vdots &= \qquad \vdots \\ J_n(x) &= \lim_{h_n \to 0} \frac{F(x) - F(x_1, x_2, \cdots, x_n + h)}{h_n}, \end{aligned}$$

donde $J_i(x)$ es el vector columna con las n derivadas parciales con respecto a x_i. Por tanto $J(x) = [J_1(x), J_2(x), \cdots, J_n(x)]$. Podemos fijar en un ordenador un incremento h suficientemente pequeño como para poder escribir

$$\begin{aligned} J_1(x) &\approx \frac{F(x) - F(x_1 + h_1, x_2, \cdots, x_n)}{h_1}, \\ J_2(x) &\approx \frac{F(x) - F(x_1, x_2 + h_2, \cdots, x_n)}{h_2}, \\ \vdots &\approx \qquad \vdots \\ J_n(x) &\approx \frac{F(x) - F(x_1, x_2, \cdots, x_n + h_n)}{h_n}, \end{aligned}$$

y de esta manera aproximar las derivadas parciales.

Ejemplo 2 Ahora queremos encontrar los ceros de la función del Ejemplo 1, pero haciendo uso de una aproximación numérica a la derivada de y con respecto a x. Para ello construímos el siguiente programa en el que definimos un cociente incremental para el jacobiano y un incremento que fijamos en **h = 1e-8**.

```
%Este programa resuelve por el metodo de Newton-Raphson la
%ecuación trivial y = (x + 4) * (x - 4) haciendo uso de una
%aproximación numérica a la derivada de y con respecto a x
clear
%Punto inicial
x0 = -10;
%Número máximo de iteraciones permitido
maxit = 1000;
```

```
%Incremento
h = 1e-8;
x(1) = x0;
for s=1:maxit
    J(s) = 1/(2*h)*((x(s)+h+4)*(x(s)+h-4)-(x(s)-h+4)*(x(s)-h-4));
    x(s+1) = x(s)-J(s)^(-1)*(x(s)-4)*(x(s)+4);
    if abs(x(s+1)-x(s))<1e-20; break; end
end
if s>=maxit
sprintf('Atención: Numero máximo de %g iteraciones alcanzado', maxit)
end
sprintf('La solución es %g', x(s))
```

3.4 Sistemas no lineales

Queremos encontrar el par (x^*, y^*) que resuelve el sistema de ecuaciones

$$\begin{aligned} ay - (x-b)(x+c) &= 0 \\ yx - d &= 0. \end{aligned}$$

Para resolver este problema usando el algoritmo de Newton-Raphson debemos calcular el jacobiano del sistema.

$$J = \begin{bmatrix} -2x + (b-c) & a \\ y & x \end{bmatrix}$$

Una vez que lo tenemos calculado basta con reescribir el programa del ejemplo 1 de la sección anterior para acomodarlo a un caso en el que tenemos dos funciones y dos variables. El programa sería el siguiente:

```
%Este programa resuelve por el metodo de Newton-Raphson el sistema
de
%ecuaciones
%a*y-(x-b)*(x+c)=0
%y*x-d=0
clear
%Parámetros
a = 1;
b = 4;
c = 4;
```

```
d = 1;
%Punto inicial
x0 = [-0.01; 0.01];
x = x0;
%Número máximo de iteraciones permitido
maxit = 1000;
%Criterio de convergencia
crit = 1e-4;
for i=1:maxit
    J = [-2*x(1)+(b-c) a; x(2) x(1)];
    f = [a*x(2)-(x(1)-b)*(x(1)+c); x(2)*x(1)-d];
    x = x0-inv(J)*f;
    if norm(x-x0)<crit; break; end;
    x0=x;
end
if i>=maxit
sprintf('Atención: Numero máximo de %g iteraciones alcanzado', maxit)
end
sprintf('La solución de x es %g', x(1))
sprintf('La solución de y es %g', x(2))
```

Ejemplo

Deseamos calcular los niveles de consumo que desea un consumidor con función de utilidad $U(c_1, c_2) = \frac{1}{\rho}(\alpha c_1^\rho + (1-\alpha)c_2^\rho)$ cuando se enfrenta a la restricción presupuestaria $p_1c_1+p_2c_2 = M$. Para resolver el problema primero planteamos la función auxiliar de Lagrange $L(c_1, c_2, \lambda) = \frac{1}{\rho}(\alpha c_1^\rho + (1-\alpha)c_2^\rho) - \lambda[M - p_1c_1 - p_2c_2]$ y luego formamos el sistema de ecuaciones

$$\begin{aligned}
\frac{\partial L}{\partial c_1} &= \alpha c_1^{\rho-1} - \lambda p_1 = 0 \\
\frac{\partial L}{\partial c_2} &= (1-\alpha)c_2^{\rho-1} - \lambda p_2 = 0 \\
\frac{\partial L}{\partial \lambda} &= M - p_1c_1 - p_2c_2 = 0
\end{aligned}$$

Dividiendo la primera condición por la segunda y reescribiendo la tercera obtenemos un sistema de dos ecuaciones en dos incógnitas

$$\frac{\alpha c_1^{\rho-1}}{(1-\alpha)c_2^{\rho-1}} - \frac{p_1}{p_2} = 0$$

$$M - p_1 c_1 - p_2 c_2 \quad = \quad 0$$

Ejercicio: resolver el anterior sistema cuando $\alpha = 0.3, \rho = 0.96, M = 10, p_1 = 1$ y $p_2 = 3$.
Ejercicio: construír una función en la que metiendo los valores de los parámetros como argumentos obtenemos los consumos como respuesta.