1 Lección segunda

Una vez que hemos entendido la lógica del algoritmo de Newton-Raphson y la aproximación numérica a las derivadas parciales nos gustaría disponer de un programa más general que resolviera estos problemas con sólo especificar el sistema de ecuaciones que queremos resolver. Esto es posible hacerlo gracias a que disponemos de rutinas muy eficientes que nos permiten resolver el problema de encontrar las soluciones a un sistema de ecuaciones llamando a esas rutinas. A continuación vamos a presentar dos de esas rutinas Urrutia (1998), una para el método de Newton, llamada **newton.m** y otra para el método de aproximación numérica a las derivadas parciales llamada **secant.m**. No son las únicas y de hecho Matlab posee en el "toolbox optim" una rutina llamada **fsolve.m** que permite resolver sistemas de ecuaciones no lineales y que incorpora tanto al método de Newton como al de aproximación numérica de las derivadas, además de poder especificar criterios de tolerancia, máximo número de iteraciones etc. La razón por la que usaremos fundamentalmente el programa **secant.m** se debe a que es fácil intervenir en este programa para obtener información sobre el proceso de optimización. Pero para entender esto, lo mejor es presentar y discutir brevemente estos programas**.** La diferencia entre el programa **newton.m** y el programa **secant.m** es simplemente que en el primero nosotros especificamos una función en la que está definido el jacobiano del sistema y en el otro hacemos una aproximación numérica a las derivadas parciales, con lo que sólo es necesario especificar en el programa el sistema de ecuaciones. Normalmente utilizaremos el programa **secant.m** dado que a medida que los problemas que nos planteemos sean más complejos no dispondremos de información sobre el jacobiano o éste será demasiado engorroso como para introducirlo dentro del programa.

```
function x = newton(func, x0, param, crit, maxit)
%newton.m      Programa para resolver un sistema de cuaciones
%simultáneas.
%x=newton(func, x0, param, crit, maxit) usa el método de
%Newton-Raphson para resolver el siguiente sistema:
%              f1(z1, z2, ···, zn)=0
%              f2(z1, z2, ···, zn)=0
%              ⋮                              (*)
%              fn(z1, z2, ···, zn)=0
%en donde x=[z1, z2, ···, zn]' es el vector que resuelve (*).
%Para usar esta función, debe especificarse 'func' que es un "string"

%con el nombre del archivo .m que contiene la función f y su matriz
%jacobiana J. En caso necesario, puede usarse el vector
%param para incluir parámetros adicionales de dicha función.
%Los argumentos x0, crit, y maxit son la semilla inicial para x,
%el criterio de convergencia (tolerancia) y el número máximo de
%iteraciones permitidas respectivamente.
%Programa de Carlos Urrutia
for i=1:maxit
        [f, J]=feval(func, x0, param);
        x=x0-inv(J)*f;
        if norm(x-x0)<crit;      break;      end
        x0=x;
end
if i>=maxit
sprintf('Advertencia: Número máximo de %g iteraciones ...
alcanzado', maxit)
end
```

Lo primero que observamos al mirar al archivo **newton.m** es que comienza con el comado **function.** En Matlab hay dos tipos de archivo que tienen extensión **.m,** los primeros que hemos visto en los ejemplos 1 y 2, son archivos del tipo "script" y contienen una colección de comandos de Matlab que son ejecutados con sólo declarar el nombre del archivo sobre la ventana de comandos de Matlab tras el símbolo >>. Los segundos son los que estamos viendo ahora, son del tipo "function" y tambien contienen comandos de Matlab pero necesariamente tienen que empezar con el comado **function**. Su utilidad consiste en que dentro de ellos podemos definir nuevas funciones y lo que haremos será definir en su interior el sistema de ecuaciones (funciones) que queremos resolver. Para ver cómo funciona plantearemos un ejercicio sencillo.

1.0.1 Ejemplo3

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 la función **newton.m** debemos calcular el jacobiano del sistema.

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

A continuación escribimos un programa similar a los que ya hemos escrito para resolver los ejemplos 1 y 2, con la salvedad de que ahora vamos a introducir el sistema de ecuaciones y el jacobiano en un archivo a parte y vamos a llamar a la rutina **newton.m** para que se encarge del asunto. Los pasos son los siguientes: primero construimos un archivo llamado **sistema.m** donde definimos nuestro problema, fijamos la semilla con la que el algoritmo debe empezar, fijamos el número máximo de iteraciones, fijamos el criterio de tolerancia, fijamos el valor de los parámetros y llamamos al programa **newton.m** indicando el nombre del archivo sobre el que tiene que operar y que vamos a llamar **aqui.m,** puesto que aqui es donde vamos a escribir el valor de la función y el valor del jacobiano.

```
%Archivo sistema.m
%Este programa resuelve por el metodo de Newton-Raphson
%el sistema de ecuaciones general
%
%ay-(x+b)*(x-c) = 0
%xy-d = 0
%
%para el caso particular
%
%y-(x+4)*(x-4) = 0
%xy-1 = 0
%
%Para ello hacemos uso de la función newton.m
clear
%Punto inicial
%x0 = [-0.001; 0.001];
%x0 = [-0.001; -2];
x0 = [-0.001; -0.001];
%Número máximo de iteraciones permitido
maxit = 1000;
%Criterio de convergencia (tolerancia)
crit = 1e-3;
%Vector de parámetros del sistema
param = [1 4 4 1];
%Llamamos a newton.m y especificamos el archivo sobre el que
debe actuar
sol = newton('aqui', x0, param, crit, maxit);
sprintf('x= %g', sol(1))
sprintf('y= %g', sol(2))
```

Ahora mostramos el contenido del archivo **aqui.m** que como podeis ver es un archivo del tipo function puesto que en él se define una función que debe ser evaluada.

```
function [f, J]=aqui(z, p)
%archivo aqui.m
x = z(1);
y = z(2);
a = p(1);
b = p(2);
c = p(3);
d = p(4);
f = [a*y-(x-b)*(x+c); y*x-d];
J = [-2*x+b-c a; y x];
```

El comando **function [f,J]=aqui(z,p)** dice que el resultado de los cálculos realizados son las matrices **f** (de dimensión 2x1) y **J** de dimensión (4x4), y que el archivo **aqui.m** tiene dos argumentos, el primero es **z** que se identifica con el valor de **x0** del archivo **sistema.m** y el segundo es **p,** que se identifica con el vector **param** del mismo archivo. Las lineas sucesivas asignan valores a las variables y a los parámetros.
Ahora podemos ver cómo funciona el programa **secant.m** en el mismo ejemplo.

1.0.2 Ejemplo 4

Las dos únicas diferencias con respecto a los programas del ejemplo 3 son que naturalmente en el programa **sistema.m** ya no vamos a llamar al programa **newton.m** sino al programa **secant.m**, y la otra diferencia es que el programa **aqui.m** ya no necesita tener especificado el jacobiano del sistema de ecuaciones puesto que la rutina **secant.m** hace una aproximación numérica al jacobiano. El programa **aqui.m** quedaría asi:

```
function f=aqui(z, p)
%Programa aqui.m sin el jacobiano
x = z(1);
```

```
y = z(2);
a = p(1);
b = p(2);
c = p(3);
d = p(4);
f = [a*y-(x-b)*(x+c); y*x-d];
```

Como vemos, el valor del jacobiano no es computado en esta versión de **aqui.m**, ya que en la rutina **secant.m** que mostramos a continuación se realiza una aproximación numérica al jacobiano.

```
function x=secant(func, x0, param, crit, maxit)
%secant.m Programa para resolver un sistema de ecuaciones
%simultaneas.
% x=secant(func, x0, param, crit, maxit) usa el metodo de secante
% para resolver el siguiente sistema:
%
% f1(z1,z2, ...,zn)=0
% f2(z1,z2, ...,zn)=0 (*)
% : :
% fn(z1,z2, ...,zn)=0
%
% donde x=[z1,z2, ...,zn]' es el vector que resuelve (*).
% Para usar esta función, debe especificarse 'func' que es un string
% con el nombre de un archivo .m que contiene la función f.
%En caso necesario, puede usarse el vector 'param' para incluir
%parámetros adicionales de dicha función.
% Los argumentos x0, crit y maxit son la semilla inicial para x,
% el criterio de convergencia (tolerancia) y el número máximo de
% iteraciones permitidas.
% Programa de Carlos Urrutia
del = diag(max(abs(x0)*1e-4, 1e-8));
n = length(x0);
          for i=1:maxit
           f=feval(func,x0,param);
            for j=1:n
             J(:,j)=(f-feval(func,x0-del(:,j),param))/del(j,j);
          end
            x=x0-inv(J)*f;
            if norm(x-x0)<crit;       break;       end
            x0=x;
           end
if i>=maxit
sprintf('Advertencia: Número máximo de %g iteraciones ...
alcanzado', maxit)
end
```