Análisis Numérico I:

Resolución de ecuaciones no lineales

Alumno:
Fernando Sanz Clavijo

Asignatura:
Análisis Numérico I

Práctica **1**:
Resolución de ecuaciones no lineales

Indice

· Enunciado de la Práctica

Básicamente el objetivo de esta práctica es encontrar las soluciones de una función determinada, más concretamente encontrar la única solución de la siguiente ecuación:

$$f(x) = x^7 + 6x^3 - 2x^2 + 3x - 2$$

Para ello se emplearán varios de los métodos estudiados hasta el momento, como puedan ser: Bisección, Punto fijo, Aiken, Newton-Raphson y Steffensen.

Aunque como primera aproximación se utilizará la representación gráfica de la ecuación a estudiar.

· Estudio Gráfico

Fundamento teórico

Para comenzar con la búsqueda de las soluciones de la ecuación se empleará el método gráfico, en el que inicialmente se representará la función en un intervalo que podríamos denominar grande, más concretamente en el intervalo [-10, 10]. Y a partir de la observación de dicha gráfica se podría hacer una nueva representación más cercana a la solución, por ejemplo la representación en el intervalo [0, 1].

Aparentemente la solución de la función de estudio parece única, pero es necesario un método de rigor para realizar dicha afirmación, en este caso el método que se empleará será el estudio del crecimiento, es decir, el estudio del signo de la primera derivada en el intervalo de trabajo.

Metodología

Para la realización del estudio gráfico inicialmente se construirá un archivo (**f.m**) que contenga la función a estudiar, que tenga como entrada un valor de X determinado y nos devuelva el valor de la función para dicho valor de entrada. La representación gráfica en sí, será realizada por otro archivo (**grf_f.m**) que simplemente, tomará una serie de puntos equidistantes en el intervalo de entrada, calculará el valor de la función para estos puntos, y representará los pares formados por dichas entradas y salidas.

El estudio gráfico de la derivada es totalmente análogo al de la función original, simplemente se creará otro archivo (**fprima.m**) que contenga la derivada de la función a estudiar. Y la representación gráfica también será realizada por otro archivo (**grf_fp.m**) prácticamente idéntico al anterior, en el cual a partir de unos valores de entrada se representará una serie de pares formados por dichas entradas y los correspondientes valores de la derivada para dichas entradas.

Una vez implementados los archivos anteriores, representar la función f(x) en el intervalo [-10,10] se limita a una única orden del tipo:

```
» grf_f(-10,10,0.1)
```

La realización de la ampliación de esta gráfica para el intervalo [0,1] igualmente se reduce a la orden siguiente:

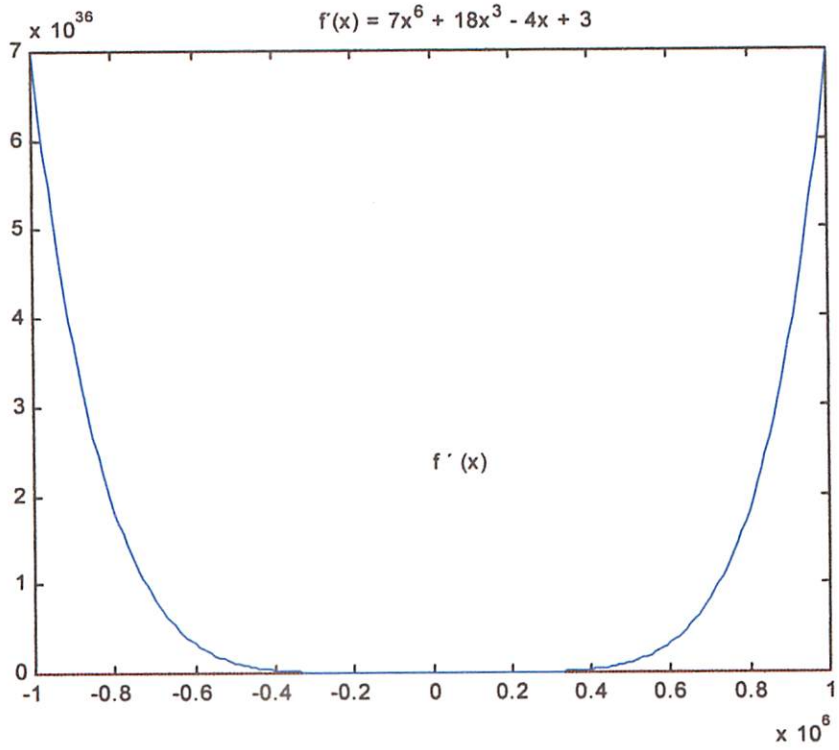
```
» grf_f(0,1,0.01)
```

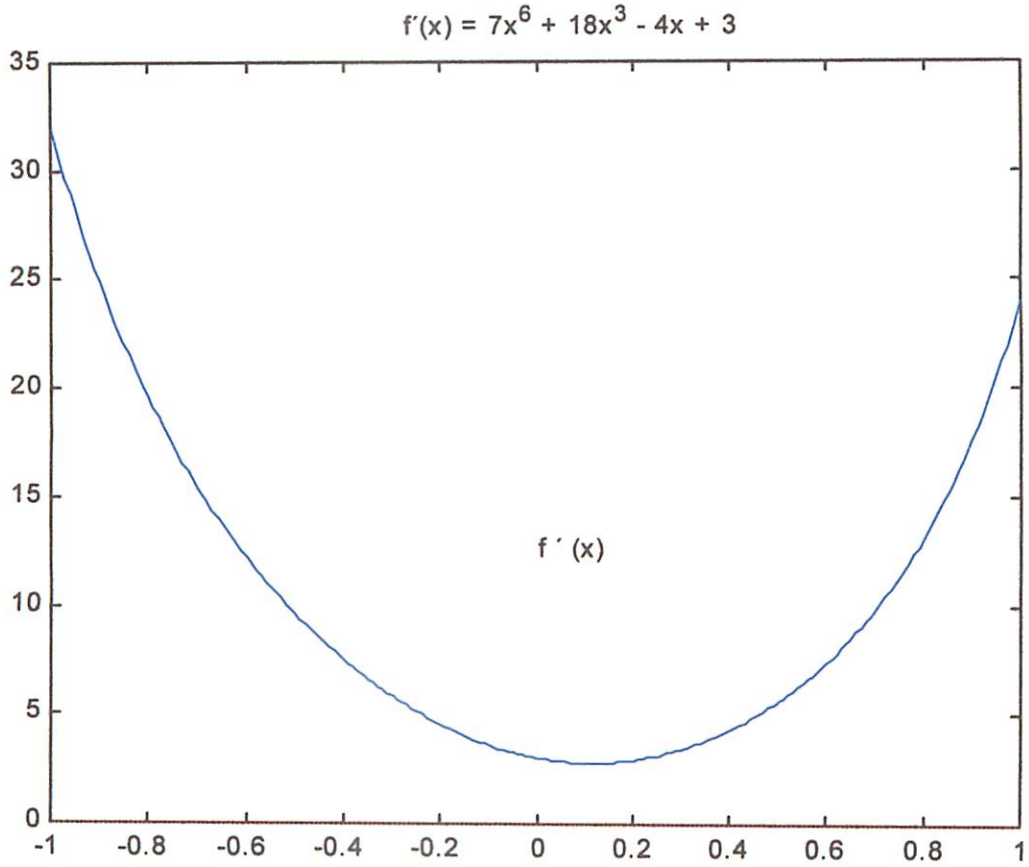
Para concluir con el estudio gráfico, falta comprobar que la solución es única en toda la recta real, para ello se realizará la representación de f ´(x) en varios intervalos, con lo cual se podrá comprobar que la derivada es siempre positiva, lo que implica que la función original es siempre creciente, y por lo tanto tan sólo puede tener un corte con el eje horizontal, es decir, una única solución siempre que la función original sea continua, y en este caso lo es.

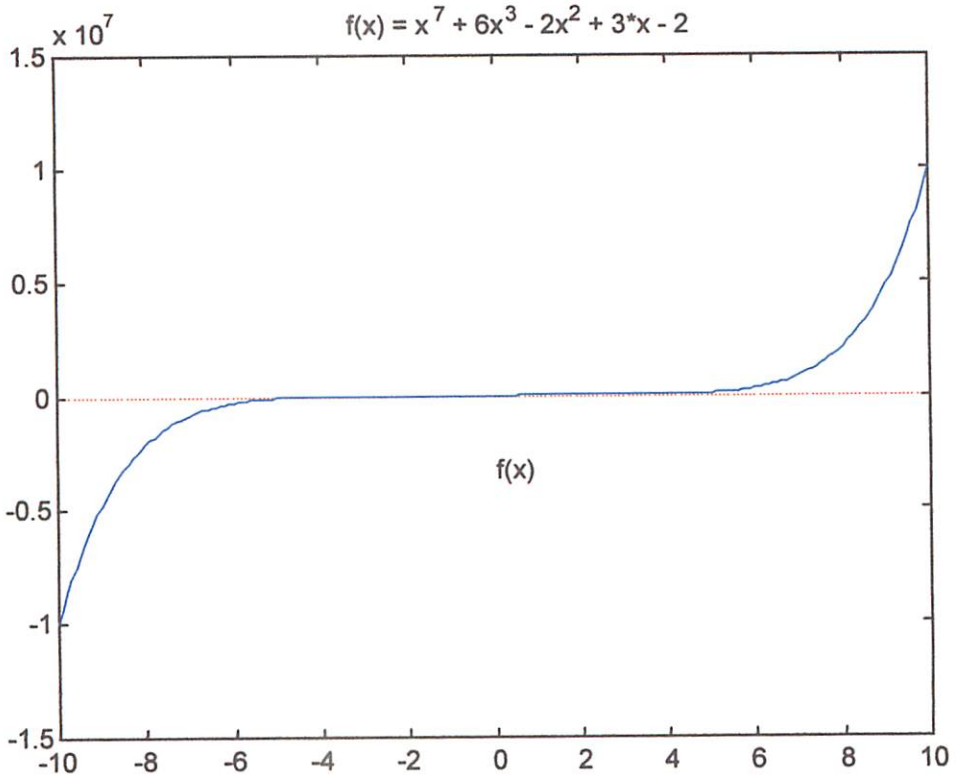
De forma análoga a la representación de la función original, la representación de la derivada en varios intervalos se reduce a las órdenes siguientes:

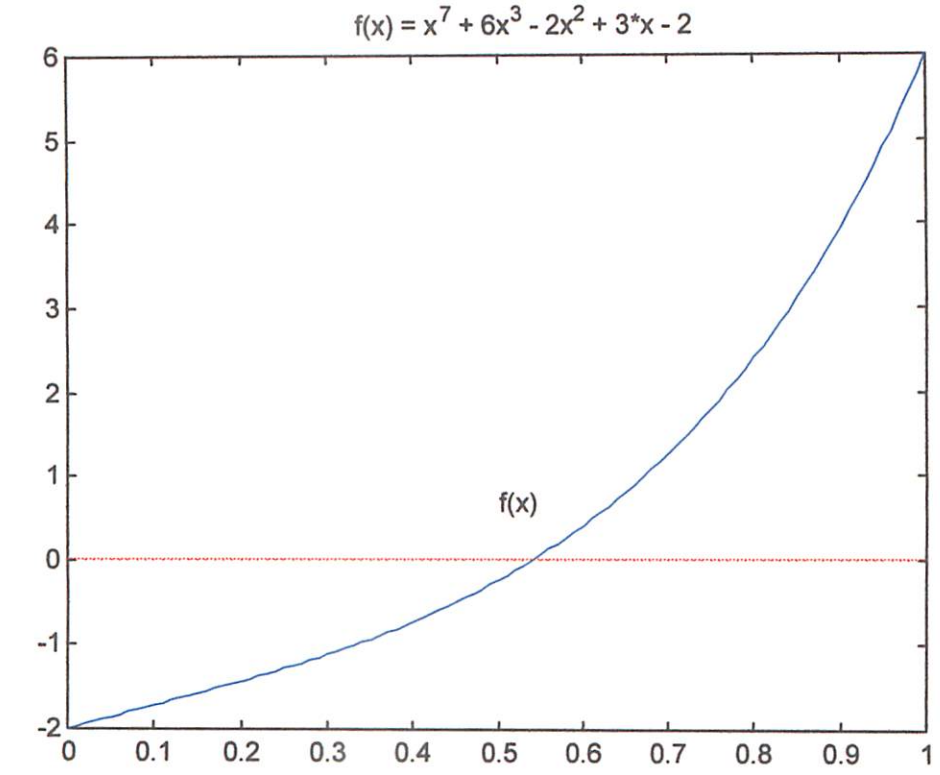
```
» grf_fp(-1000000,1000000,10000)
» grf_fp(-1,1,0.01)
```

Los resultados de todas las gráficas anteriores pueden verse a continuación:






x 10^7
f(x) = x^7 + 6x^3 - 2x^2 + 3*x - 2
1.5
1
0.5
0
-0.5
-1
-1.5
-10
-8
-6
-4
-2
0
2
4
6
8
10
f(x)


f(x) = x^7 + 6x^3 - 2x^2 + 3*x - 2
6
5
4
3
2
1
0
-1
-2
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
f(x)

Implementación y resultados

Los algoritmos necesarios hasta el momento, es decir, los empleados para el cálculo de la función original (**f.m**), la derivada de ésta (**fprima.m**), así como las representaciones gráficas de las mismas son muy sencillos, por lo cual no creo que sea necesario explicar detalladamente el funcionamiento de los mismos.

Si acaso, como anotación explicar brevemente el funcionamiento de los representadores gráficos (**grf_f.m** y **grf_fp.m**), que inicialmente reordenan los limites del intervalo a representar, y los divide en fragmentos iguales, para posteriormente calcular el valor de la función deseada (mediante **f.m** o **fprima.m**), representar los pares resultantes y añadir las etiquetas necesarias. Como ambos archivos son prácticamente idénticos, tan sólo se listará aquí el código de uno de ellos.

Como resultado de este método, se obtiene que la solución está alrededor de X=0.55, pero lógicamente es una apreciación muy inexasta.

f.m

```
function y=f(x)
% f(x) calcula en el punto dado, el valor de
%  la funcion:  x.^7 + 6*x.^3 - 2*x.^2 + 3*x - 2
y= x.^7 + 6*x.^3 - 2*x.^2 + 3*x - 2;
```

f prima.m

```
function y=fprima(x)
% fprima(x) calcula en el punto dado, el valor de
%  la funcion:  7*x.^6 + 18*x.^2 - 4*x + 3,
%  que es la derivada de f(x)=x.^7 + 6*x.^3 - 2*x.^2 + 3*x - 1
y=7*x.^6 + 18*x.^2 - 4*x + 3;
```

grf_f.m

```
function [x,y]=grf_f(x1,x2,Inc_x)
% grf_f(x1,x2,Inc_x) dibuja la funcion:  f(x)
%  en el intervalo [x1,x2], con incrementos
%  iguales a "Inc_x"

echo off all
if x1<=x2
   x=x1:Inc_x:x2;
   x_texto=x1+(x2-x1)/2;
else
   x=x2:Inc_x:x1;
   x_texto=x2+(x1-x2)/2;
end

y=f(x);          % y=fprima(x)     si fuese "grf_fp.m"

y_texto=min(y)+(max(y)-min(y))/2+0.5*10^7;
axis([min(x),max(x),min(y),max(y)]);
plot(x,0,':r',x,y,'-b'),
title('f(x) = x^7 + 6x^3 - 2x^2 + 3*x - 2'),
text(x_texto,y_texto,'f(x)');
```

· Método de Bisección

Fundamento teórico

El método de bisección se basa principalmente en aplicación del teorema de Bolzano, que afirma que si una función continua cambia de signo en los extremos de un intervalo cerrados, dentro de dicho intervalo existe al menos una solución de dicha función.

A partir de esto, el algoritmo de bisección lo que realiza es precisamente dividir en dos partes iguales el intervalo que contiene la solución, es decir, si inicialmente tenemos un determinado intervalo en el cual existe cambio de signo entre sus extremos, buscamos su punto medio y reducimos la búsqueda al medio intervalo en el cual todavía exista cambio de signo, y así sucesivamente hasta aproximarnos a la solución tanto como deseemos.

Como se puede deducir a partir del método empleado para la búsqueda de la solución, el algoritmo de bisección es bastante lento, por lo cual se suele emplear como "lanzadera" para otros métodos más veloces, ya que aunque lento, el método de bisección es seguro, es decir, siempre converge hacia la solución. Otra de las bondades de este algoritmo es que nos permite acotar el error cometido en cada iteración muy fácilmente, ya que sabemos que el error máximo que cometemos es la mitad de la longitud del intervalo en cada iteración.

Metodología

La implementación que he realizado del algoritmo de bisección (**bisecion.m**) funciona como se describe a continuación: inicialmente se comprueba si la solución es alguno de los extremos del intervalo de estudio, sino esa sí (lo cual ocurrirá normalmente) se comprueba si realmente existe cambio de signo en la función a estudiar en los extremos de dicho intervalo, si no hubiese cambio de signo se produciría un error si no se continuaría con la ejecución normal del algoritmo. Una vez comprobados estos casos especiales se calcula el número máximo de iteraciones necesarias para llegar a la solución, y se entra en un bucle que irá reduciendo el intervalo de estudio a la mitad en cada iteración, y comprobando en cada una de ellas que no se ha obtenido la solución, o que no se ha sobrepasado el número máximo de iteraciones permitidas. Como anteriormente se comentó este método nos permite ir calculando el error cometido en cada iteración con muy poco gasto computacional, por lo cual se utilizará como criterio de paro el que dicho error sea menor al que deseamos, es decir, que daremos por encontrada la solución cuando el error cometido sea menor al que inicialmente se pasa como parámetro al algoritmo.

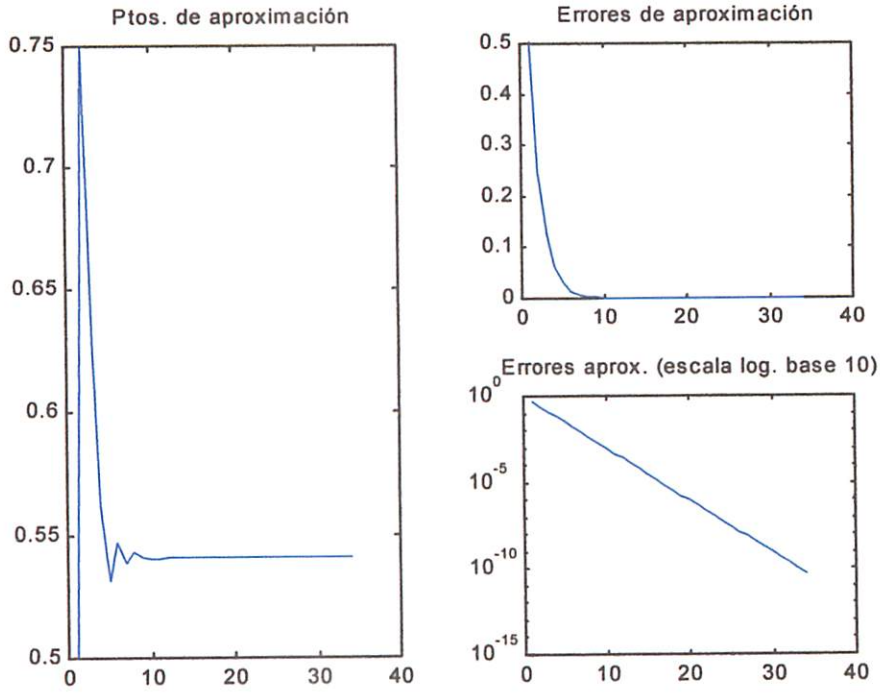
El algoritmo utilizado no solamente nos suministra la solución aproximada que deseamos, sino también toda la serie de puntos empleados durante la aproximación a dicha solución, así como toda la serie de errores cometidos mediante dicha serie de aproximaciones. Lógicamente el algoritmo de bisección hace uso del archivo que implementa la función original (**f.m**), ya que necesita conocer los valores de la misma para los sucesivos puntos de estudio.

Por lo cual el cálculo de la solución aproximada, así como la serie de puntos intermedios hasta llegar a la solución y sus errores asociados se calcularían con la siguiente llamada a la función bisección:

```
» [solucion,ptos,errs]=bisecion(0,1,10^-10)
```

Y a partir de los datos devueltos por dicha función es bastante sencillo la representación de la serie de convergencia hacia la solución, así como de los errores que se van cometiendo en dicha serie. La representación gráfica de las magnitudes anteriores se realizaría con un par de órdenes de la siguiente forma:

```
» a=size(ptos)
» subplot(121),plot(1:a(2),ptos),title('Ptos. de aproximación'),
  subplot(222),plot(1:a(2),errs),title('Errores de aproximación'),
  subplot(224),
   semilogy(1:a(2),errs),title('Errores aprox. (escala log. base 10)')
```



Implementación y resultados

```
function [S,p,err]=bisecion(x1,x2,toleran)
% [S,p,err]=biseccion(x1,x2,toleran)
%   calcula la serie de aproximaciones y la solución para f(x),
%   por el método de biseccion, entre los extremos (x1,x2)
%   y con un error máximo igual o menor a la tolerancia
if abs(f(x1)*f(x2))<toleran    % la solucion es x1 o x2
   if abs(f(x1))<toleran       % la solucion es x1
      S=x1; err=0; p=x1;
   else                        % la solucion es x2
      S=x2; err=0; p=x2;
   end
   break
elseif f(x1)*f(x2)>0     % no hay solucion
   disp('ERROR: No hay cambio de signo entre los extremos, o
         hay más de un cambio de signo.');
   break
else                        % si hay solucion
   n=round(log2((1/toleran)*abs(x2-x1))+3);
   % iteraciones necesarias para llegar a solucion
end

count=1;     S=(x1+x2)/2;       p(1)=S;  % punto inicial
err(1)=abs(x1-x2)/2;
% actualizacion de extremos
if (f(p(count))*f(x1))<0   x2=p(count);
else  x1=p(count); end

while (count<=n)&(err(count)>=toleran)
   count=count+1;
   p(count)=(x1+x2)/2;          % puntos siguientes
   err(count)=abs(x1-x2)/2;
   % actualizacion de extremos
   if (f(p(count))*f(x1))<0   x2=p(count);
   else  x1=p(count);    end
end
if err(count)>=toleran  % sobrepasa las iteraciones necesarias
   S=p(count-1);        % no se llega a la solucion deseada
else
   S=p(count);     % se ha llegado a la solucion deseada
end
```

En lugar de la utilización de diagrama de flujo, los cuales en alguna ocasión sobrepasaban la extensión del folio, me he decantado por la explicación literal del código (que se puede encontrar en el apartado de metodología) y la inclusión de comentarios autoexplicativos en el mismo código, lo cual se puede apreciar en el código, de la página anterior, del método de bisección.

Los resultados obtenidos para este método se presentan en la tabla siguiente, en la que cual se pueden observar que son necesaria **34** iteraciones para obtener la exactitud deseada (tolerancia=10^{-10}), también cabría destacar que el número de operaciones necesarias para alcanzar el resultado final es **931**.

nº iteración	Aproximación	Error máximo
1	0.50000000000000	0.50000000000000
4	0.56250000000000	0.06250000000000
7	0.53906250000000	0.00781250000000
10	0.54003906250000	0.00097656250000
13	0.54089355468750	0.00012207031250
16	0.54078674316406	0.00001525878906
19	0.54080009460449	0.00000190734863
22	0.54080176353455	0.00000023841858
25	0.54080179333687	0.00000002980232
28	0.54080179706216	0.00000000372529
31	0.54080179939046	0.00000000046566
34	0.54080179944867	0.00000000005821

· Método del Punto Fijo

Fundamento teórico

El método del punto fijo se basa en la búsqueda de una función "ligada" a la función original, de forma no unívoca, y trasladar la búsqueda de la solución de la función original, a la búsqueda de la intersección de la nueva función con la función identidad.

Este método no es tan genérico como el de bisección, ya que necesita de un punto inicial cercano a la solución, así como de la búsqueda de una función para lo cual no existe una metodología. Además aunque encontrar una función ligada a la original es fácil, no lo es tanto encontrar una que realmente sea útil, es decir, que realmente converge, y además rápido.

Asociado al método del punto fijo existe el denominado orden de convergencia, el cual nos sirve de medida de la bondad de la convergencia de la función hallada.

Otro método para ver la efectividad de la convergencia de las posibles funciones calculadas es el método gráfico, que será el que se empleará a continuación, y consiste en la representación de la derivada de la función utilizada por el método del punto fijo, y si los valores de esta en el intervalo de trabajo se mueven en intervalo [-1, 1] es que la función posee un nivel de convergencia aceptable.

Metodología

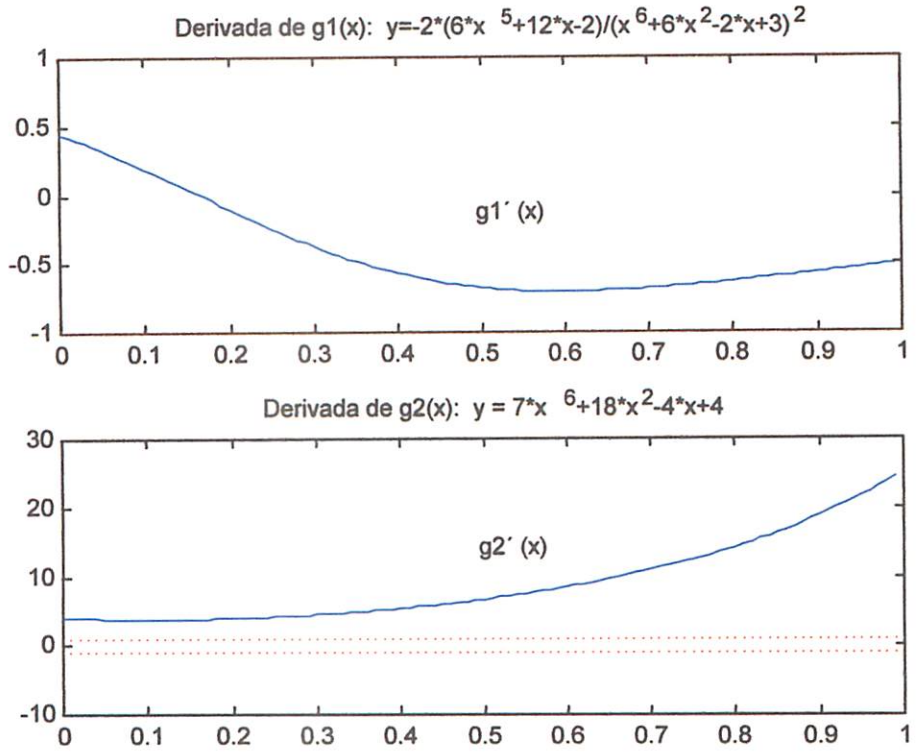
Se va a trabajar con dos posibles funciones de convergencia (**g1.m**, **g2.m**) para el método del punto fijo, como pueden ser por ejemplo:

$$g_1(x) = \frac{2}{x^6 + 6x^2 - 2x + 3} \qquad g_2(x) = x^7 + 6x^3 - 2x^2 + 4x - 2$$

Como he dicho anteriormente para comprobar la efectividad de la convergencia de dichas funciones, de recurrir al método gráfico representando las derivadas de ambas funciones en el intervalo de trabajo [0,1], y viendo si realmente se mueven en el intervalo [-1, 1].

Para ello seguiré la metodología empleada en el estudio de f(x), es decir, el cálculo de su derivada, la inclusión de ésta en un archivo (**g1prima.m**, **g2prima.m**), y finalmente la representación mediante un gráfico de características similares a las usadas en "*grf_f.m*":

$$g_1'(x) = \frac{-12x^5 - 24x + 4}{(x^6 + 6x^2 - 2x + 3)^2} \qquad g_2'(x) = 7x^6 + 18x^2 - 4x + 4$$



Como se puede observar, la función "g2" no es probable que converga nunca, en cambio, la función "g1" si que convergerá para valores próximos al intervalo de trabajo, y por lo tanto será la candidata para usarse en el método del punto fijo.

Por lo tanto, se creará un nuevo fichero (**g.m**) que implemente la función de convergencia que utilizará el método del punto fijo:

g.m

```
function y=g(x)
% g(x) calcula en el punto dado, el valor de
%  la funcion: y = 2/(x.^6 + 6*x.^2 - 2*x + 3)

y = 2/(x.^6 + 6*x.^2 - 2*x + 3);
```

La forma de trabajo de la algoritmo del punto fijo (**pto_fijo.m**) es la siguiente: inicialmente comprueba que el punto dado no es la solución buscada, después calcula el primer punto de la serie de aproximación a la solución final, y si éste punto no es aún la solución, entra en un bucle en el cual simplemente se continua calculando puntos de la serie de aproximación hasta que se cumplan los criterios de paro. En este caso los criterios de paro son múltiples, es decir, no se concluida el algoritmo hasta que la diferencia entre dos puntos de la serie consecutivos sea menor a una tolerancia predeterminada y el valor de la función para el último de estos puntos también sea menor a dicha tolerancia. La otra forma de concluir el algoritmo es que se haya sobrepasado el número de iteraciones máximas permitidas (es un parámetro de entrada de la función) lo que implica que no se ha llegado a solución esperada.

Por lo cual, el cálculo de la solución se reduce a una llamada de la forma:

```
» solucion=pto_fijo(0.4,10^-10,200)
```

Si en lugar de un solo parámetro de salida le facilitamos dos, en el segundo nos devolverá un vector cuyas componentes son las sucesivas aproximaciones hacia la solución final.

Implementación y resultados

pto_fijo.m

```
function [S,p]=pto_fijo(x0,toleran,N_max)
% [S,p]=pto_fijo(x0,toleran,N_max)
%  calcula la serie de aproximaciones y la solución para f(x),
%  por el método del punto fijo, para un valor inicial x0
%  y con un error máximo igual o menor a la tolerancia.
%     En caso de no convergencia, se llegará hasta la iteración N_max.
format long

if abs(f(x0))<=toleran      % comprueba si el pto inicial es solucion
   S=x0;
   p=x0;
   break
end

count=1;
p(count)=g1(x0);            % calcula la primera aproximacion
if abs(f(p(count)))<=toleran
   S=g1(x0);                % comprueba si el punto es solucion
   break
end

count=2;
p(count)=g1(p(count-1));    % calcula la segunda aproximacion

% terminacion cuando sobrepase el n° de iteraciones maximas, o
% si el la resta de dos ptos consecutivos y el valor de la funcion son
% menor que la tolerancia
while (count<=N_max)&...
      ((abs(p(count)-p(count-1))>=toleran)|...
      (abs(f(p(count)))>=toleran))
   count=count+1;
   p(count)=g1(p(count-1));    % calcula la n-esima aproximacion
end

S=p(count);                    % devuelve la solucion
```

Para obtener los resultados para este y los próximos métodos, se utilizarán siempre los mismos parámetros en las llamadas a los correspondientes métodos, con objeto de que las comparaciones entre ellos sean lo más correcta posible. Más concretamente, se presentan los resultados obtenidos para los siguientes parámetros:

- punto inicial $X = 0.4$
- tolerancia 10^{-10}
- nº máximo de iteraciones 200

En la tabla siguiente se puede observar que son necesaria **64** iteraciones para obtener el resultado con la tolerancia deseada, también es importe destacar y no aparece en la tabla que el número de operaciones necesarias para alcanzar el resultado es **824**.

nº iteración	Aproximación
1	0.63209207305973
4	0.51004821896238
7	0.55113212206588
10	0.53729631546403
13	0.54198857143123
16	0.54039965337237
19	0.54093802921391
22	0.54075564598020
25	0.54081743531358
28	0.54079650227558
31	0.54080359404092
34	0.54080119147496
37	0.54080200542297
40	0.54080172967147
43	0.54080182309132
46	0.54080179144229
49	0.54080180216443
52	0.54080179853196
55	0.54080179976258
58	0.54080179934566
61	0.54080179948691
64	0.54080179943906

· Método de Aiken

Fundamento teórico

El método de Aiken consiste básicamente en una forma para la reducción de errores en la sucesión de aproximación a la solución final, es decir, dicho método emplea los puntos hallados mediante, por ejemplo, el método del punto fijo, para a partir de éstos crear una nueva sucesión de puntos que se aproxima mucho más rápidamente a solución deseada. es decir el método de Aiken es capaz de conseguir una sucesión super lineal a partir de una sucesión lineal, lo cual se puede demostrar sencillamente mediante la resolución del siguiente límite:

Sea p_n una serie de puntos que convergen linealmente hacia la solución p, y sea p'_n la serie de Aiken que converge hacia la misma solución, entonces:

$$\lim \frac{|p'_n - p|}{|p_n - p|} = 0,$$

que nos indica que los errores cometidos por la serie de Aiken para un determinado número de iteraciones (n), es siempre mucho menor que el cometido por la serie de convergencia lineal para las mismas iteraciones.

· Construcción de la sucesión de Aiken:

Dada una sucesión p_n que convergen linealmente a la solución, entonces la sucesión superlineal de Aiken p'_n, está formada por los términos siguientes:

$$p'_n = p_n - \frac{(p_{n+1} - p_n)^2}{(p_{n+2} - 2p_{n+1} + p_n)}$$

Metodología

Puesto que el método de Aiken (**aiken.m**) necesita tres puntos de la sucesión lineal por cada nuevo punto de la sucesión superlineal, lo primero que hará la implementación de este método será calcular los tres primeros puntos de la sucesión lineal (mediante el método del punto fijo) y comprobar que ninguno de estos tres primeros puntos es la solución buscada.

Una vez hallados estos puntos, se introducirá en un bucle que irá calculando un nuevo punto de la sucesión lineal, así como un nuevo punto de la sucesión super líneal (usando el último punto calculado y los dos anteriores de la sucesión lineal), y así sucesivamente hasta que se cumpla alguno de los criterios de paro, que en este caso son: que la proximidad de dos puntos consecutivos sea menor que la tolerancia predeterminada o que el valor de la función original para cualquier punto sea menor que dicho tolerancia.

Como viene siendo habitual en la implementación de los métodos utilizados hasta ahora, este método también nos devuelve dos salidas, una la solución final y la otra la serie de puntos de aproximación a dicha solución final.

```
» [solucion,ptos]=aiken(0.4,10^-10,200)

solucion =      0.54080179938041
```

Implementación y resultados

Los resultados obtenidos con este método para los parámetros por defecto ($X_0 = 0.4$, tolerancia $= 10^{-10}$, y nº máximo de iteraciones 200) se presentan en la siguiente tabla:

nº iteración	Aproximación
1	0.54030451745985
4	0.54073318051667
7	0.54079463960645
10	0.54080095403048
13	0.54080170338971
16	0.54080178838878
19	0.54080179818294
22	0.54080179930555
23	0.54080179938041

Se puede observar que son necesaria **23** iteraciones para obtener el resultado con la tolerancia deseada, aunque es importe destacar que el número de operaciones necesarias para alcanzar el resultado no ha descendido demasiado, pues aún son necesarias **778**.

Como se ha comentado en el apartado de metodología, la implementación del algoritmo de Aiken sigue la tónica general utilizada en todos los algoritmos de esta práctica, es decir, comprobar inicialmente las posibles soluciones especiales (o soluciones no esperadas), para seguidamente sumergirse en el verdadero método de Aiken hasta encontrar la solución buscada.

También como viene siendo costumbre, utilizare un código suficientemente autocomentado, para evitar el uso de organigramas, ya que como he dicho anteriormente la extensión de los mismos sobrepasa a la del folio (seguramente por carecer de un software lo suficientemente adecuado para tal fin).

Aiken.m

```
function [S,p]=aiken(x0,toleran,N_max)
% [S,p]=aiken(x0,toleran,N_max)
%   calcula la serie de aproximaciones y la solución para f(x),
%   por el método superlineal de Aiken, utilizando para ello
%   la funcion de apoyo "punto fijo", para un valor inicial x0
%   y con un error máximo igual o menor a la tolerancia.
%      En caso de no convergencia, se llegará hasta la iteración N_max.

if abs(f(x0))<=toleran      % comprueba si el pto inicial es solucion
   S=x0; p=x0;
   break
end

count=1;
temp(count)=g(x0);       % calcula el 1° pto mediante "pto_fijo"
if abs(f(temp(count)))<=toleran      % comprueba si el pto es solucion
   S=temp(count);
   break
end

count=2;
temp(count)=g(temp(count-1)); % calcula el 2° pto mediante "pto_fijo"
if abs(f(temp(count)))<=toleran      % comprueba si el pto es solucion
   S=temp(count);
   break
end

count=3;
temp(count)=g(temp(count-1)); % calcula el 3° pto mediante "pto_fijo"
if abs(f(temp(count)))<=toleran      % comprueba si el pto es solucion
   S=temp(count);
   break
end

% calculo del primer punto de la sucesion de Aiken
p(1)=temp(1)-(((temp(2)-temp(1))^2)/(temp(3)-2*temp(2)+temp(1)));
if abs(f(p(1)))<=toleran       % comprueba si el pto es solucion
   S=p(1);
   break
end

for count=2:N_max
   temp(count+2)=g(temp(count+1));  % siguiente pto mediante "pto_fijo"
   % calculo del siguiente punto de la sucesion de Aiken
   p(count)=temp(count)-(((temp(count+1)-temp(count))^2)...
            /(temp(count+2)-2*temp(count+1)+temp(count)));

   % criterio de paro 1 = dos puntos sucesivos de Aiken menor que tolerancia
   if abs(p(count)-p(count-1))<toleran
      S=p(count);
      break
   end
   % criterio de paro 2 = valor de la funcion original menor que tolerancia
   if abs(f(p(count)))<toleran
      S=p(count);
      break
   end

end

S=p(count);          % devolucion de la solucion final
```

· Método de Newton-Raphson

Fundamento teórico

El método de Newton-Raphson es un método de conversión cuadrático, lo cual implica que a la velocidad de convergencia es mucho mayor que en los métodos lineales y super lineales.

· Teorema local de Newton-Raphson:

Sea $f \in C^2[a,b]$, si $p \in [a,b]$ es tal que $f(p)=0$ y $f'(p) \neq 0$, entoces existe $k > 0$ tal que el método de Newton converge cuadraticamente a la solución p para cualquier punto inicial $p_0 \in [p-k,p+k]$.

Lo cual implica que el método de Newton local es válido sólo para puntos iniciales cercanos a la solución.

Otra de las implicaciones de que sea un método cuadrático, es que la primera derivada de la función a estudiar deben ser siempre distinta de cero en el intervalo de trabajo, y esto es lógico, ya que el método de Newton se basa precisamente en la pendiente de la tangente a la curva original, es decir, la derivada, y no debe ser cero puesto que dicha derivada aparece como denominador de una fracción, además de porque los sucesivos puntos se van calculando como resultado de la intersección entre la tangente en el punto actual y el eje horizontal. Lógicamente si la derivada es igual a cero para cualquier punto del intervalo de estudio, esto significaría que la recta tangente es paralela al eje horizontal y por lo tanto nunca se cruzará con este último, ya que o son coincidentes (eje y tangente) o no tienen ningún punto en común.

· Construcción de la sucesión de Newton-Raphson:

Dado un punto inicial (p_0) cercano a la solución buscada, la sucesión p_n que converge hacia la solución, se construye entonces por los siguientes términos:

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \quad n \geq 1$$

Metodología

Como los métodos anteriores, el método de Newton-Raphson (**newton.m**) calcula una aproximación a la solución tan cercana como deseemos (determinado por el parámetro de entrada tolerancia) y la serie de aproximaciones a dicha solución.

El funcionamiento del algoritmo es el siguiente: comprueba que el punto de partida no es la solución deseada, después calcula los dos primeros puntos mediante la fórmula de aproximación de Newton, y si ninguno de estos puntos anteriores son la solución deseada, se introduce en un bucle que va calculando la serie de aproximaciones hasta alcanzar alguno de los criterios de paro, que como habitualmente son: que la diferencia de dos puntos sucesivos sea menor que la tolerancia deseada o que el valor de la función en algún punto sea menor que dicha tolerancia .

El motivo de calcular los dos primeros puntos fuera del bucle es bien sencillo, puesto que para controlar los criterios de paro es necesario comprobar los valores de los puntos actual y anterior, y dicha comprobación se realiza en la "entrada" al bucle, éstos valores deben haber sido calculados previamente.

Para finalizar este apartado, explicar la sintaxis de la llamada al método de Newton, que al igual que los métodos anteriores nos devuelve dos salidas, una con la solución final y la otra con toda la sucesión de puntos de aproximación a la solución.

```
» [solucion,ptos]=newton(0.4,10^-10,200)

solucion =    0.54080179945655
```

Implementación y resultados

Newton.m

```
function [S,p]=newton(x0,toleran,N_max)
% [S,p]=newton(x0,toleran,N_max)
%  calcula la serie de aproximaciones y la solución para f(x),
%  por el método de Newton-Raphson, para un valor inicial x0
%  y con un error máximo igual o menor a la tolerancia.
%  En caso de no convergencia, se llegará hasta la iteración N_max.
format long

if abs(f(x0))<=toleran      % comprueba si el pto inicial es solucion
   S=x0;
   p=x0;
   break
end

count=1;
p(count)=x0-f(x0)/fprima(x0);     % calcula el 1 pto mediante newton
if abs(f(p(count)))<=toleran      % comprueba si el pto es solucion
   S=p(count);
   break
end

count=2;                          % calcula el 2° pto mediante newton
p(count)=p(count-1)-f(p(count-1))/fprima(p(count-1));
if abs(f(p(count)))<=toleran      % comprueba si el pto es solucion
   S=p(count);
   break
end

% terminacion cuando sobrepase el n° de iteraciones maximas, o
% si la resta de dos ptos consecutivos es menor que la tolerancia, o
% se el valor de la funcion es menor que la tolerancia
while (count<N_max)&(abs(p(count)-p(count-1))>=toleran)...
       &(abs(f(p(count)))>=toleran)

   if fprima(p(count))==0      % comprueba si la derivada es nula
      break                    %   si lo fuese, finaliza la ejecuación
   end
   count=count+1;
   % calcula la n-esima aproximacion de Newton-Raphson
   p(count)=p(count-1)-f(p(count-1))/fprima(p(count-1));
end

S=p(count);                  % devuelve la solucion
```

Para la obtención de los resultados con el método de Newton-Raphson, y con objeto de realizar unas comparativas lo más objetivas posibles, se volverán a utilizar en la llamada a la función que implementa dicho método los parámetros de entrada que he denominado "por defecto", es decir, que como siempre se llamará a la función con los siguentes parámetros:

- punto inicial $X = 0.4$
- tolerancia 10^{-10}
- nº máximo de iteraciones 200

La tabla con los resultados obtenidos se puede encontrar en la página siguiente:

Tabla de resultados del método de Newton-Raphson:

nº iteración	Aproximación
1	0.57043803752061
2	0.54199418007371
3	0.54080376992256
4	0.54080179945655

Como se puede observar, este método tan solo ha necesitado **4** iteraciones para obtener el resultado final con la tolerancia deseada, también hay que destacar que el gasto computacional aha sido "mínimo", es decir, tan sólo han sido necesarias **168** operaciones para llegar a la solución, lo cual supone prácticamente una quinta o sexta parte del gasto computacional de los métodos anteriores.

· Método de Steffensen

Fundamento teórico

El método de Steffensen se basa en el método de Aiken y punto fijo, es decir, utiliza los tres primeros puntos hallados mediante el método de punto fijo, y a partir de ellos calcula un primer punto mediante el método de Aiken, a partir de este último punto calcula dos puntos nuevos mediante la aplicación del método del punto fijo al punto hallado, con estos tres nuevos puntos vuelve a usar Aiken para calcular un nuevo punto, para volver a aplicar el método del punto fijo y hallar otros dos nuevos puntos, y así sucesivamente.

· Teorema local de Steffensen:

Sea g una función de iteración de clase $C^3[a, b]$ que define un método lineal para calcular su punto fijo p, si $g'(p) \neq 1$, entonces la función g del algoritmo de Steffensen determina localmente un método de orden dos para calcular la solución p.

· Construcción de la sucesión de Steffensen:

$$p = p_0$$
$$x_n = g(p_n)$$
$$y_n = g(x_n)$$

$$p_{n+1} = p_n - \frac{(x_n - p_n)^2}{y_n - 2x_n + p_n}, \quad n \geq 0$$

Metodología

La implementación del método de Steffensen (**stefesen.m**) es posiblemente la más compleja de todas las de la práctica, ya que el trabajo con "índices" no sucesivos vuelve al código menos legible.

El funcionamiento del algoritmo es el siguiente:

Inicialmente calcula secuencialmente los tres primeros puntos mediante el método del punto fijo aplicado al punto inicial, si alguno de estos tres puntos fuese la solución buscada concluiría el algoritmo, sino calcula el cuarto punto (primer punto de la serie de Steffensen) mediante la fórmula de Steffensen, y comprueba si éste es la solución.

Sino sea alcanzado la solución, que hasta momento es el caso más normal, se entra en un bucle dentro del cual por cada iteración se calculan dos puntos mediante método del punto fijo aplicado sobre el punto anterior de la serie de Steffensen, y un nuevo punto de la serie de Steffensen a partir de los dos puntos anteriores. Las formas de conclusión del bucle son: haber llegado al máximo de iteraciones permitidas (pasado como parámetro), o cumplir alguno de los criterios de paro, que son: la diferencia entre dos puntos de la serie de Steffensen consecutivos menor que la tolerancia (pasada como parámetro) o el valor de la función en el último punto encontrado menor que la tolerancia .

Implementación y resultados

Stefesen.m

```
function [S,ptos]=steffensen(x0,toleran,N_max)
% [S,ptos]=steffensen(x0,toleran,N_max)
%   calcula la serie de aproximaciones y la solución para f(x),
%   por el método cuadrático de Steffensen, para un valor inicial x0
%   y con un error máximo igual o menor a la tolerancia.
%      En caso de no convergencia, se llegará hasta la iteración N_max.
format long

p(1)=x0;
if abs(f(x0))<=toleran          % comprueba si el pto inicial es solucion
   S=x0;
   p=x0;
   break
end

count=2; p(count)=g(x0);        % calcula el 2° pto mediante "pto_fijo"
if abs(f(p(count)))<=toleran    % comprueba si el pto es solucion
   S=p(count);
   break
end

count=3; p(count)=g(p(count-1)); % calcula el 3° pto mediante "pto_fijo"
if abs(f(p(count)))<=toleran    % comprueba si el pto es solucion
   S=p(count);
   break
end

count=4;
% calculo del primer punto de la sucesion de Steffense mediante Aiken
p(count)=p(1)-(p(2)-p(1))^2/(p(3)-2*p(2)+p(1));
ptos(1)=p(count);
if abs(f(p(count)))<toleran     % comprueba si el pto es solucion
   S=p(count);                  % mediante el valor de la funcion
   break
end
if abs(p(count)-p(count-3))<toleran % comprueba si el pto es solucion
   S=p(count);                      % por la resta de dos ptos consecutivos
   break
end

for count=5:3:N_max
   p(count)=g(p(count-1));      % siguiente pto mediante "pto_fijo"
   p(count+1)=g(p(count));      % y posterior pto mediante "pto_fijo"

   % calcula el siguiente punto de la sucesion de Steffensen mediante Aiken
   p(count+2)=p(count-1)-(p(count)-p(count-1))^2/...
              (p(count+1)-2*p(count)+p(count-1));

   ptos(fix(count/3)+1)=p(count+2); % ptos de la serie de Steffensen
   if abs(f(p(count+2)))<toleran    % comprueba si el pto es solucion
      S=p(count+2);                 % mediante el valor de la funcion
      break
   end
   if abs(p(count+2)-p(count-1))<toleran  % comprueba si es solucion
      S=p(count+2);                       % mediante dos ptos consecutivos
      break
   end
end
S=ptos(fix(count/3)+1);         % devuelve el último pto de Steffensen
```

Los resultados para el método de Steffensen se obtendrán , como en las ocasiones anteriores, utilizando en la llamada a dicho método los parámetros de entrada "por defecto", es decir, los siguentes parámetros:

- punto inicial X = 0.4
- tolerancia 10^{-10}
- nº máximo de iteraciones 200

```
» [solucion,ptos]=stefesen(0.4,10^-10,200)

solucion =    0.54080179945116
```

nº iteración	Aproximación
1	0.53896254853810
2	0.54080156743656
3	0.54080179945116

Se puede observar, que este método es el que menos iteraciones ha necesitado, concretamente sólo han sido **3** iteraciones para obtener la solución final con la tolerancia deseada. En cambio, el gasto computacional ha sido prácticamente identico al obtenido con el m´çetodo de Newton, es decir, tan sólo han sido necesarias **161** operaciones para llegar a la solución, frente a la 168 que necesita Newton.

· Análisis comparativo de los distintos métodos

En la siguiente tabla se resumen los resultados obtenidos para todos los métodos implementados en esta práctica, se han utilizado los mismos valores iniciales para todos los métodos excepto para el método de bisección en el cual el lugar de un punto inicial se ha utilizado el intervalo de estudio [0,1].

Además de los resultados parciales y finales obtenidos, se representa también una última fila que nos indica el gasto computacional que cada método necesita hasta llegar a la solución final deseada.

	Bisección	Pto. fijo	Aiken	Newton	Steffensen
0	[0 , 1]	0.4	0.4	0.4	0.4
1	0.5000000000000	0.6320920730597	0.5403045174598	0.5704380375206	0.5389625485381
2	0.7500000000000	0.4765492693570	0.5404802542190	0.5419941800737	0.5408015674365
3	0.6250000000000	0.5845886432772	0.5406800250788	0.5408037699225	0.5408017994511
4	0.5625000000000	0.5100482189623	0.5407331805166	0.5408017994565	----------
9	0.5410156250000	0.5458245092858	0.5408000907565	----------	----------
14	0.5408325195312	0.5399742453524	0.5408017525682	----------	----------
19	0.5408000946044	0.5409380292139	0.5408017981829	----------	----------
24	0.5408018231391	0.5407793700807	0.5408017993804	----------	----------
29	0.5408017989248	0.5408054922096	----------	----------	----------
34	0.5408017994486	0.5408011914749	----------	----------	----------
39	----------	0.5408018995483	----------	----------	----------
44	----------	0.5408017829711	----------	----------	----------
49	----------	0.5408018021644	----------	----------	----------
54	----------	0.5408017990044	----------	----------	----------
59	----------	0.5408017995247	----------	----------	----------
64	----------	0.5408017994390	----------	----------	----------
FLOPS	**931**	**824**	**778**	**168**	**161**

Análisis Numérico I :

Resolución de sistemas de ecuaciones no lineales

Alumno:
Fernanso Sanz Clavijo

Asignatura:
Análisis Numérico I

Práctica **2**:
Resolución de sistemas de ecuaciones no lineales

Indice

· Enunciado de la Práctica

La finalidad de esta práctica es encontrar una solución concreta al sistema de ecuaciones lineales formado por las siguientes ecuaciones:

$$\begin{cases} f_1(x,y) = x^3 + x^2 \cdot y + x \cdot y^7 - 2y - 1 = 0 \\ f_2(x,y) = x^4 \cdot y - x^6 + x \cdot y^5 - 3y + 2 = 0 \end{cases}$$

Más concretamente nos centraremos en la solución conocida de antemano (1,1), y para conseguir llegar a esta solución utilizaremos los siguientes métodos: Punto fijo, Seidel, Newton, y Broyden.

También se incrementará los algoritmo de Newton global y máximo descenso, aunque su finalidad no es posiblemente el obtener la solución final sino el acercarnos a ésta para posteriormente afinar más mediante alguno de los métodos anteriores.

Al igual que la práctica anterior la primera aproximación la realizaremos mediante un estudio gráfico, ya que la visualización del problema aclara mucho las diferencias entre trabajar con ecuaciones no lineales o con sistemas de ecuaciones no lineales.

· Estudio Gráfico

Fundamento teórico

El estudio de un sistema de ecuaciones por el método gráfico sólo resulta viable cuando trabajamos con sistemas de dos o tres incógnitas. Como en el caso que nos ocupa el sistema de ecuaciones es de dos incógnitas, la búsqueda de una solución se limita a la búsqueda de las intersecciones entre las dos curvas que representan cada una de las ecuaciones.

Como el resto del estudio lo vamos a centrar en el punto (1,1), la representación gráfica la realizaremos en torno a dicho punto, más concretamente iniciaremos la representación en la ventana formada entre X = [-5, 5], Y= [-5, 5], con lo cual observaremos que existen al menos cinco soluciones singulares del sistema de ecuaciones en la ventana de representación. Posteriormente reduciremos la zona de representación para poder observar fielmente que la solución buscada está en el punto (1, 1). Pero lógicamente el estudio gráfico que es simplemente aproximativo.

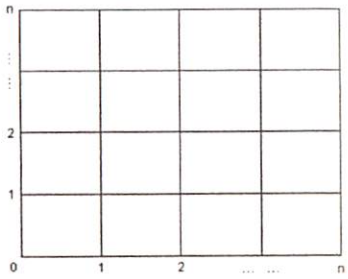
Metodología

Para implementar la representación gráfica he optado por dividir el funcionamiento del algoritmo en dos bloques bien diferenciados, inicialmente el usuario llama a un programa denominado **grafica.m**, al cual le pasa como parámetros la zona de representación (mediante el vértice superior izquierdo y el vértice inferior derecho), el número de cuadrículas en las que dividir la zona de representación (la zona de representación se divide en una matriz cuadrada, aunque la zona en sí no sea cuadrada sino rectangular, es decir, que el número de cuadrículas indica la longitud de las filas y columnas que se usa para dividir la zona), y el color de representación para la curva.

Desde este primer programa se divide la ejecución en dos subfunciones, la primera denominada **grf_xy.m** se encarga de buscar los valores de la función en sus cortes con las rejillas, y la segunda denominada **interpol.m** que se encarga de buscar todos los valores intermedios entre dos puntos consecutivos de corte con las rejillas. A continuación se describen ambas funciones en profundidad.

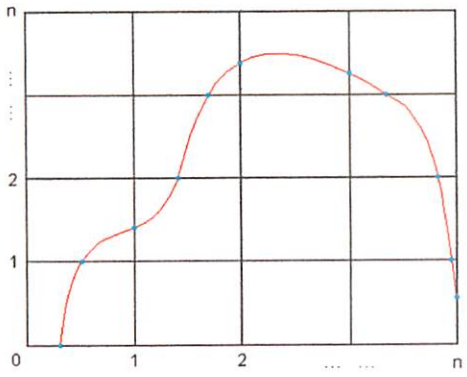
· **grf_xy.m**

La función **grf_xy.m** recibe los siguientes parámetros de entrada desde la función de interfaz **grafica.m:** el tamaño de la ventana de representación, el número de divisiones para esta ventana (número de cuadrículas por fila o columna) así como la tolerancia y el número máximo de iteraciones a emplear en la búsqueda de los puntos de la curva.



El funcionamiento del algoritmo a grosso modo es el siguiente:

· divide la zona de representación en **n** partes iguales tanto horizontal como verticalmente, lo cual gráficamente sería equivalente a una rejilla como la representada en la figura adjunta.



· una vez obtenidos las **n** rectas horizontales y las **n** rectas verticales procedemos a buscar las intersecciones entre estas rectas y la curva a representar, con lo cual obtenemos una serie puntos que nos dan una idea aproximada de como va a ser la curva final.

La implementación de la función **grf_xy.m** realmente es la siguiente:

Se crea un vector X que contiene los **n** puntos que dividen a la amplitud de representación horizontal en **n** partes iguales. Se crea el vector equivalente Y que contiene los **n** puntos que dividen a la amplitud de representación vertical en **n** partes iguales.

Inicialmente se recorre la zona de representación de izquierda a derecha, suponiendo a la primera variable "x" como constante e igual al primer valor del vector X, y buscando una solución de la ecuación no lineal resultante (ecuación en "y"). Para ello se irá probando con varios puntos iniciales, los contenidos en el vector Y, ya que se utilizará el método de Newton local en una variable, y éste es un método local.

Una vez obtenido el primer punto, se continúa buscando desde el último valor del vector X utilizado, hasta encontrar todas las posibles soluciones locales para los diferentes valores del vector Y.

Este proceso se repite para todos los valores del vector X, con lo cual obtenemos todos los puntos [1] de corte de la curva a representar con las rectas verticales que representa el vector X .

Exactamente el mismo procedimiento empleado con X se realiza también con el vector Y, en esta ocasión se recorre la zona de representación desde abajo hacia arriba considerando los valores de "y" constantes, resultando una ecuación no lineal en "x", y obteniendo así todos los puntos de corte de la curva con las rectas horizontales que representa el vector Y.

Una vez obtenidos todos los puntos de corte tanto con las rectas horizontales como verticales se reúnen ambos en un conjunto, y se almacena el conjunto resultante junto con alguna información adicional en un fichero de datos denominado "temp.mat", finalmente se representan gráficamente todos los puntos obtenidos, lo que da como resultado una especie de esqueleto de la ecuación a representar.

---

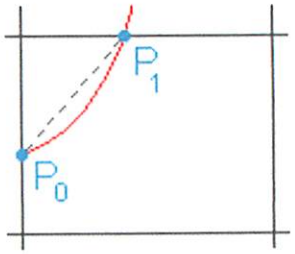
[1] como los puntos obtenidos son para una representación gráfica en pantalla no se considerarán como puntos de corte aquellos que estén extremadamente cerca de otro punto anterior, es decir, aquellos nuevos puntos de intersección cuya distancia a otro punto ya obtenido sea menor que la tolerancia prefijada no serán incluidos como puntos de intersección.

· **interpol.m**

El comportamiento de la función de interpolación de puntos es bastante más complejo que el de la función anterior, ya que aunque para una persona sea muy fácil intuir la forma de la curva a partir del esqueleto, la traducción analítica de este problema no es tan sencilla.

Los datos con los que comienza la función **interpol.m** son proporcionados por el fichero creado anteriormente por la función **grf_xy.m** (temp.mat).

Una primera aproximación a la forma de trabajo de la función "interpoladora" podría ser la siguiente:



Coger un punto inicial P_0 cualquiera del conjunto de puntos del esqueleto [2] y buscar en el resto de puntos del esqueleto el más cercano al punto inicial [3]. Una vez encontrada se traza imaginariamente la recta que une ambos puntos, y se recorre dicha recta considerando alguna de las variables ("x" o "y") como constante, y buscando la solución de la ecuación no lineal en una variable resultante. Mediante esta técnica se consiguen los puntos intermedios de la curva que hay entre dos puntos consecutivos del esqueleto.

Si se repite esta fórmula desde el punto P_1 hasta el siguiente y así sucesivamente se consigue recorrer alguna de las ramas de la gráfica de la función, ya que para evitar que se interconecten ramas que no están directamente conectadas, antes de buscar los puntos intermedios entre los dos puntos del esqueleto P_0 y P_1 se comprueba que realmente: además de ser el más cercano el uno al otro, también la distancia que los separa debe ser pequeña [4].

Antes de empezar a recorrer alguna otra rama de la gráfica hay que comprobar que no se han recorrido ya todos los puntos, ya que si esto fuese así podría suponer dos cosas: o que en la ventana de representación existe una sola rama de la función, o que la curva es cerrada y además esta contenida completamente en la ventana de representación. Si se diese el segundo caso habría que reinsertar el punto inicial en el conjunto de puntos aún no recorridos, ya que sino la curva no se terminaría de cerrar.

En cambio si el conjunto de puntos aún no ha sido recorrido se procede usando como distintos puntos iniciales los extremos de más arriba, más abajo y más a la derecha del conjunto de puntos del esqueleto que aún queden por recorrer, y se repite este proceso hasta que el conjunto quede vacío.

La implementación de la función **interpol.m** se basa principalmente en el trabajo con conjuntos, y más concretamente su funcionamiento es el siguiente:

Inicialmente cargamos el conjunto de puntos suministrados por la función **grf_xy.m**, los consideramos como un conjunto de puntos no recorridos, y creamos dos parámetros denominados tolerancia y tolerancia gráfica que serán utilizados para determinar cuándo dos puntos del esqueleto son realmente cercanos, y para determinar el número de puntos que se deben intercalar[5] entre dos puntos consecutivos del esqueleto, respectivamente.

Después se calcularán los puntos extremos del esqueleto, es decir se buscarán los puntos más a la derecha, a la izquierda, arriba y abajo del todo el conjunto.

Ahora, y comenzando desde el extremo izquierdo se realizan iterativamente las siguientes acciones:

· representamos el punto,
· lo extraemos del conjunto de puntos sin recorrer,
· calculamos el punto más cercano [3], y
· comprobamos que la distancia que los separa es menor que la tolerancia.

---

[2] por convenio el de más a la izquierda

[3] mediante la función auxiliar **cercano.m**

[4] por defecto se considerará que dos puntos son realmente cercanos cuando la distancia que los separan es menor que la longitud de la diagonal de una de las cuadrícula de la rejilla usada en la función **grf_xy.m**

[5] el número esperado de puntos intermedios entre dos puntos consecutivos del esqueleto viene determinado por el producto del parámetro tolerancia gráfica y la distancia que separa a estos dos puntos.

Si no existiese ningún otro punto en el conjunto que cumpliese ambos requisitos, se pasaría directamente a reintentar todas las operaciones desde alguno de los otros tres extremos. En cambio si sí se encontrase algún punto en el que se cumplieran los requisitos:

- se extraería también éste del conjunto de puntos por recorrer
- se representaría en pantalla,
- después se calcula el número de puntos a intercalar entre estos dos puntos cercanos,

Se comprueba si la distancia recorrida entre el punto inicial y el siguiente es mayor horizontal ó verticalmente. Si la distancia recorrida en horizontal es mayor que la recorrida en dirección vertical, se calcularán los puntos intermedios reduciendo el sistema de ecuaciones a una ecuación sólo en "y", obteniendo los distintos valores de "x" constantes como los **n** puntos equidistante que separan a los puntos extremos a unir. Todos los puntos internos encontrados que además no estén demasiado cercanos el uno del otro, se incluirán en un conjunto acumulador [6] de puntos que constituirán la gráfica final y se representarán en pantalla.

Una vez concluido se considerará como punto inicial al que antes se consideraba como punto siguiente, y se vuelve a calcular el punto más cercano a éste, y así sucesivamente hasta el momento en el cual no encontremos ningún punto que cumpla el requisito de distancia para ser realmente cercano.

Cuando finalmente no se encuentra ningún punto que esté realmente cerca, esto puede ser debido o a que ya hemos recorrido toda la gráfica [7], o a que existen más ramas desconectadas de la rama anterior.

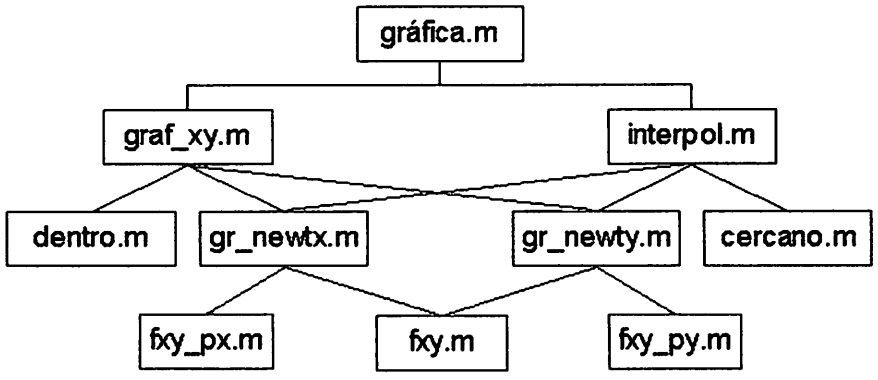
Por lo tanto se comprueba si el resto de puntos extremos [8] del esqueleto, han sido ya recorridos y si no es así, se repite todo el proceso realizado para el extremo izquierdo con el resto de los extremos, con lo cual se terminará recorriendo todas las ramas posibles dentro de la zona de visualización.

Si una vez concluido el proceso para los cuatro extremos aún quedan puntos en el conjunto de puntos por recorrer, se recalculan tres de los extremos[7] de lo que queda del esqueleto, y se repiten todos los cálculos desde el inicio, hasta que finalmente el conjunto de puntos por recorrer queda realmente vacío, es decir, ya sea han recorrido todos los puntos de la gráfica.

Finalmente habremos obtenido toda la gráfica completa dentro de la zona de visualización, y se grabarán todos los puntos calculados en un fichero [9] denominado "_fig.mat"

Implementación y resultados

Los algoritmos necesarios para la representación gráfica son muchos, por lo cual comenzaremos con la presentación de un árbol que esquematiza la comunicación entre las distintas funciones:



[6] para evitar desbordamientos, los puntos intermedios realmente se van almacenando en un conjunto temporal de tamaño predefinido por el número de puntos esperados inicialmente calculado, y finalmente se realizará la unión de los conjuntos "puntos intermedios" y el conjunto "acumulador de puntos"

[7] sólo si la gráfica tiene una sola rama continua en la ventana de visualización

[8] superior, derecho e inferior

[9] para su posterior representación si fuese necesaria

El orden de los siguientes códigos vendrá dado por una serie de capas que nos lleven desde los niveles más altos, la "gráfica.m", hasta los más bajos como son los cálculos de: las funciones de $R_N \rightarrow R$, las derivadas de éstas respecto a "x" y respecto a "y", los métodos de Newton en una variable ("x" e "y"), el punto más cercano a uno dado, etc.

grafica.m

```
function grafica(inicio_xy,final_xy,num_cuadriculas,color)
%
% SINTAXIS: grafica(inicio_xy,final_xy,num_cuadriculas,color)
%
%   inicio_xy   =   vertice superior-izquierdo de la ventana de visualizacion
%   final_xy    =   vertice inferior-derecho de la ventana de visualizacion
%   num_cuadr.  =   nº de divisiones de la ventana de visualizacion,
%                   tanto verticalmente como horizontalmente
%   color   = color usado para dibujar la curva (con el mismo formato que plot)

    clf;      % borra la ventana gráfica de MATLAB

    %  Calcula la mínima amplitud de la zona de representación
       % amplitud horizontal
       amp=abs(inicio_xy(1)-final_xy(1));
       % minimo de ambas amplitudes
       amp=min(amp,abs(inicio_xy(2)-final_xy(2)));

    TOL=amp/1000;   % TOLerancia a usar en la busqueda de ptos de la gráfica

   % iteraciones usadas para la busqueda mediante Newton en una variable
    N_max_iteraciones=5*fix(log10(1/TOL));


%   graf_xy dibuja el "esqueleto" de la función contenida en "fxy"
%       también necesita las derivadas parciales "fxy_px" y "fxy_py"
graf_xy(inicio_xy,final_xy,TOL,N_max_iteraciones,num_cuadriculas,color);

%   "interpol.m" busca los puntos intermedios entre los puntos del "esqueleto"
interpol(color);
```

graf_xy.m (1)

```
function conjunto=graf_xy(inicio,final,TOL,N_max,num_cuadriculas,color)
%   Dibuja la funcion f(x,y) en el rectangulo formado por los vertices
%       opuestos inicio(sup-izq) y final(inf-der)
% SINTAXIS:
%  ptos=graf_xy(inicio_xy,final_xy,TOL,N_max_iteraciones,num_cuadriculas,color)

temp=zeros(1,2);  % inicializa un punto temporal de dos coordenadas

% calcula las amplitudes de la zona de representación
ampli_x=final(1)-inicio(1);    % amplitud horizontal
ampli_y=inicio(2)-final(2);    % amplitud vertical

if ampli_x<=0 | ampli_y<=0     % si alguna de las amplitudes es negativa
   disp('Vertices desordenados');
   break;
end

% Sectorizacion de la zona de representacion en "num_" de cuadriculas
rejilla_x=inicio(1):ampli_x/num_cuadriculas:final(1);
rejilla_y=final(2):ampli_y/num_cuadriculas:inicio(2);
% Numero de ejes horizontales (debe coincidir con el de ejes verticales)
tam_rejilla_y=size(rejilla_y);
tam_rejilla_x=size(rejilla_x);
........
```

graf_xy.m (2)

```
% mini-barrido en OX
%   Puntos de corte de la funcion con los ejes verticales de la cuadricula

n_ptos=0;    % nº de ptos de corte encontrados (con ejes verticales), ninguno
ok=0; n=1;   % comenzar por el eje vertical mas a la izquierda,
m=1;         % y el horizontal mas abajo

% repetir hasta que no quede ningun eje vertical,o se haya encontrado el 1º pto
while n<=tam_rejilla_x(2) & n_ptos==0
   % repetir hasta que no quede ningun eje horizontal,o se encuentre el 1º pto
    while m<=tam_rejilla_y(2) & n_ptos==0
        % hasta que no quede ningun eje horizontal,o se encuentre una solucion
        while ok==0 & m<=tam_rejilla_y(2)
         % busca una solucion para la ec. en una variable resultante si X =cte.
         [temp,ok]=gr_newty([rejilla_x(n) rejilla_y(m)],TOL,N_max);
         m=m+1;
        end
        if ok==1     % si se ha encontrado alguna solucion (pto de la curva)
           %  se comprueba que esta dentro de la zona de representacion
           if dentro(temp,inicio,final)==1
              % si lo esta, se añade al conjunto de ptos de corte "verticales"
              ptos_gridX=temp;
              n_ptos=1;
           end
           ok=0;
        end
    end
    %se ha probado con todos los ejes horizontales,o se ha encontrado el 1º pto
    n=n+1;       % se pasa al siguiente eje vertical,
    m=1;         % y se repite con todos los horizontales
end
% ya se ha probado con todos los ejes verticales, o se ha encontrado el 1º pto

n=n-1; %se continua con el anterior eje vertical,por si hay mas ptos de corte
while n<=tam_rejilla_x(2)      % hasta que no quede ningun eje vertical...
    while m<=tam_rejilla_y(2)  % hasta que no quede ninguno horizontal...
        % hasta que no quede ningun eje horizontal o se encuentre algun punto...
        while ok==0 & m<=tam_rejilla_y(2)
           % buscar una solucion para la ec. resultante si X es constante
           [temp,ok]=gr_newty([rejilla_x(n) rejilla_y(m)],TOL,N_max);
           m=m+1;
        end
        if ok==1     % si se ha encontrado alguna solucion (pto de la curva)
           %  se comprueba que esta dentro de la zona de representacion
           if dentro(temp,inicio,final)==1
              min_dist=inf;
              %calcula la separacion del pto mas cercano de los ya encontrados
              for k=1:n_ptos
                 min_dist=min(norm(temp-ptos_gridX(k,:)),min_dist);
              end
              if min_dist>TOL    % si dicha distancia no es muy pequeña
                 % se añade al conjunto de ptos de corte "verticales"
                 ptos_gridX=union(ptos_gridX,temp,'rows');
                 n_ptos=n_ptos+1;
              end
           end
           ok=0;
        end
    end
   % se ha probado con todos los ejes horizontales (para uno determinado vertical)
    n=n+1;       % se pasa al siguiente eje vertical,
    m=1;         % y se repite con todos los horizontales
end
% se ha probado ya con todos los ejes verticales
.............
```

graf_xy.m (3)

```
% mini-barrido en OY
%   Puntos de corte de la funcion con los ejes horizontales de la cuadricula

n_ptos=0;    % nº de ptos de corte encontrados(con ejes horizontales),ninguno
ok=0; m=1;   % comenzar por el eje horizontal mas abajo,
n=1;         % y el vertical mas a la izquierda

% hasta que no quede ningun eje horizontal,o se haya encontrado el 1º pto
while m<=tam_rejilla_y(2) & n_ptos==0
   % hasta que no quede ningun eje vertical,o se encuentre el 1º pto
   while n<=tam_rejilla_x(2) & n_ptos==0
      % hasta que no quede ningun eje vertical,o se encuentre una solucion
      while ok==0 & n<=tam_rejilla_x(2)
       % busca una solucion para la ec. en una variable resultante si Y=cte.
       [temp,ok]=gr_newtx([rejilla_x(n) rejilla_y(m)],TOL,N_max);
       n=n+1;
      end
      if ok==1     % si se ha encontrado alguna solucion (pto de la curva)
         %  se comprueba que esta dentro de la zona de representacion
         if dentro(temp,inicio,final)==1
            %si lo esta,se añade al conjunto de ptos de corte "horizontales"
            ptos_gridY=temp;
            n_ptos=1;
         end
         ok=0;
      end
   end
   % se ha probado con todos los ejes verticales,o se ha encontrado el 1º pto
   m=m+1;       % se pasa al siguiente eje horizontal,
   n=1;         % y se repite con todos los verticales
end
% ya se ha probado con todos los ejes horizontales,o se ha encontrado el 1º pto

m=m-1; %se continua con el ultimo eje horizontal,por si hay mas ptos de corte
while m<=tam_rejilla_y(2)  % hasta que no quede ningun eje horizontal...
   while n<=tam_rejilla_x(2)  % hasta que no quede ninguno vertical...
      % hasta que no quede ningun eje vertical o se encuentre algun punto
      while ok==0 & n<=tam_rejilla_x(2)
         % buscar una solucion para la ec. resultante si Y es constante
         [temp,ok]=gr_newtx([rejilla_x(n) rejilla_y(m)],TOL,N_max);
         n=n+1;
      end
      if ok==1     % si se ha encontrado alguna solucion (pto de la curva)
         %  se comprueba que esta dentro de la zona de representacion
         if dentro(temp,inicio,final)==1
            min_dist=inf;
            %calcula la separacion del pto mas cercano de los ya encontrados
            for k=1:n_ptos
               min_dist=min(norm(temp-ptos_gridY(k,:)),min_dist);
            end
            if min_dist>TOL    % si dicha distancia no es muy pequeña
               % se añade al conjunto de ptos de corte "horizontales"
               ptos_gridY=union(ptos_gridY,temp,'rows');
               n_ptos=n_ptos+1;
            end
         end
         ok=0;
      end
   end
   % se ha probado con todos los ejes verticales (para uno determinado horizontal)
   m=m+1;       % se pasa al siguiente eje horizontal,
   n=1;         % y se repite con todos los verticales
end
% se ha probado ya con todos los ejes horizontales
.............
```

graf_xy.m (4)

```
.............
% Preparacion de los datos para ser grabados
   %  Creacion del "Conjunto total" de los puntos encontrados
   conjunto=union(ptos_gridX,ptos_gridY,'rows');
   tam_conj=size(conjunto);          % tamaño de dicho conjunto

   %  Caracteristicas de la rejilla
      % tamaño de la base de una cuadricula
   base_celda=abs(rejilla_x(1)-rejilla_x(2));
      % tamaño de la altura de una cuadricula
   altura_celda=abs(rejilla_y(1)-rejilla_y(2));

% Guarda los datos para la funcion "interpol.m",en el archivo "temp.mat"
save('temp.mat','conjunto','tam_conj'...
      ,'base_celda','altura_celda','num_cuadriculas'...
      ,'inicio','final','N_max');

% Representa todos los puntos de corte (esqueleto de la curva)
      % ejes principales
axis([inicio(1) final(1) final(2) inicio(2)]), hold on; box on;
for k=1:tam_conj(1)
   % puntos de corte con OX y OY
   plot(conjunto(k,1),conjunto(k,2),color), hold on;
end
```

interpol.m (1)

```
function interpol(color)
%   Dibuja la grafica por interpolación, usando para ello
%       los puntos suministrados por el fichero de "temp.mat"

load temp.mat;          % carga el fichero de datos creado por "graf_xy.m"

% TOLerancia para: la cercania de ptos y las llamadas a Newton en una variable
TOL=sqrt(base_celda^2+altura_celda^2);
% TOL_grf = proporcionalidad usada en el calculo de ptos intermedios esperados
TOL_grf=100/TOL;

conj_temp=conjunto;   % copia del conjunto de ptos del esqueleto

%   Calculo de los 4 puntos extremos (derecho, izquierdo, superior, inferior)
p_minx=conjunto(1,:);                % extremo izquierdo
p_maxx=conjunto(tam_conj(1),:);      % extremo derecho
conjunto=conjunto(:,[2 1]);       % reordenacion del conjunto para
conjunto=unique(conjunto,'rows');%   obtener los extremos en Y
p_miny=conjunto(1,[2 1]);            % extremo inferior
p_maxy=conjunto(tam_conj(1),[2 1]); % extremo superior

% Restablece el conjunto original y libera memoria
conjunto=conj_temp;          % restablece copia de seguridad
clear conj_temp;             % destruye la copia

%  Acumulador de todos los puntos
AC_ptos=conjunto;            % almacenará todos los puntos de la grafica

%  Inicializacion de la ventana grafica
axis([inicio(1) final(1) final(2) inicio(2)]), hold on;
box on;
.........
```

interpol.m (2)

```
% Buscar el punto mas cercano y calcular los intermedios, comenzando por....
%  1° por el extremo izquierdo
p0=p_minx;                          % pto de inicio = extremo izquierdo
   % quita el pto de inicio de los ptos no recorridos
conjunto=setdiff(conjunto,p0,'rows');
plot(p0(1),p0(2),color), hold on;       % representa el pto en pantalla
[p1,d01]=cercano(p0,conjunto);       % busca el pto mas cercano al inicial
while d01<=TOL     % mientras existan ptos cercanos sin recorrer...
   plot(p1(1),p1(2),color), hold on;    %presenta pto mas cercano en pantalla
      % y lo quita del conjunto de puntos no recorridos
   conjunto=setdiff(conjunto,p1,'rows');
   Ax=p1(1,1)-p0(1,1);                  % calcula la recta que une
   Ay=p1(1,2)-p0(1,2);                  % los dos ptos consecutivos
   n_ptos=round((d01*TOL_grf)+2);    % n° de ptos intermedios esperados segun
   ptos_int=zeros(n_ptos,2);         %    la distancia que los separa
   if abs(Ax)>abs(Ay)    % si el desplazamiento horizontal > el vertical
      m=0;     % n° de ptos intercalados (inicialmente ninguno)
      for n=1:n_ptos
         % busca los ptos intermedios considerando a X como constante
         [temp2,ok]=gr_newty(p0+[n*Ax/n_ptos n*Ay/n_ptos],TOL,N_max);
         % si encuentra algun punto suficientemente cercano...
         if ok==1 & norm(temp2-p1,inf)<TOL
            m=m+1;                  % actualiza el n° de ptos encontrados
            ptos_int(m,:)=temp2; % y lo almacena en un conjunto temporal
         end
      end
   else      % si el desplazamiento vertical es mayor que el horizontal
      m=0;     % n° de ptos intercalados (inicialmente ninguno)
      for n=1:n_ptos
         % busca los ptos intermedios considerando a Y como constante
         [temp2,ok]=gr_newtx(p0+[n*Ax/n_ptos n*Ay/n_ptos],TOL,N_max);
         % si encuentra algun punto suficientemente cercano...
         if ok==1 & norm(temp2-p1,inf)<TOL
            m=m+1;                  % actualiza el n° de ptos encontrados
            ptos_int(m,:)=temp2; % y lo almacena en un conjunto temporal
         end
      end
   end
   if n>m      % si no se encontraron todos los ptos inicialmente esperados
      temp3=ptos_int(1:m-1,:);    %quita la posiciones no usadas(si reservadas)
      ptos_int=0;
      ptos_int=temp3;       % y recorta el conjunto al tamaño justo necesario
      temp3=0;
   end
   % añade los ptos encontrados al acumulador y los representa en pantalla
   AC_ptos=union(AC_ptos,ptos_int,'rows');
   plot(ptos_int(:,1),ptos_int(:,2),color), hold on;
   p0=p1;             % reemplaza el pto origen por el anterior pto destino
   [p1,d01]=cercano(p0,conjunto);    % busca el pto cercano a este ultimo
   tam_conjunto=size(conjunto);  % si tras el recorrido de la 1ª rama
   if tam_conjunto(1)==1       %  no quedan mas ptos en el conjunto, puede ser
     conjunto=union(conjunto,p_minx,'rows'); % ..una curva cerrada, asi que
   end;                          % ..re-introduzco el punto inicial "p_minx".
end

% Union de los tres extremos restantes del esqueleto
extremos=union(p_maxy,p_miny,'rows');  % extremos = superior e inferior
extremos=union(extremos,p_maxx,'rows');%extremos superior,inferior y derecho

tam_conj=size(conjunto);       % tamaño de lo que queda de esqueleto
..............
```

interpol.m (3)

```
k=1;                                % comenzamos por extremos(k=1)=superior
% Posibles segundo,tercer... barrido, para casos de numerosas ramas de la funcion
while tam_conj(1)~=0      % repetir hasta que no haya ptos por recorrer...
   % interseccion, para comprobar que el extremo sigue entre los ptos no recorridos
   interseccion=intersect(conjunto,extremos(k,:),'rows');
   tam_inter=size(interseccion);
   if tam_conj(1)~=0     % comprueba que el conjuto de ptos por recorrer no vacio
      if tam_inter(1)~=0    % comprueba que el extremo no ha sido ya recorrido
         p0=extremos(k,:); % considera a dicho extremo como pto origen
            % y lo quita del conjunto de puntos no recorridos
         conjunto=setdiff(conjunto,p0,'rows');
            % presenta el pto origen en pantalla
         plot(p0(1),p0(2),color), hold on;
         [p1,d01]=cercano(p0,conjunto);% busca el pto mas cercano al mismo
         while d01<=TOL     % mientras existan ptos cercanos sin recorrer...
               % representa el pto mas cercano en pantalla
            plot(p1(1),p1(2),color), hold on;
               % y lo quita del conjunto de puntos no recorridos
            conjunto=setdiff(conjunto,p1,'rows');
            Ax=p1(1,1)-p0(1,1);              % calcula la recta que une
            Ay=p1(1,2)-p0(1,2);              % los dos ptos consecutivos
            n_ptos=round((d01*TOL_grf)+2);%  n° de ptos esperados segun
            ptos_int=zeros(n_ptos,2);        %  la distancia que los separa
            if abs(Ax)>abs(Ay)    % si el desplazamiento horizontal > vertical
               m=0;    % n° de ptos intercalados (inicialmente ninguno)
               for n=1:n_ptos
                  % busca los ptos intermedios considerando a X constante
                  [temp2,ok]=gr_newty(p0+[n*Ax/n_ptos n*Ay/n_ptos],TOL,N_max);
                  % si encuentra algun punto suficientemente cercano...
                  if ok==1 & norm(temp2-p1,inf)<TOL
                     m=m+1;              % actualiza el n° de ptos encontrados
                     ptos_int(m,:)=temp2; % lo almacena en un conjunto temp
                  end
               end
            else         % si el desplazamiento vertical > el horizontal
               m=0;       % n° de ptos intercalados (inicialmente ninguno)
               for n=1:n_ptos
                  % busca los ptos intermedios considerando a Y constante
                  [temp2,ok]=gr_newtx(p0+[n*Ax/n_ptos n*Ay/n_ptos],TOL,N_max);
                  % si encuentra algun punto suficientemente cercano...
                  if ok==1 & norm(temp2-p1,inf)<TOL
                     m=m+1;              % actualiza el n° de ptos encontrados
                     ptos_int(m,:)=temp2; % lo almacena en un conjunto temp
                  end
               end
            end
            if n>m       % si no se encuentran todos los ptos esperados
               temp3=ptos_int(1:m-1,:);    % quita la posiciones no usadas
               ptos_int=temp3;             % y recorta al tamaño necesario
            end
            % añade los ptos encontrados al acumulador y los representa en pantalla
            AC_ptos=union(AC_ptos,ptos_int,'rows');
            plot(ptos_int(:,1),ptos_int(:,2),color), hold on;
            p0=p1;    % reemplaza el pto origen por el anterior pto destino
            [p1,d01]=cercano(p0,conjunto);    % busca el pto cercano a éste
      end
   end

   k=k+1;          % seguimos con extremos(k=2,3)=inferior y derecho
   %  y actualizamos el tamaño del conjunto de ptos por recorrer
   tam_conj=size(conjunto);
.............
```

interpol.m (4)

```
   % Comprueba que realmente no hemos recorrido todos los ptos, pero si los extremos
   if tam_conj(1)~=0 & k>3
      conj_temp=conjunto;      % copia del conjunto de ptos del esqueleto
      % re-calcula los 3 puntos extremos (derecho, superior, inferior)
      p_maxx=conjunto(tam_conj(1),:);      % extremo derecho
      conjunto=conjunto(:,[2 1]);          % reordenacion del conjunto para
      conjunto=unique(conjunto,'rows');    %   obtener los extremos en Y
      p_miny=conjunto(1,[2 1]);            % extremo inferior
      p_maxy=conjunto(tam_conj(1),[2 1]); % extremo superior
      % restablece el conjunto original y libera memoria
      conjunto=conj_temp;                  % restablece copia de seguridad
      clear conj_temp;                     % destruye la copia
      k=1;                    % y comienza otra vez desde el primer extremo
   end

end            %%    .... (del "while" de multiples ramas)  %%


save('_fig.mat','AC_ptos');    % guarda los ptos para usos posteriores
```

cercano.m

```
function [p,dist_min]=cercano(pto,conjunto)
% Busca en un conjunto, el pto mas cercano al dado, y la distancia que los separa
%      [p,dist_min]=cercano(pto,conjunto)

dist_min=inf;
tam_conj=size(conjunto);    % tamaño del conjunto de puntos
if tam_conj(1)==0           % si el conjunto esta vacio no hay "cercano"
   p=[];
   break
end

for n=1:tam_conj(1)         % si no esta vacio...
   % calcula la distancia de cada pto del conjunto al pto dado
   temp=min(dist_min,norm(pto-conjunto(n,:)));
   if temp<dist_min  % si es menor que la minima,la considera como nueva minina
      dist_min=temp;
      p=conjunto(n,:);      % y selecciona al pto como el mas cercano
   end
end
```

dentro.m

```
function booleano=dentro(pto,Vhl,Vlr)
% calcula si el punto dado, esta en el interior
%      del rectangulo formado por Vhigh-letf,Vlow-rigth
%         Vhl =   vertice superior-izquierdo de la ventana de visualizacion
%         Vlr =   vertice inferior - derecho de la ventana de visualizacion
% SINTAXIS:  booleano=dentro([x y],Vhl,Vlr)

booleano=0;
x=pto(1);   y=pto(2);
if x>Vlr(1) | x<Vhl(1)  % si la coordenada X esta dentro de los limites...
   break
end
if y>Vhl(2) | y<Vlr(2)  % y la coordenada Y esta dentro de los limites...
   break
end
booleano=1;          % ...es que el punto esta dentro del rectangulo
```

gr_newtx.m

```
function [S,ok]=gr_newtx(XY0,toleran,N_max)
% S=gr_newtx([x0 y0],toleran,N_max_iteraciones)
%   calcula la solución (sx,sy) para f(x,y), con y=cte.,
%   por el método de Newton-Raphson, para un valor inicial (x0,y0)
%   y con un error máximo igual o menor a la tolerancia.

ok=0;           % ok=0 => inicialmente no se ha encontrado solucion
x0=XY0(1); y0=XY0(2);

if abs(fxy(x0,y0))<=toleran      % el pto inicial es la solucion
   S=[x0 y0];
   ok=1;           % ok=1 => se ha encontrado una solucion
   break
end

den=fxy_px(x0,y0);    % denominador del metodo de Newton
if den==0             % si denominador = 0, la solucion sera erronea
   break
end
p(1)=x0-fxy(x0,y0)/den;        % primer pto mediante Newton
if abs(fxy(p(1),y0))<=toleran % el primer pto es la solucion
   S=[p(1) y0];
   ok=1;           % ok=1 => se ha encontrado una solucion
   break
end


den=fxy_px(p(1),y0); % denominador del metodo de Newton
if den==0             % si denominador = 0, la solucion sera erronea
   break
end
p(2)=p(1)-fxy(p(1),y0)/den;    % segundo pto mediante Newton
if abs(fxy(p(2),y0))<=toleran % el segundo pto es la solucion
   S=[p(2) y0];
   ok=1;           % ok=1 => se ha encontrado una solucion
   break
end

count=2;
% repetir hasta encontrar la solucion, o consumir las iteraciones maximas
while (count<N_max)&(abs(fxy(p(count),y0))>=toleran)
   den=fxy_px(p(count),y0);       % denominador del metodo de Newton
   if den==0                      %   para evitar la division por cero
      break
   end
   count=count+1;
   p(count)=p(count-1)-fxy(p(count-1),y0)/den;  % nuevo pto de Newton
end

   % si finalmente se ha encontrado la solucion, se devuelve en "S"
if (abs(fxy(p(count),y0))<=toleran)
   S=[p(count) y0];
   ok=1;           % ok=1 => se ha encontrado una solucion
end

% sino se encuentra solucion, simplemente no se devuelve S=solucion
p=0;
```

gr_newty.m

```
function [S,ok]=gr_newtx(XY0,toleran,N_max)
% S=gr_newtx([x0 y0],toleran,N_max_iteraciones)
%   calcula la solución (sx,sy) para f(x,y), con x=cte.,
%   por el método de Newton-Raphson, para un valor inicial (x0,y0)
%   y con un error máximo igual o menor a la tolerancia.

ok=0;           % ok=0 => inicialmente no se ha encontrado solucion
x0=XY0(1); y0=XY0(2);

if abs(fxy(x0,y0))<=toleran      % el pto inicial es la solucion
   S=[x0 y0];
   ok=1;           % ok=1 => se ha encontrado una solucion
   break
end

den=fxy_py(x0,y0);    % denominador del metodo de Newton
if den==0             % si denominador = 0, la solucion sera erronea
   break
end
p(1)=y0-fxy(x0,y0)/den;              % primer pto mediante Newton
if abs(fxy(x0,p(1)))<=toleran % el primer pto es la solucion
   S=[x0 p(1)];
   ok=1;           % ok=1 => se ha encontrado una solucion
   break
end


den=fxy_py(x0,p(1)); % denominador del metodo de Newton
if den==0             % si denominador = 0, la solucion sera erronea
   break
end
p(2)=p(1)-fxy(x0,p(1))/den;    % segundo pto mediante Newton
if abs(fxy(x0,p(2)))<=toleran % el segundo pto es la solucion
   S=[x0 p(2)];
   ok=1;           % ok=1 => se ha encontrado una solucion
   break
end

count=2;
% repetir hasta encontrar la solucion, o consumir las iteraciones maximas
while (count<N_max)&(abs(fxy(p(count),y0))>=toleran)
   den=fxy_py(x0,p(count));       % denominador del metodo de Newton
   if den==0                      %   para evitar la division por cero
      break
   end
   count=count+1;
   p(count)=p(count-1)-fxy(x0,p(count-1))/den;  % nuevo pto de Newton
end

   % si finalmente se ha encontrado la solucion, se devuelve en "S"
if (abs(fxy(x0,p(count)))<=toleran)
   S=[x0 p(count)];
   ok=1;           % ok=1 => se ha encontrado una solucion
end

% sino se encuentra solucion, simplemente no se devuelve S=solucion
p=0;
```

fxy.m

```
function p=fxy(x,y)
% fxy(x,y) calcula en el punto dado, el valor de
%  la funcion:  x^3 + x^2*y + y^7*x - 2*y - 1 = 0
%  o
%  la funcion:  x^4*y - x^6 + y^5*x - 3*y + 2 = 0

% Comentar la que no convenga
%p=x^3 + x^2*y + y^7*x - 2*y - 1;
p=x^4*y - x^6 + y^5*x - 3*y + 2;
```

f xy_px.m

```
function p=fxy_px(x,y)
% fxy_p(x,y) calcula en el punto dado, el valor de
%  la derivada respecto "x" de la funcion:
%     x^3 + x^2*y + y^7*x - 2*y - 1 = 0
%  o
%     4*x^3*y - 6*x^5 + y^5  =  0
%
%  es decir, de la f'(x,Y).

% Comentar la que no convenga
%p=3*x^2+2*x*y+y^7;
p=4*x^3*y-6*x^5+y^5;
```

fxy_py.m

```
function p=fxy_py(x,y)
% fxy_p(x,y) calcula en el punto dado, el valor de
%  la derivada respecto "y" de la funcion:
%     x^3 + x^2*y + y^7*x - 2*y - 1 = 0
%  o
%     4*x^3*y - 6*x^5 + y^5  =  0
%
%  es decir, de la f'(X,y).

% Comentar la que no convenga
%p=x^2+7*y^6*x-2;
p=x^4+5*y^4*x-3;
```

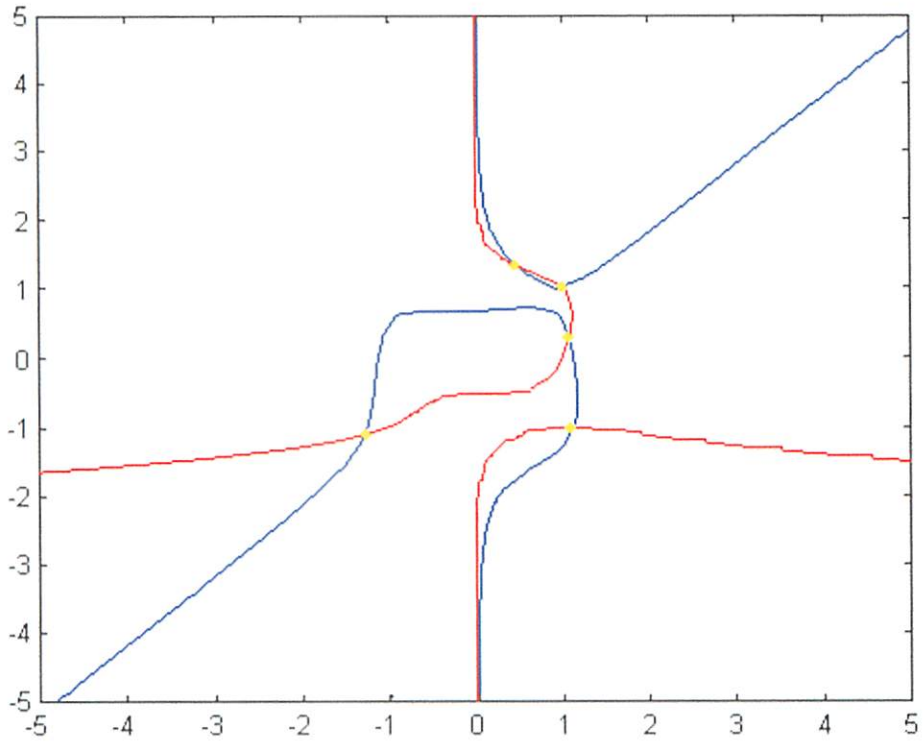
Como resultado de este método, se obtiene que la solución buscada está alrededor de $p \approx (1,1)$, pero es una apreciación muy inexacta.

También se puede ver que cerca de ésta solución existe otras soluciones singulares como $p_1 \approx (1.36, 0.48)$, y $p_2 \approx (0.28, 1.08)$.....

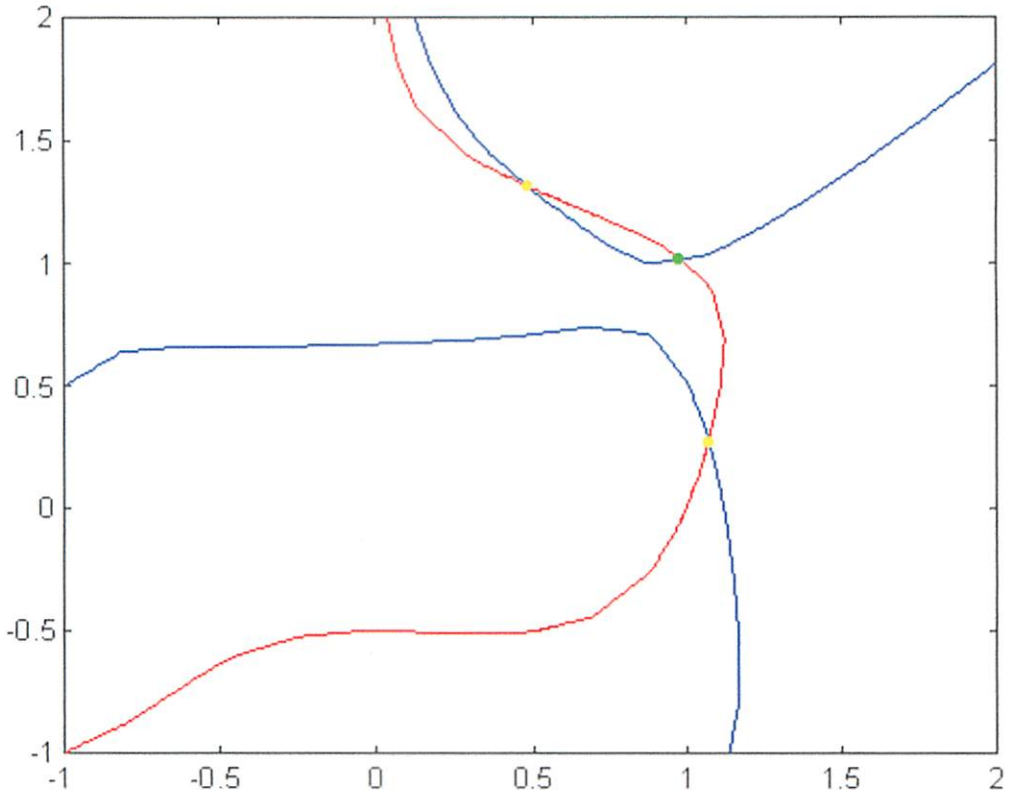
Y en la representación más pequeña se puede ver que hay soluciones generales, es decir, existe una recta vertical, X=0, en la cual coinciden ambas funciones, es más, dicha recta es asíntota para ambas funciones.

Todas las gráficas comentadas previamente pueden observase en la página siguiente.

```
» grafica([-5,5],[5,-5],16,'r-')
     % cambiar las funciones:  fxy, fxy_px y fxy_py
» grafica([-5,5],[5,-5],16,'b-')
```




```
» grafica([-1,2],[2,-1],16,'r-')
     % cambiar las funciones:  fxy, fxy_px y fxy_py
» grafica([-1,2],[2,-1],16,'b-')
```



· Método del Punto fijo

Fundamento teórico

La forma general de un sistema de ecuaciones no lineales en un espacio n- dimensional que es la siguiente:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = f_1(\mathbf{x}) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = f_2(\mathbf{x}) = 0 \\ \cdot \\ \cdot \\ \cdot \\ f_n(x_1, x_2, \ldots, x_n) = f_n(\mathbf{x}) = 0 \end{cases}$$

donde cada función $f_i(...) = 0$ puede considerarse como una proyección del vector $X \in R_N$ en la recta real R, por lo tanto podemos representar este sistema de forma vectorial definiendo una función $\mathbf{F}: R_N \rightarrow R_N$, con lo cual el sistema queda de la forma:

$$\mathbf{F(x) = 0}$$

nota: se utilizará el formato negrilla para indicar que algo no es escalar, sino que tiene carácter vectorial.

Las funciones f_i se denomina funciones coordenadas de **F**, igual que x_i se denominan coordenadas del punto **x**.

Al igual que para las ecuaciones lineales en una sola variable, un requisito imprescindible para el correcto funcionamiento de algunos de los siguientes métodos, es que el sistema de ecuaciones sea continuo al menos en el dominio de estudio.

Sea **F** una función definida en el conjunto D: $R_N \rightarrow R_N$, se dice que la función **F** es continua en **x** perteneciente al conjunto si el límite de dicha función existe y además coincide con el valor de la función en el punto **x**. Y por extensión se dice que **F** es continua en el conjunto D si **F** es continua en cada **x** del conjunto, y se expresa como $\mathbf{F} \in C(D)$.

Se puede también extender el teorema del punto fijo al caso n-dimensional, y que por definición para funciones vectoriales también se dice que **G**: $R_N \rightarrow R_N$ tiene un punto fijo en $P \in D$ si $\mathbf{G(p)=p}$, y por lo tanto:

> "dado un conjunto n-dimensional acotado, y un sistema de ecuaciones **F**, suponemos que **G** es una función continua, con primeras derivadas parciales continuas y con la propiedad de que $\mathbf{G(x)} \in D$ para todos los valores de $\mathbf{x} \in D$, entonces **G** tiene un punto fijo en el conjunto D. Si además existe una constante $K < 1$ tal que:
>
> $$\left| \frac{\delta g_i(\mathbf{x})}{\delta x_j} \right| = \frac{K}{n}$$
>
> entonces la sucesión de puntos $\{ \mathbf{x}^{(k)} \}_{k=0..\infty}$ definida a partir de un punto inicial arbitrario $\mathbf{x}^{(0)}$ perteneciente al conjunto D y generada por:
>
> $$\mathbf{x}^{(k)} = \mathbf{G}(\mathbf{x}^{(k-1)}), \quad \text{para } k > 0$$
>
> converge al único punto fijo P del conjunto.
>
> Y además puede acotarse el error cometido en cada iteración por la fórmula:
>
> $$\|\mathbf{x}^{(k)} - \mathbf{p}\|_\infty = \frac{K^k}{1 - K} = \|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\|_\infty \quad \text{“}$$

Metodología

Siguiendo el estilo de trabajo empleado en la primera práctica, los algoritmos implementados en ésta pretenderán ser lo más genéricos posibles. Así que comenzamos creando un archivo que contenga el sistema de ecuaciones a resolver **f2d.m**, que aunque no será utilizado por el método de punto fijo, será empleado por otros, y su ubicación aquí permite seguir mejor el "hilo" del desarrollo de la práctica.

A partir del sistema de ecuaciones original creamos otro archivo que contenga el sistema de ecuaciones g´s correspondientes, denominados **g2d.m**. Como hasta este momento no he conseguido calcular una función **G(x)** que cumpliese los requisitos para ser contractiva, se implementarán dos versiones del archivo **g2d.m**, uno con una función que no converge y otro con las g´s de Newton, y que por lo tanto si convergen.

La implementación del método del punto fijo (**pfijo_2d.m**) en sí es prácticamente igual al caso de una sola ecuación, simplemente se trabaja con vectores de la dimensión correspondiente (dimensión igual número de incógnitas).

Inicialmente se reserva una matriz con la dimensión suficiente para almacenar todos los puntos intermedios hasta llegar a la solución o al número máximo de iteraciones permitidas (parámetro entrada), se considera como punto inicial p_0 al suministrado por el usuario, y se calcula el punto siguiente mediante la función **g2d.m**. En este momento se comprueban por primera vez los criterios de paro, que son: que la distancia (tomada como norma infinita) entre dos puntos consecutivos sea menor a la tolerancia predefinida, o que se haya excedido el número máximo de iteraciones. Se repite el cálculo de nuevos puntos mediante la función **g2d.m** aplicada sobre los puntos que se van hallando, hasta cumplir alguno de los dos criterios de paro, y finalmente se comprueba si realmente se ha obtenido la solución deseada o se informa de lo contrario.

Implementación y resultados

Al igual que para el caso de una sola ecuación no lineal, el método del punto fijo tiene el problema de la búsqueda de las funciones contractivas adecuadas, además de ser un método lineal, por lo que la búsqueda de una solución con una tolerancia muy pequeña necesita de muchas iteraciones.

Concretamente, en la tabla siguiente se representan los resultados obtenidos tomando como punto inicial p_0 = (0.8, 0.8), tanto para las funciones g's que convergen como para las que no.

Se puede observar que son necesaria **6** iteraciones para obtener la exactitud deseada (tolerancia=10^{-6}) para las funciones convergentes, también se puede ver que para las funciones no convergentes, los resultados son ya dispares desde la segunda iteración.

Es importante destacar que el número de operaciones necesarias para alcanzar dicha solución final, en el caso de poder encontrarse, es **662**.

```
» [solucion,ptos,n_it]=pfijo_2d([0.8,0.8],10^-6,20)
```

nº iteración	g2d que NO convergente	g2d que SI convergente
0	(0.8 , 0.8)	(0.8 , 0.8)
1	(3.0003560000 , 0.7758933333)	(1.21541085188956 , 0.72403361344538)
2	(-10.72431537 , -221.26610505)	(1.09236480787471 , 1.05266620011927)
3	(-1.1735168944 , 0.0189593060) $\cdot 10^{14}$	(1.01637802132815 , 1.00439939732694)
4	(-0.0000000004 , -8.7059230344) $\cdot 10^{83}$	(1.00063244174590 , 0.99972466948386)
5	(-Inf , NaN)	(1.00000130190530 , 0.99999896744666)
6	(NaN , NaN)	(1.00000000000631 , 0.99999999999654)
FLOPS	----------	662

f2d.m

```
function v=f2d(p)
% f2d([x y]) calcula en el punto dado, el valor de
%  las funciones:
%                x^3 + x^2*y + y^7*x - 2*y - 1 = 0
%                x^4*y - x^6 + y^5*x - 3*y + 2 = 0

v=zeros(2,1);          % inicializa el vector solucion
x=p(1); y=p(2);

v(1)=x^3 + x^2*y + y^7*x - 2*y - 1; % 1ª componente de la solucion
v(2)=x^4*y - x^6 + y^5*x - 3*y + 2; % 2ª componente de la solucion
```

g2d.m

```
function v=g2d(p)
% g2d([x y]) calcula en el punto dado, el valor de
%  las funciones g1, g2 de forma "paralela"

x=p(1); y=p(2);
v=zeros(1,2);          %  inicializa el vector solucion

% Funciones g´s que NO convergen
   v(1)=-(y*x^3  + y^8*x - 2*y^2 - y)/(x*y^2);  % 1ª componente
   v(2)=(x^4*y - x^6 + y^5*x  +2)/3;            % 2ª componente

% Funciones g´s que SI convergen  (g´s de Newton)
   % temp=-jacob_2d(p)\f2d(p);
   % v(1)=p(1)+temp(1);          % 1ª componente de la solucion
   % v(2)=p(2)+temp(2);          % 2ª componente de la solucion
```

pfijo 2d.m

```
function [S,ptos,n_it]=pfijo_2d(p,TOL,N_max)
% calcula la raiz de la funcion f2d(Xp,Yp) por el metodo del pto. fijo
%  SINTAXIS:
%           [S,ptos,n_it]=pfijo_2d(p0,TOL,N_max)

p0=p;           % p0 = punto de inicio ("anterior")
p1=g2d(p0);     % p1 = primer punto mediante punto fijo ("siguiente")

n=1;
dimension=size(p);
% inicializa una matriz con capacidad para los maximos puntos necesarios
ptos=zeros(N_max,dimension(2));
ptos(n,:)=p0;            % primer pto de la sucesion el de inicio

% criterio de paro 1 = norma de dos ptos seguidos menor que tolerancia
% criterio de paro 2 = sobrepasa iteraciones maximas predefinidas
while ~(norm(p0-p1,inf)<TOL)&(n<N_max)
   n=n+1;
   p0=p1;          % actualiza el "anterior"
   p1=g2d(p0);     % calcula un nuevo "siguiente"
   ptos(n,:)=p0;   % almacena el "anterior" en la serie de ptos
end

n_it=n;         % devuelve el numero de iteraciones necesarias
if (norm(p0-p1,inf)<TOL) %si la norma entre los ultimos 2 ptos calculados
   S=p1                  % es menor que la tolerancia,tenemos la solucion
else
   'solución no encontrada para ',N_max,'iteraciones.'
end
```

· Método del Seidel

Fundamento teórico

El método de Seidel (**seidel.m**) es prácticamente idéntico al del punto fijo, si bien introduce una pequeña modificación en la forma de calcular los puntos de aproximación. La diferencia de cálculo entre ambos métodos es la siguiente:

· el método del punto fijo calcula en cada iteración todas las coordenadas de un nuevo punto en forma "paralela", es decir, sin usar los nuevos valores calculados progresivamente.

· en cambio Seidel concluye que es preferible ir utilizando en el cálculo de cada una de las coordenadas i-ésima el nuevo valor actualizado de las coordenadas anteriores (i-1), y por lo tanto el método de Seidel utiliza una forma de cálculo en "serie" de cada de las funciones g´s que forma el nuevo punto.

Metodología

El método en sí no ha cambiado, tan sólo ha cambiado la forma de calcular los nuevos puntos, lo cual requiere que en la función auxiliar **g2seidel.m**, que es prácticamente idéntica a la función anterior **g2d.m**, simplemente se utiliza el nuevo valor calculado de "x" durante el cálculo de "y".

Implementación y resultados

Para obtener los resultados para este método, se utilizarán los parámetros de entrada siguientes:

· punto inicial **x** = (0.8, 0.8)
· tolerancia 10^{-6}
· nº máximo de iteraciones 20

```
» [solucion,ptos,n_it]=seidel2d([0.8,0.8],10^-6,20)
```

En la tabla siguiente se puede observar que son necesaria **5** iteraciones para obtener el resultado con la tolerancia deseada, y también es importe el número de operaciones necesarias para alcanzar el resultado que es **1108**.

Como se puede apreciar existe una ligera mejora en el número de iteraciones respecto del punto fijo original, aún basándose prácticamente en lo mismo, no así en cuanto al número de operaciones (ya que debe calcular las g´s de Newton el doble de veces, es decir, debe resolver el jacobiano el doble de veces debido a la actualización de las componentes).

Se puede apreciar que, aunque se utilicen funciones no convergentes, la sucesión es más rápida incluso para no converger, es decir, alcanza los mismos valores que el punto fijo en menos iteraciones.

nº iteración	g2Seidel que NO convergente	g2Seidel que SI convergente
0	(0.8 , 0.8)	(0.8 , 0.8)
1	(3.0003560000 , -2.2056842715)	(1.21541085188956 , 0.92672587041896)
2	(0.0000000000 , -7.7703255111) $\cdot 10^{83}$	(1.07860497460425 , 0.99253030247615)
3	(-Inf , NaN)	(1.01385459108316 , 0.99953533139917)
4	(NaN , NaN)	(1.00051098351389 , 0.99999956259622)
5	(NaN , NaN)	(1.00000071705912 , 0.99999999999875)
FLOPS	----------	1108

seidel2d.m

```
function [S,ptos,n_it]=seidel2d(p,TOL,N_max)
% calcula la raiz de la funcion f2d(p) por el metodo
%      del pto. fijo de forma secuencial o Seidel.
%      [S,ptos,n_it]=Seidel2d(p0,TOL,N_max)

p0=p;                    % p0 = punto de inicio ("anterior")
p1=g2seidel(p0);         % p1 = primer punto mediante Seidel ("siguiente")
n=1;
dimension=size(p);
ptos=zeros(N_max,dimension(2));  % inicializa una matriz con capacidad
ptos(n,:)=p0;                    % para los maximos ptos necesarios

% criterio de paro 1 = norma de dos ptos seguidos menor que tolerancia
% criterio de paro 2 = sobrepasa iteraciones maximas predefinidas
while ~(norm(p0-p1,inf)<TOL)&(n<N_max)
   n=n+1;
   p0=p1;                % actualiza el "anterior"
   p1=g2seidel(p0);      % calcula un nuevo "siguiente"
   ptos(n,:)=p0;         % almacena el "anterior" en la serie de ptos
end

n_it=n;                  % devuelve el numero de iteraciones necesarias
echo on
if (norm(p1-p0,inf)<TOL) % si la norma entre los ultimos dos ptos calculados
   S=p1                  % es menor que la tolerancia, obtenemos la solucion
   ptos(n,:)=p0;
else
   'solución no encontrada para ',N_max,'iteraciones.'
end
```

g2seidel.m

```
function v=g2seidel(p)
% g2seidel([x y]) calcula en el punto dado, el valor de
%  las funciones g1, g2 de forma "secuencial"

x=p(1); y=p(2);
v=zeros(1,2);            %  inicializa el vector solucion

% Funciones g´s que NO convergen
   v(1)=-(y*x^3  + y^8*x - 2*y^2 - y)/(x*y^2);      % 1ª componente
      x=v(1);      % actualiza el valor de X
   v(2)=(x^4*y - x^6 + y^5*x  +2)/3;                % 2ª componente

% Funciones g´s que SI convergen  (g´s de Newton)
   % temp=-jacob_2d(p)\f2d(p);
   % v(1)=p(1)+temp(1);          % 1ª componente de la solucion
   %   p(1)=p(1)+temp(1);  % actualiza el valor de X
   % temp=-jacob_2d(p)\f2d(p);
   % v(2)=p(2)+temp(2);          % 2ª componente de la solucion
```

· Método de Newton

Fundamento teórico

Como ha quedado patente en la resolución del sistema de ecuaciones mediante punto fijo, no siempre es fácil encontrar unas funciones de convergencia para un sistema dado, lo que nos lleva a implementar un procedimiento más general para la resolución de un sistema con éste es el procedimiento Newton .

El algoritmo de Newton parte al igual que en el caso unidimensional de la búsqueda de una función ϕ de la forma:

$$g(x) = x - \phi(x) \cdot f(x),$$

que obtuviese una serie de convergencia cuadrática al punto fijo P. De aquí surge el método de Newton, escogiendo $\phi(x) = 1 / f'(x)$.

De la misma manera se puede extender para el caso n-dimensional, teniendo en cuenta que en lugar de la función $\phi(x)$, en forma vectorial lo que buscamos es la función vectorial $\mathbf{A(x)}$, que viene determinada mediante la matriz:

$$\mathbf{A(x)} = \begin{pmatrix} a_{11}(\mathbf{x}) & a_{12}(\mathbf{x}) & \ldots\ldots & a_{1n}(\mathbf{x}) \\ a_{21}(\mathbf{x}) & a_{22}(\mathbf{x}) & \ldots\ldots & a_{2n}(\mathbf{x}) \\ . & . & & . \\ . & . & & . \\ a_{n1}(\mathbf{x}) & a_{n2}(\mathbf{x}) & \ldots\ldots & a_{nn}(\mathbf{x}) \end{pmatrix}$$

y que además debe cumplir la propietaria de que: $\mathbf{G(x)} = \mathbf{x} - \mathbf{A(x)}^{-1} \cdot \mathbf{F(x)}$

y por lo tanto debe ser una matriz no singular, ya que si no sería posible hallar su inversa.

El método de Newton por lo tanto se basa en el cumplimiento del siguiente teorema:

> "supóngase que **p** es una solución de $\mathbf{G(x)} = \mathbf{x}$, con **G**: $R_N \rightarrow R_N$, y que existe un número $\delta > 0$ con la propiedad de que:
>
> · $\frac{\delta g_i(\mathbf{x})}{\delta x_j}$ es continua en $N_\delta = \{ \mathbf{x} / \ \|\mathbf{x} - \mathbf{p}\| < \delta \}$ para cada $i = 1, 2, \ldots\ldots, n$ y $j = 1, 2, \ldots., n$.
>
> · $\frac{\delta^2 g_i(\mathbf{x})}{\delta x_j \delta x_k}$ es continua y $\left| \frac{\delta^2 g_i(\mathbf{x})}{\delta x_j \delta x_k} \right| \leq M$ para alguna constante M, siempre que $\mathbf{x} \in N_d$ para cada $i = 1, 2, \ldots\ldots, n$, $j = 1, 2, \ldots., n$ y $k = 1, 2, \ldots., n$.
>
> · $\frac{\delta g_i(\mathbf{p})}{\delta x_j} = 0$ para cada $i = 1, 2, \ldots\ldots, n$ y $j = 1, 2, \ldots., n$.
>
> entonces existe un número $\delta' < \delta$ tal que la sucesión generada por $\mathbf{x}^{(k)} = \mathbf{G}(\mathbf{x}^{(k-1)})$ converge cuadráticamente a **p** para cualquier $\mathbf{x}^{(0)}$ siempre que $\|\mathbf{x}^{(k)} - \mathbf{p}\| < \delta'$ "

Una elección apropiada para esta matriz $\mathbf{A(x)}$ es la matriz Jacobiana, pero aún así sigue siendo necesaria que la inversa de esta matriz exista, además de conocer un valor inicial suficientemente cercano a la solución final.

Aunque este método posee muchos puntos a favor, como el ser un método "general", adolece de un problema y es el gran gasto computacional que supone trabajar como matrices inversas.

Metodología

La implementación del algoritmo de Newton (**newton2d.m**) comienza, al igual que las anteriores, reservando la memoria suficiente para albergar a toda la serie de puntos de aproximación a solución (se reserva una matriz con el máximo de elementos posibles que es igual al número máximo de iteraciones permitidas).

Después, y partiendo del punto inicial suministrado como parámetro, se calcula el descenso del error o "incremento de Newton" (z), y se entra en un bucle en el cual mientras la diferencia entre dos puntos consecutivos no sea menor que la tolerancia, se calcula un nuevo punto a partir de la anterior, así como el nuevo incremento de Newton asociado al último punto calculado.

Para el cálculo de los sucesivos incrementos (z) se utilizan dos funciones auxiliares, una anteriormente presentada **f2d.m** que proporciona el valor de la función original en un determinado punto y otra que calcular el valor de la matriz Jacobiana para dicho punto (**jacob_2d.m**).

Finalmente se comprueba si se ha salido del bucle porque realmente se ha obtenido la solución deseada, o bien porque se ha excedido el número de iteraciones máximas permitidas, y se informa adecuadamente de cuál de los casos se ha dado.

Implementación y resultados

Para obtener los resultados con este método se utilizarán los parámetros de entrada utilizados también con los anteriores métodos:

- punto inicial $\mathbf{x}$ = (0.8, 0.8)
- tolerancia 10^{-6}
- nº máximo de iteraciones 20

es decir, que los resultados se obtienen mediante la llamada siguiente:

```
» [solucion,ptos,n_it]=newton2d([0.8,0.8],10^-6,20)
```

Y estos resultados se presentan en la siguiente tabla:

nº iteración	Sucesión de Newton
0	(0.8 , 0.8)
1	(1.21541085188956 , 0.72403361344538)
2	(1.09236480787471 , 1.05266620011927)
3	(1.01637802132815 , 1.00439939732694)
4	(1.00063244174590 , 0.99972466948386)
5	(1.00000130190530 , 0.99999896744666)
6	(1.00000000000631 , 0.99999999999654)
FLOPS	646

Se puede observar que son necesaria **6** iteraciones para obtener el resultado con la tolerancia deseada, y que el número de operaciones necesarias para alcanzarlo es **646**.

Newton2d.m

```
function [S,ptos,n_it]=Newton2d(p,TOL,N_max)
% calcula la raiz de la funcion f2d(p) por el metodo de Newton
%     [S,ptos,n_it]=Newton2d(p0,TOL,N_max)

z=zeros(2,1);
ptos=zeros(N_max,2);
p0=p;                          % p0 = punto de inicio ("anterior")
ptos(n,:)=p0;
% corrector del error de Newton:  z = jacobiano / funcion original
z=-jacob_2d(p0)\f2d(p0);

n=1;
% criterio de paro 1 = norma de dos ptos seguidos menor que tolerancia
%                      equivalente a que: "z" sea menor que tolerancia
% criterio de paro 2 = sobrepasa iteraciones maximas predefinidas
while ~(norm(z,inf)<TOL)&(n<N_max)
   n=n+1;
   p0=p0+z';           % actualiza el "anterior", es decir, siguiente pto de Newton
   ptos(n,:)=p0;       % almacena el pto en la serie de ptos de aproximacion
   z=-jacob_2d(p0)\f2d(p0);    % recalcula el corrector del error de Newton "z"
end

n_it=n;                % devuelve el numero de iteraciones necesarias
echo on
if (norm(z,inf)<TOL) % si la norma entre los ultimos dos ptos calculados,es decir
   S=p0+z'             %   si "z" es menor que la tolerancia, obtenemos la solucion
else
   'solución no encontrada para ',N_max,'iteraciones.';
end
```

jacob_2d.m

```
function M=jacob_2d(p)
% calcula la matriz Jacobiana de f2d(p) en el punto dado por el vector 'p',
%  y devuelve la matriz numerica M de dimension 2x2

M=zeros(2,2);
x=p(1); y=p(2);

% derivada de f1 respecto x , y
M(1,:)=[3*x^2+2*x*y+y^7    x^2+7*y^6*x-2];
% derivada de f2 respecto x , y
M(2,:)=[4*x^3*y-6*x^5+y^5  x^4+5*y^4*x-3];
```

f2d.m

```
function v=f2d(p)
% f2d([x y]) calcula en el punto dado, el valor de
%  las funciones:
%                x^3 + x^2*y + y^7*x - 2*y - 1 = 0
%                x^4*y - x^6 + y^5*x - 3*y + 2 = 0

v=zeros(2,1);          % inicializa el vector solucion
x=p(1); y=p(2);

v(1)=x^3 + x^2*y + y^7*x - 2*y - 1; % 1ª componente de la solucion
v(2)=x^4*y - x^6 + y^5*x - 3*y + 2; % 2ª componente de la solucion
```

· Método de Broyden

Fundamento teórico

El método de Broyden es una generalización del método de la secante aplicado a sistemas de ecuaciones no lineales. Es uno de los métodos conocidos como "renovaciones de secante de mínimo cambio" y que por lo tanto produce un algoritmo denominado cuasi-Newton.

Éste método sigue la estrategia del método de Newton, si bien reemplaza la matriz Jacobiana que se usaba en el método de Newton por una matriz de aproximación que se va renovando en cada iteración.

Como ventaja obtenemos el no tener que resolver la inversa de la matriz Jacobiana en cada iteración, si bien también tiene desventajas, como que no es un método cuadrático sino que es tan sólo super-lineal, y que generalmente el método de Newton es autocorregido, es decir que corrige errores de redondeo mediante iteraciones sucesivas, en cambio, el método de Broyden no lo es .

Realmente el método de Broyden es totalmente equivalente al de Newton hasta la primera iteración, a partir de la segunda iteración se separa del método de Newton, y reemplaza la matriz Jacobiana por la matriz de aproximación, que puede definirse como:

$$\mathbf{A}_0 = \mathbf{J}(\mathbf{x}_0)$$

$$\mathbf{A}_n = \mathbf{A}_{n-1} + \frac{[\mathbf{F}(\mathbf{x}_n) - \mathbf{F}(\mathbf{x}_{n-1}) - \mathbf{A}_{n-1} \cdot (\mathbf{x}_n - \mathbf{x}_{n-1})] \cdot [\mathbf{x}_n - \mathbf{x}_{n-1}]'}{\|\mathbf{x}_n - \mathbf{x}_{n-1}\|_2^2}$$

Con lo cual se consigue una disminución del coste computacional a cambio de una menor potencia en la resolución del sistema de ecuaciones .

Metodología

Como se ha comentado en el apartado de fundamento teórico el método de Broyden (**broyden.m**) también denominado cuasi Newton es idéntico al método original de Newton hasta la primera iteración, entonces se entra en el bucle cuyos criterios de paro son los mismos que en Newton, es decir, que dos puntos sucesivos sean muy próximos o que se haya sobrepasado el número máximo de iteraciones permitidas, y en el interior de este bucle se van calculando tanto los nuevos puntos de aproximación como las nuevas matrices de corrección del error (matrices de aproximación $\mathbf{A}_n$).

Finalmente, y al igual que en el método de Newton , se comprueba si se ha salido del bucle por qué realmente se ha obtenido la solución deseada, o bien porque se ha excedido el número de iteraciones máximas permitidas, y se informa adecuadamente de lo sucedido .

Implementación y resultados

Para la obtención de los resultados con el método de Broyden, no se volverán a utilizar en la llamada a la función que implementa dicho método los parámetros de entrada que he considerado por defecto, es decir, que la llamada a la función será como siempre:

```
» [solucion,ptos,n_it]=broy_2d([1.2,1.6],10^-6,20)
```

Ya que utilizando el punto inicial $\mathbf{x} = (0.8, 0.8)$ el método converge muy lentamente hacia la solución deseada, en cambio, para muchos otros puntos la convergencia es mucho más veloz. Por lo tanto creo que el caso del punto inicial $\mathbf{x} = (0.8, 0.8)$, no es realmente una buena generalización del comportamiento del método, ya que hace que los resultados oscilen durante demasiado tiempo hasta llegar a la solución $\mathbf{p} \approx (1.0 , 1.0)$, más concretamente para dicho caso necesita de **350** iteraciones, además de un gasto computacional igual a **54342** operaciones.

Como se puede observar en la tabla adjunta, que resume los resultados para el punto inicial **x** = (1.2, 1.6), este método necesita **16** iteraciones para obtener el resultado final con la tolerancia deseada, también hay que destacar que el gasto computacional ha sido el máximo hasta el momento, es decir, han sido necesarias **2316** operaciones para llegar a la solución, lo cual supone prácticamente el doble gasto computacional respecto del peor de los métodos anteriores. Cabe la posibilidad de que esto sea debido a la "lejanía" de punto inicial a la solución, pero aún así, el número de iteraciones y de operaciones necesarias es muchísimo menor al obtenido para el punto de inicio **x** = (0.8, 0.8).

nº iteración	Sucesión de Broyden
0	(1.2 , 1.6)
1	(1.65601387799386 , 1.25799792569164)
2	(1.13106025022449 , 1.37632561321901)
3	(1.12851730911349 , 1.30042562554186)
4	(1.11944576397478 , 1.15829900814644)
5	(1.09834045410916 , 1.07935011648677)
6	(1.06831858531754 , 1.02278222617867)
7	(1.04340947318809 , 0.99543546388034)
8	(1.02850882544378 , 0.98802705240078)
9	(1.02069903612890 , 0.98932221466217)
10	(1.01129767433681 , 0.99387343549227)
11	(1.00197191951337 , 0.99880903318257)
12	(1.00049253946264 , 0.99972960668975)
13	(0.99984583241510 , 1.00007969239815)
14	(1.00001247258969 , 0.99999354859836)
15	(1.00000001003408 , 0.99999999626456)
16	(0.99999999545764 , 1.00000000237089)
FLOPS	2316

Broyden.m

```
function [S,ptos,n_it]=Broy_2d(p,TOL,N_max)
% calcula la raiz de la funcion f2d(p) por el metodo de Broyden (quasi-Newton)
%      [S,ptos,n_it]=Broy_2d(p0,TOL,N_max)

format long
ptos=zeros(N_max,2);
p0=p;                          % p0 = punto de inicio ("anterior")
ptos(n,:)=p0;
A=jacob_2d(p0);                % primera matriz de aproximacion = jacobiano
z=-A\f2d(p0);                  % primer corrector del error = al de Newton

n=1;
% criterio de paro 1 = norma de dos ptos seguidos menor que tolerancia
%                      equivalente a que: "z" sea menor que tolerancia
% criterio de paro 2 = sobrepasa iteraciones maximas predefinidas
while ~(norm(z,inf)<TOL)&(n<N_max)
   n=n+1;
   p1=p0+z';           %  siguiente pto de Broyden
   ptos(n,:)=p0;       % almacena el pto en la serie de ptos de aproximacion
   % recalcula la matriz de aproximacion desde el nuevo punto
   A=A+((f2d(p1)-f2d(p0)-A*(p1-p0)')*(p1-p0))/(norm(p1-p0,2)^2);
   z=-A\f2d(p1);       % recalcula el corrector del error de Broyden "z"
   p0=p1;              % actualiza el punto "anterior" con el "siguiente"
end

n_it=n;                % devuelve el numero de iteraciones necesarias
if (norm(z,inf)<TOL)   % si la norma entre los ultimos 2 ptos calculados,es decir
   S=p0+z'             % si "z" es menor que la tolerancia, obtenemos la solucion
else
   'Solución no encontrada, excedido el n° de iteraciones: ',N_max
end
```

· Método del Máximo descenso

Fundamento teórico

Frente a los métodos locales en los cuales prima su rapidez, existen métodos menos potentes pero que eliminan la necesidad de conocer una aproximación inicial lo suficientemente cercana a la solución final para asegurar la convergencia del método. Por lo tanto éste es un método de naturaleza global, lo que implica que, aunque su convergencia es sólo lineal, sea un método aceptable para encontrar una aproximación suficiente para posteriormente afinar con el método de Newton local.

La finalidad del método del descenso mediante el gradiente es encontrar un mínimo de una función **g**: $R_N \rightarrow R$, que representa el error cometido entre la aproximación y el valor real de la solución, y su funcionamiento puede describirse de la manera siguiente:

· evaluar la función **g** en una aproximación inicial $\mathbf{x_0}$
· encontrar la dirección a partir de $\mathbf{x_0}$ en la cual se dé como resultado un descenso en el valor de la función de error **g**.
· decidir qué distancia nos debemos desplazar en esa dirección, y llamar a éste nuevo valor destino $\mathbf{x_1}$.
· repetir los pasos anteriores reemplazando a $\mathbf{x_0}$ por $\mathbf{x_1}$

El gradiente de una función con múltiples variables es el análogo de la derivada en una función de una sola variable, por lo tanto si en una sola variable solo podían haber un mínimo local en aquellos puntos donde la derivada es igual a cero, igualmente para funciones multivariadas sólo es posible hallar un mínimo en aquellos puntos donde el gradiente sea igual a cero.

Otra característica del gradiente es que mide la "cantidad" de cambio en el valor de la función **g(x)**, indicando siempre la dirección de máximo ascenso. Por lo tanto la técnica del descenso más rápido se basará precisamente en ir avanzando en sentido contrario a lo indicado por el gradiente, con lo cual iremos descendiendo en la dirección de máximo descenso posible en cada "paso". Por lo tanto si utilizamos "pasos" no demasiado grandes nos conducirá a un mínimo local cercano al punto de inicio.

Ahora bien, si consideramos como función **g** una función que nos cuantifique el error cometido al aproximarnos hacia la solución **p**, la aplicación del máximo descenso nos conducirá a minimizar el error, con lo cual llegaremos a un punto tan cercano a la solución de la función original como deseemos. Más concretamente la función de cuantificación del error que utilizaremos será la siguiente:

h: $R_N \rightarrow R$
h: $X,Y \rightarrow X^2 + Y^2$

Metodología

Como se ha puesto de manifiesto en el apartado teórico, el método de máximo descenso (**max_grd2.m**) necesita dos magnitudes fundamentales que son: el gradiente en cada punto determinado y lo que podríamos denominar como distancia de recorrido (lambda).

Por ser éste un método de carácter lanzadera no nos preocuparemos por devolver todos los puntos de la serie de aproximación hasta el valor final, sino que la respuesta de este método será simplemente el valor final encontrado, para usarlo en la llamada a otro método.

El funcionamiento en sí del algoritmo es el siguiente: partiendo de un punto inicial cualquiera se determina la dirección de máximo descenso de la función de error (**h2d.m**) mediante el uso del gradiente (**grad_h2d.m**) así como el error actual en este punto inicial. Una vez calculada la dirección adecuada se

va probando desplazamientos en esta dirección cada vez más pequeños, hasta encontrar algún desplazamiento en el cual el error obtenido en el destino es menor que en el punto actual. Si se consigue disminuir el error, se considera como nuevo punto origen el último punto calculado, y se repiten los pasos del cálculo de la dirección de máximo descenso para seguir buscando puntos destinos con menor error a la actual, y así sucesivamente hasta encontrar un punto en el cual el error sea menor a la tolerancia predefinida.

Finalmente se comprueba si efectivamente hemos logrado una solución satisfactoria, en caso contrario se informa de que no ha sido posible encontrar la solución para el número iteraciones máximo permitidas.

Implementación y resultados

Los resultados que se obtendrán para el método del descenso mediante el gradiente, no serán como en las ocasiones anteriores, ya que utilizaremos un punto inicial mucho más alejado de la solución buscada, pero también le pediremos menos exactitud (tolerancia 10^{-3}) en la llamada a dicho método:

```
» solucion=max_grd2([4 5],10^-3,100)

     solucion =   (1.0083   ,   0.9928)
```

Sin duda, este es posiblemente el método más lento de todos los estudiados hasta el momento, ya que para una solución con una tolerancia bastante alta, como la utilizada en la llamada anterior, el número de operaciones necesarias se dispara hasta el límite de necesitar ni más ni menos que **4130**.

Tomando el punto inicial **x** = (4, 5) la convergencia es excelente, pero no es así por ejemplo en puntos cercanos a éste, e incluso en puntos bastante mas cercanos a la solución, es decir, que el método del máximo descenso no nos asegura que para un punto dado vaya a converger a la solución más cercana a éste, por ejemplo, para los puntos **x** = (2, 2), **x** = (0, 0), **x** = (2, 2):

```
» solucion=max_grd2([2 2],10^-3,100)

     solucion =   (1.0716 ,   0.2943)

» solucion=max_grd2([0 0],10^-3,100)

     solucion no encontrada para 100 iteraciones

» solucion=max_grd2([2 -1],10^-3,100)

     solucion =   (1.0710   ,   0.3010)
```

en ninguno de ellos la solución suministrada, si la hay para el número de iteraciones máximo, es la solución [10] más cercana al punto de inicio.

[10] de todas las soluciones posibles.

max_grd2.m

```
function v=max_grd2(p,TOL,N_max)
% max_grd2([x y],TOL,N_max) calcula un punto tan cercano a la solucion
%      como deseemos, a partir del punto dado, y usando como maximo
%      el numero de iteraciones indicadas.
%
%      La proximidad de un determinado punto a la solucion, es decir,
%   la TOLerancia sera calculada por medio de la funcion error "h_f2d".

x=p(1); y=p(2);
v=zeros(2,1);
temp=p;
error=inf;  % el error inicial sera infinito, ya que facilita la busqueda del minimo
n=0;

% criterio de paro 1 = error sea menor o igual al esperado (TOLerancia)
% criterio de paro 2 = sobrepasar las iteraciones maximas predefinidas
while (error>TOL)&(n<N_max)
   p0=temp;
   grad=grad_h2d(p0);       % direccion de maximo ascenso = gradiente
   err_actual=h_f2d(p0);    % error actual calculado por la funcion de error "h"
   error=inf;               % error siguiente, inicialmente infinito
   lambda=1;    %  proporcionalidad del desplazamiento: (n=0) => 2^-n = 1

   % repetir hasta encontrar un error mejor al actual
   while error>=err_actual
      temp=p0-lambda*grad; % calcula un pto siguiente mediante el maximo descenso
      error=h_f2d(temp);   % calcula el error cometido en este nuevo punto
      lambda=lambda/2;     % reduce la "cantidad de desplazamiento" a la mitad
   end
   n=n+1;   temp           % visualiza la nueva aproximacion
end

if (error<=TOL)       % si el error final esta dentro del preestablecido
   v=temp             %   devuelve la solucion
else
   'solución no encontrada para ',N_max,'iteraciones.';
end
```

h_f2d.m

```
function v=h_f2d(p)
% h_f2d([x y]) calcula el error cometido en la aproximacion
%      a la solucion en el punto dado, para las funciones
%      que contiene el fichero "f2d", es decir,
%      para las funciones:
%                  x^3 + x^2*y + y^7*x - 2*y - 1 = 0
%                  x^4*y - x^6 + y^5*x - 3*y + 2 = 0
temp=f2d(p);                % calcula el valor de las funciones
v=temp(1)^2+temp(2)^2;      % y el error de la forma: error = (x^2+ y^2)
```

grad_h2d.m

```
function v=grad_h2d(p)
% grad_h2d([x y]) calcula el valor numerico del gradiente de
%      la funcion error "h_f2d" en el punto dado.
x=p(1); y=p(2);
v=zeros(1,2);
% derivada de h(x,y)=R respecto de X
v(1)=2*(x^3+x^2*y+y^7*x-2*y-1)*(3*x^2+2*x*y+y^7)+...
     2*(x^4*y-x^6+y^5*x-3*y+2)*(4*x^3*y-6*x^5+y^5);
% derivada de h(x,y)=R respecto de Y
v(2)=2*(x^3+x^2*y+y^7*x-2*y-1)*(x^2+7*y^6*x-2)+...
     2*(x^4*y-x^6+y^5*x-3*y+2)*(x^4+5*y^4*x-3);
```

· Método de Newton-global

Fundamento teórico

Como su propio nombre indica el método de Newton global es la generalización del método de Newton local en el cual no es necesario un punto inicial realmente próximo a la solución buscada. Éste método aúna las ventajas de la potencia del método local de Newton junto con la generalidad del método de máximo descenso, y de hecho basa su funcionamiento en ambos métodos.

En sí el método global es prácticamente igual al método local, es decir, el cálculo de los sucesivos puntos de aproximación se basa en la misma fórmula, la matriz Jacobiana. Si bien el método global añade un sistema para verificar que el punto siguiente de la sucesión realmente es más próximo a solución que el punto del cual partimos. Y para ello utiliza una técnica semejante a la empleada en el método del máximo descenso, es decir, cuantificar la longitud de los "pasos" que damos desde un punto al siguiente, y comprobar que el error cometido en este nuevo punto es menor que el que cometíamos en el punto anterior.

Metodología

Al igual que en el método de máximo descenso, el método de Newton global (**new_gl2d.m**) necesita dos magnitudes fundamentales para trabajar, que son: el valor de la "z" de Newton en cada punto determinado y lo que podríamos denominar como distancia de recorrido o factor de cuantificación (lambda).

Inicialmente, como hacíamos en los métodos locales, inicializamos una matriz vacía con capacidad suficiente para albergar a toda la sucesión de puntos de aproximación a solución final, por si fuese necesarios la dimensión de esta matriz será igual al número máximo de iteraciones permitidas.

Después entramos en el bucle más externo, que se comporta exactamente igual que el implementado en el método local, y una vez calculado el "descenso del error" de Newton (z) así como el error cometido en el punto actual (mediante la función **h2d.m**), nos introducimos en el bucle más interno que desestima todos los puntos "alcanzables" desde el actual (mediante la cuantificación de "z"), en los cuales el error no disminuya. Una vez conseguido algún punto para el cual el error es menor que en el punto actual, se reemplaza el punto origen por el nuevo punto encontrado, y repetimos los cálculos sucesivamente hasta encontrar un punto que cumpla con la tolerancia exigida mediante el parámetro de entrada.

Implementación y resultados

Para la ejecución de este método se utilizarán como parámetros de entrada un punto relativamente lejano de la solución buscada, más concretamente lanzaremos la búsqueda de la solución desde el punto **x** = (18, 8). Pero utilizaremos una tolerancia pequeña (10^{-6}), como las utilizadas también con los métodos locales, es decir, que los resultados se obtienen mediante la llamada siguiente:

```
» [solucion,ptos,n_it]=new_gl2d([18,8.1],10^-6,30)
```

nº iteración	Sucesión de Newton
0	(18 , 8)
1	(15.02873242297618, 7.04566289009782)
4	(8.78195425067476 , 4.80981394096448)
7	(5.18656985319653 , 3.27614224258536)
10	(3.12476856594444 , 2.21595893369366)
13	(1.92806482600319 , 1.46179548167544)
16	(1.19147228735975 , 0.95647749234021)
19	(1.00046232215629 , 0.99963393112702)
22	(1.00000000000214 , 0.99999999999842)
FLOPS	2963

Como puede observarse en la tabla de la página anterior, el número de iteraciones no es excesivo, ya que tan sólo necesita **22** iteraciones hasta llegar a una aproximación del orden de una millonésima, si bien el número de operaciones necesarias es considerable **2963**, aunque no está nada mal para un punto inicial tan lejano.

new_gl2d.m

```
function [S,ptos,n_it]=New_gl2d(p,TOL,N_max)
% New_gl2d([x y],TOL,N_max) calcula un punto tan cercano a la solucion
%      como deseemos, a partir del punto dado, usando como maximo
%      el numero de iteraciones indicadas.
%
%      La proximidad de un determinado punto a la solucion, es decir,
%   la TOLerancia sera calculada a partir de la norma de Z.

x=p(1); y=p(2);
S=zeros(1,2);
ptos=zeros(N_max,2);     % reserva la matriz de ptos de salida
ptos(1,:)=p;             % primer pto de la serie = pto inicial
n=2;
temp=p;
errorZ=inf; % el error inicial sera infinito, ya que facilita la busqueda del minimo
z=zeros(2,1);
err_actual=h_f2d(p); % error inicial calculado por la funcion de error "h"

% criterio de paro 1 = error sea menor o igual al esperado (TOLerancia)
% criterio de paro 2 = la funcion de error "h" sea menor o igual a TOLerancia
% criterio de paro 3 = sobrepasar las iteraciones maximas predefinidas
while (errorZ>TOL)&(err_actual>TOL)&(n<=N_max)
   p0=temp;
   z=-jacob_2d(p0)\f2d(p0);    %calcula el corrector de error de Newton "z"
   error=inf;                  % error siguiente, inicialmente infinito
   lambda=1;       %  proporcionalidad del desplazamiento: (n=0) => 2^-n = 1

   % repetir hasta encontrar un error mejor al actual
   while error>=err_actual
      temp=p0+lambda*z';    %calcula un pto siguiente mediante submultiplos de "z"
      error=h_f2d(temp);    % calcula el error cometido en este nuevo punto
      lambda=lambda/10;     % reduce el proximo submultiplo a la decima parte
   end

   err_actual=error(p0);    % actualizacion del error ual calculado por la funcion
error "h"
   ptos(n,:)=temp;          % añade un nuevo pto a la serie de aproximaciones
   errorZ=norm(z,inf);      % actualiza el error cometido
   n=n+1;
end

if n<N_max
   n_it=n-1;          % devuele el n° de iteraciones empleadas
end

if (errorZ<=TOL)|(err_actual<=TOL)  % si el error final esta dentro del
   S=temp                           % preestablecido devuelve la solucion
else
   'solución no encontrada para ',N_max,'iteraciones.';
end
```

· Análisis comparativo de los distintos métodos

En las tablas que se presentan a continuación se resumen los resultados obtenidos para los distintos métodos implementados en esta práctica para una serie de valores iniciales diferentes. Como se puede apreciar el método del punto fijo obtiene siempre los mismos resultados que el de Newton, ya que como se explicó en el apartado correspondiente no he sido capaz de encontrar funciones g's que convergesen, y por lo tanto dicho método emplea las g's de Newton, de ahí la similitud de los resultados para ambos métodos.

En el caso del método de Seidel ocurre algo parecido, si bien las iteraciones empleadas por éste difieren de las usadas por el punto fijo y Newton.

En lugar de representar los resultados parciales obtenidos, se representará el error cometido en cada iteración[11] para cada una de estas aproximaciones, ya que creo que es bastante más significativo que una ristra de puntos, muchas veces muy cercanos unos de otros. También en las últimas dos fila se exponen tanto la solución final alcanzada, como el gasto computacional de cada método, expresado en operaciones de coma flotante necesarias para llegar a la solución final.

· Resultados para el punto inicial **x** = (0.2 , 2), y tolerancia 10^{-6} (en máximo descenso tolerancia=10^{-3}) :

nº iteración	Punto Fijo	Seidel	Newton	Broyden	Máx. descenso	Newton global
0	433.7684066	433.7684066	433.7684066	433.7684066	433.7684066	433.7684066
1	31.2329135	58.8874476	31.2329135	31.2329135	41.8872631	31.2329135
2	0.7660736	7.5580831	0.7660736	2.4746188	16.6368178	0.7660736
3	0.0061194	0.6124415	0.0061194	0.9970827	4.2319299	0.0061194
4	$0.685 \cdot 10^{-4}$	0.0141561	$0.685 \cdot 10^{-4}$	3.1087447	0.4824447	$0.685 \cdot 10^{-4}$
5	$\mathbf{0.66 \cdot 10^{-10}}$	$0.142 \cdot 10^{-4}$	$\mathbf{0.66 \cdot 10^{-10}}$	3.3906376	0.2166905	$\mathbf{0.66 \cdot 10^{-10}}$
6	------	$\mathbf{0.16 \cdot 10^{-10}}$	------	3.1085410	0.2060850	------
7	------	------	------	5.0844138	0.1973074	------
9	------	------	------	101.7834683	0.1787625	------
11	------	------	------	80.4590539	0.1613649	------
13	------	------	------	0.8877013	0.1456654	------
15	------	------	------	0.1823629	0.1319618	------
17	------	------	------	0.0424439	0.1246449	------
19	------	------	------	0.0012710	0.1091675	------
21	------	------	------	$0.80 \cdot 10^{-6}$	0.0983272	------
23	------	------	------	$\mathbf{0.28 \cdot 10^{-10}}$	0.0912824	------
30	------	------	------	------	0.0704129	------
40	------	------	------	------	0.0442839	------
50	------	------	------	------	0.0283821	------
60	------	------	------	------	0.0190829	------
70	------	------	------	------	0.0118839	------
80	------	------	------	------	0.0079168	------
90	------	------	------	------	0.0051695	------
100	------	------	------	------	0.0034089	------
110	------	------	------	------	0.0023249	------
120	------	------	------	------	0.0015523	------
130	------	------	------	------	**0.0009928**	------
Solución ≈	**(0.48 , 1.30)**	**(0.48 , 1.30)**	**(0.48 , 1.30)**	**(0.99 , 1.00)**	**(0.48 , 1.30)**	**(0.48 , 1.30)**
FLOPS	**566**	**1290**	**552**	**3521**	**46893**	**615**

[11] Calculado mediante la función de error: $h = x^2 + y^2$

· Resultados para el punto inicial **x** = (-3 , -3), y tolerancia 10^{-10} (en máximo descenso tolerancia=10^{-6}) :

nº iteración	Punto Fijo	Seidel	Newton	Broyden	Máx. descenso	Newton global
0	42459968.00	42459968.00	42459968.00	42459968.00	42459968.00	42459968.00
1	4998001.75	4585050.78	4998001.75	4998001.75	15651643.55	4998001.75
2	587226.97	489740.38	587226.97	1448873.57	494.31	587226.97
3	68744.83	51390.16	68744.83	307152.82	29.23	68744.83
4	7985.07	5232.69	7985.07	76297.53	8.17	7985.07
5	909.22	505.77	909.22	19191.68	0.57	909.22
6	97.56	44.05	97.56	5507.70	0.25	97.56
7	8.64	2.88	8.64	1918.68	0.10	8.64
8	0.41	0.07	0.41	901.82	0.04	0.41
9	$0.32 \cdot 10^{-2}$	$0.13 \cdot 10^{-3}$	$0.32 \cdot 10^{-2}$	590.96	$1.18 \cdot 10^{-2}$	$0.32 \cdot 10^{-2}$
10	$0.30 \cdot 10^{-6}$	$0.45 \cdot 10^{-9}$	$0.30 \cdot 10^{-6}$	491.50	$1.01 \cdot 10^{-2}$	$0.30 \cdot 10^{-6}$
11	$0.23 \cdot 10^{-16}$	**$0.54 \cdot 10^{-20}$**	$0.23 \cdot 10^{-16}$	441.30	$0.90 \cdot 10^{-2}$	**$0.23 \cdot 10^{-14}$**
12	**$0.98 \cdot 10^{-30}$**	------	**$0.98 \cdot 10^{-30}$**	342.39	$0.77 \cdot 10^{-2}$	------
16	------	------	------	22.41	$0.50 \cdot 10^{-2}$	------
21	------	------	------	4.18	$0.33 \cdot 10^{-3}$	------
26	------	------	------	0.99	$0.19 \cdot 10^{-3}$	------
31	------	------	------	$0.22 \cdot 10^{-2}$	$0.71 \cdot 10^{-4}$	------
36	------	------	------	$1.96 \cdot 10^{-2}$	$0.62 \cdot 10^{-5}$	------
41	------	------	------	$0.38 \cdot 10^{-4}$	$0.10 \cdot 10^{-5}$	------
46	------	------	------	**$0.11 \cdot 10^{-12}$**	**$0.69 \cdot 10^{-6}$**	------
Solución ≈	**(-1.28, -1.11)**	**(-1.28, -1.11)**	**(-1.28, -1.11)**	**(-1.28, -1.11)**	**(1.07 , 0.28)**	**(-1.28, -1.11)**
FLOPS	**1224**	**2210**	**1196**	**7339**	**13343**	**1323**

· Resultados para el punto inicial **x** = (-3 , 3), y tolerancia 10^{-2} :

nº iteración	Punto Fijo	Seidel	Newton	Broyden	Máx. descenso	Newton global
0	44631908.00	44631908.00	44631908.00	44631908.00	44631908.00	44631908.00
1	5231046.96	6509546.25	5231046.96	5231046.96	27935390.72	5231046.96
2	609193.90	1357633.29	609193.90	1480480.36	16.61	609193.90
3	66858.21	766998.22	66858.21	280838.16	0.14	66858.21
4	2470.09	16867.52	2470.09	54024.49	0.13	2470.09
5	254.84	787.99	254.84	8067.42	0.12	254.84
6	15.84	112.96	15.84	1033.64	0.11	15.84
7	0.26	16.42	0.26	129.99	0.10	0.26
8	$0.34 \cdot 10^{-2}$	1.73	$0.34 \cdot 10^{-2}$	19.67	0.10	**$0.34 \cdot 10^{-2}$**
9	**$0.85 \cdot 10^{-5}$**	0.07	**$0.85 \cdot 10^{-5}$**	4.09	0.09	
10		**$0.37 \cdot 10^{-3}$**		1.37	0.08	
11				0.77	0.08	
12				0.61	0.07	
13				**0.57**	0.06	
20					0.03	
25					0.02	
30					**0.009**	
Solución ≈	**(0.48 , 1.30)**	**(0.48 , 1.30)**	**(0.48 , 1.30)**	***(0.62,1.17)***	**(1.07 , 0.21)**	**(0.48 , 1.30)**
FLOPS	**942**	**2026**	**920**	**2044**	**46893**	**969**

· Bibliografía

Los apuntes de clase, los suministrados por el profesor mediante fotocopias, y el libro:
Análisis Numérico Richard L. Burden, J. Douglas Faires. [Editorial Iberoamericana]

* el punto p ≈ (0.62, 1.17), no es solución del sistema, aunque si cumple uno de los criterios de parada del método, ya que la diferencia entre este punto y el anterior es menor que la TOLerancia dada.

Anexo A

Como se explicó en los apartados referentes a los métodos del Punto Fijo y Seidel, no me ha sido posible encontrar unas funciones que convergan hacia la solución para el sisema de ecuaciones dado, por lo tanto se presenta aquí otro sistema de ecuaciones "preparado" para que existan las funciones g's necesitadas para dichos métodos.

El sistema está "preparado" porque inicialmente he buscado dos funciones g's que cumplan las condiciones de convergencia en torno a la solución, y partiendo de ellas he contruido el sistema original, que es el siguiente:

$$\begin{cases} f_1(x, y) = x^2 - x \cdot y + y - 100x = 0 \\ f_2(x, y) = x^3 - x^2 \cdot y^2 - 100\,x^2 \cdot y + 99y^3 = 0 \end{cases}$$

Y el sistema de ecuaciones en notación de punto fijo desde el que se originan, es:

$$\begin{cases} g_1(x, y) = (x^2 - x \cdot y + y) / 100 \\ g_2(x, y) = (x^3 - x^2 \cdot y^2 - y^3) / 100 \cdot (x^2 - y^2) \end{cases}$$

Por lo tanto, cambiando en las funciones **g2d.m** y **g2seidel.m**, las g's anteriores, se consigue tener el sistema listo para que los métodos del punto fijo y Seidel convergan.

Los resultados para ambos métodos se presentan acontinuación, comparandolos con los resultados del método de Newton, que no sale muy beneficiado para este sistema de ecuaciones.

Los resultados presentados a continuación se han obtenido para una tolerancia de 10^{-16}, y tomando como punto inicial $\mathbf{x} = (0.2\ ,\ 0.6)$.

nº iteración	Punto Fijo	Seidel	Newton
0	(0.2 , 0.6)	(0.2 , 0.6)	(0.2 , 0.6)
1	(0.005200 , 0.006000)	(0.005200 , 0.006950)	(0.003730 , 0.367243)
2	(0.000059 , 0.000060)	(0.000069 , 0.000092)	(0.002431 , 0.243579)
3	$10^{-6} \cdot$ (0.6001 , 0.6001)	$10^{-5} \cdot$ (0.0923 , 0.1221)	(0.001613 , 0.161557)
4	$10^{-8} \cdot$ (0.6001 , 0.6002)	$10^{-7} \cdot$ (0.1221 , 0.1619)	(0.001070 , 0.107155)
5	$10^{-10} \cdot$ (0.6003 , 0.6003)	$10^{-9} \cdot$ (0.1619 , 0.2144)	(0.000710 , 0.071072)
6	$10^{-12} \cdot$ (0.6003 , 0.6003)	$10^{-11} \cdot$ (0.2144 , 0.2841)	(0.000471 , 0.047139)
7	$10^{-14} \cdot$ (0.6004 , 0.6004)	$10^{-11} \cdot$ (0.2144 , 0.2841)	(0.000312 , 0.031266)
8	**$10^{-16} \cdot$ (0.6004 , 0.6004)**	$10^{-13} \cdot$ (0.2841 , 0.3763)	(0.000207 , 0.020737)
9		$10^{-15} \cdot$ (0.3763 , 0.4985)	(0.000137 , 0.013754)
10		**$10^{-17} \cdot$ (0.4985 , 0.6604)**	(0.000091 , 0.009122)
15			(0.000011 , 0.001171)
25			$10^{-4} \cdot$ (0.0019 , 0.1929)
35			$10^{-6} \cdot$ (0.0031 , 0.3179)
45			$10^{-8} \cdot$ (0.0052 , 0.5238)
55			$10^{-10} \cdot$ (0.0086 , 0.8630)
65			$10^{-11} \cdot$ (0.0014 , 0.1422)
75			$10^{-13} \cdot$ (0.0023 , 0.2343)
85			**$10^{-15} \cdot$ (0.0038 , 0.3860)**
Solución ≈	**$10^{-19} \cdot$ (0.6604 , 0.8749)**	**$10^{-18} \cdot$ (0.6004 , 0.6005)**	**$10^{-15} \cdot$ (0.0016 , 0.1698)**
FLOPS	**202**	**182**	**7701**

Realmente no son unos resultados muy objetivos, ya que como he comentado, originalmente he partido de las funciones en notación de punto fijo, y además, de unas funciones de punto fijo con un grado de convergencia muy aceptable.

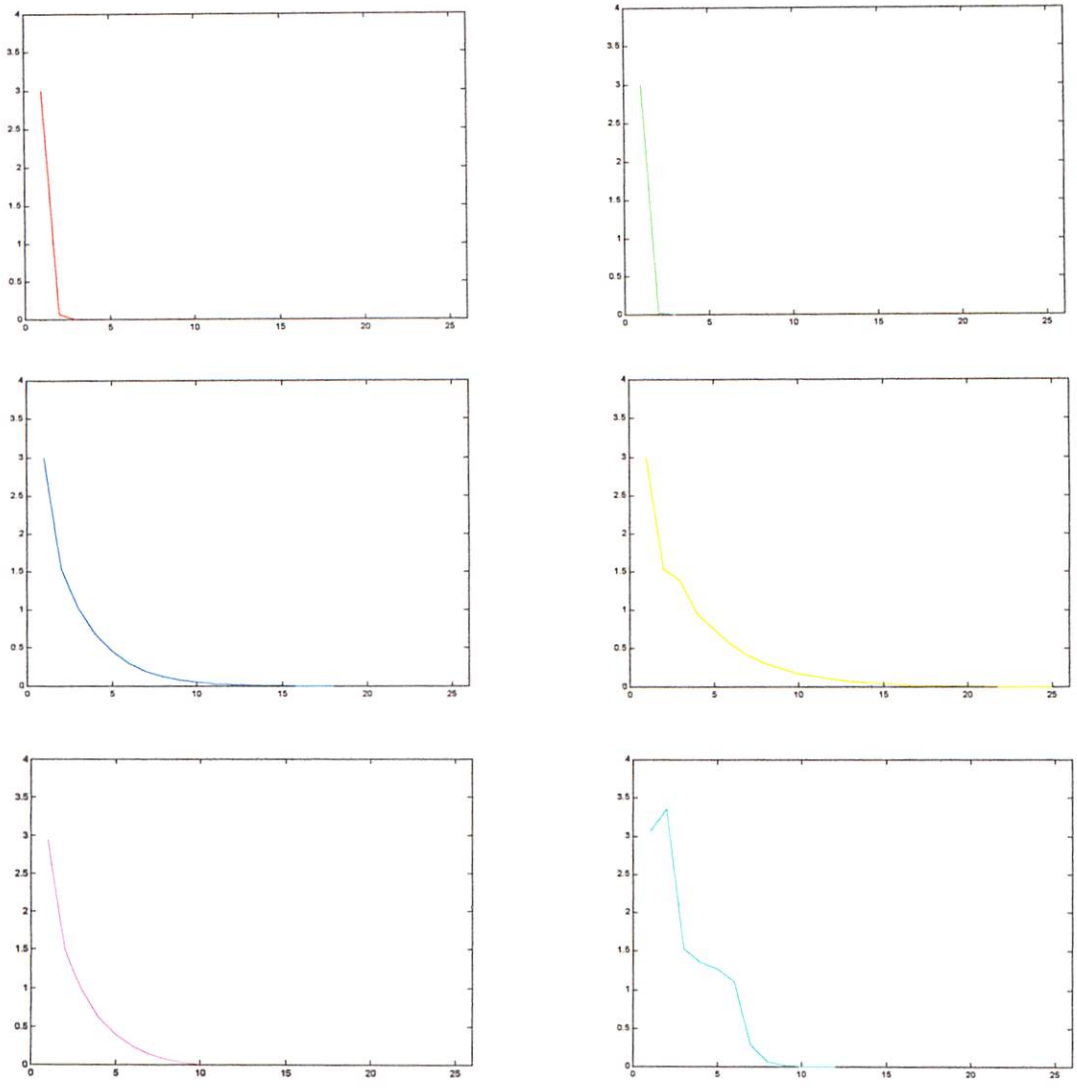
Anexo B

Para finalizar creo conveniente presentar unos gráfico que muestren la velocidad de convergencia de los distintos métodos, para lo cual, presentaré unas gráficas que representen la progresión del error cometido por cada método para las sucesivas iteraciones.

En la forma de calcular el error hay un poco de trampa, pero creo que es la manera más fidedigna, es decir, primero calcularé la solución para una determinada tolerancia, y posteriormente calcularé los sucesivos errores de cada iteración como la resta entre la solución final previamente calculada y la aproximación en cada una de las iteraciones.

Estas primeras seis gráficas corresponden a las aproximaciones hasta una tolerancia de 10^{-2}, para los métodos de[12]: punto fijo, Seidel, Newton, Broyden, Newton global y máximo descenso; aplicados a las funciones:

$$\begin{cases} f_1(x, y) = x^2 - x \cdot y + y - 100x = 0 \\ f_2(x, y) = x^3 - x^2 \cdot y^2 - 100\,x^2 \cdot y + 99y^3 = 0 \end{cases}$$

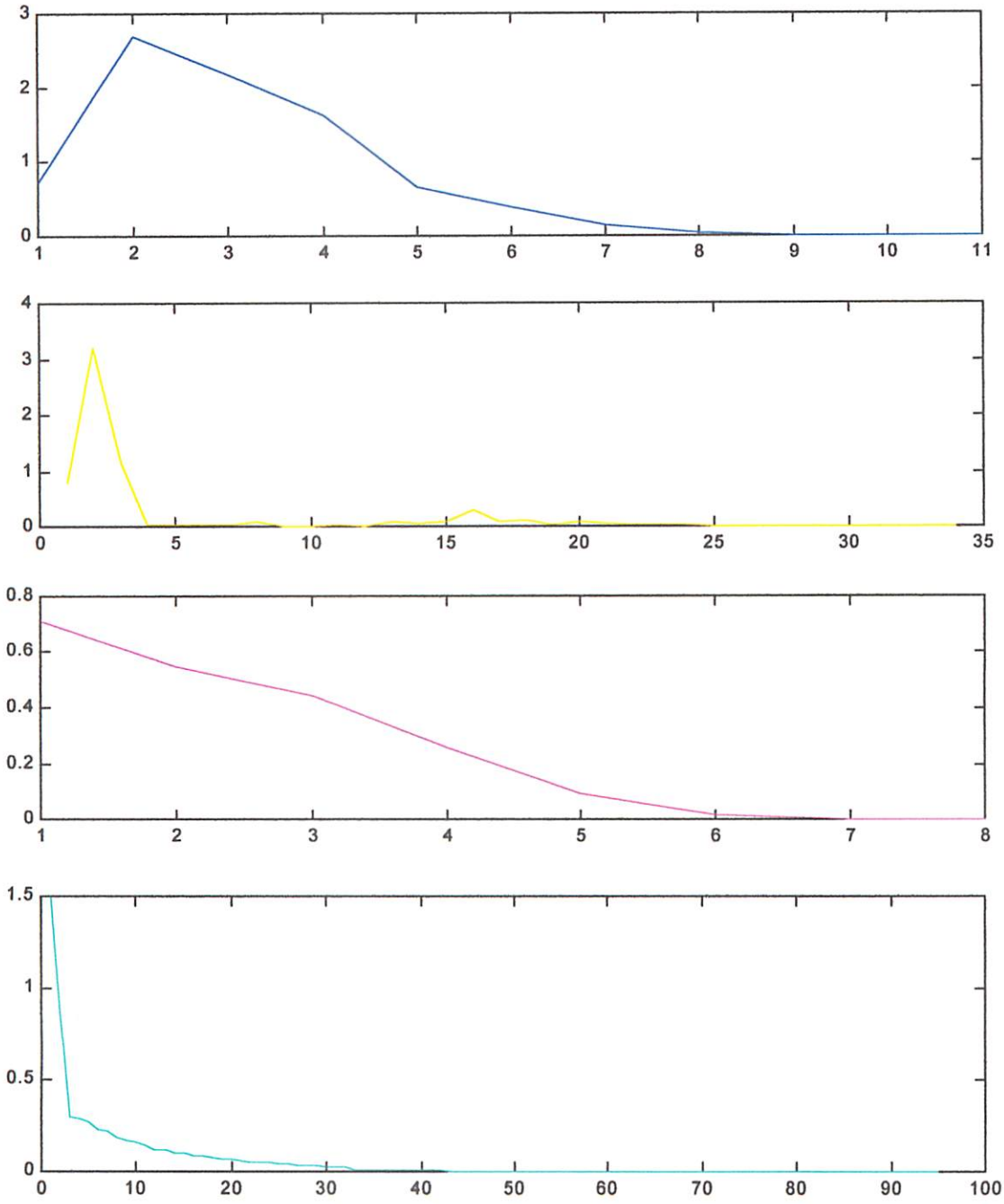


[12] Ordenados de izquierda a derecha y de arriba hacia abajo.

Y estas cuatro gráficas siguientes, corresponden a las aproximaciones hasta una mayor tolerancia, exactamente para 10^{-8}, pero sólo para los métodos de: Newton, Broyden, Newton global y máximo descenso, respectivamente; ya que serán aplicados a las funciones:

$$\begin{cases} f_1(x, y) = x^3 + x^2 \cdot y + x \cdot y^7 - 2y - 1 = 0 \\ f_2(x, y) = x^4 \cdot y - x^6 + x \cdot y^5 - 3y + 2 = 0 \end{cases}$$

, y no encuentro funciones de convergencia para dicho sistema.



Análisis Numérico I:

Interpolación Polinomial

Alumno:
Fernanso Sanz Clavijo

Asignatura:
Análisis Numérico I

Práctica **3**:
Interpolación Polinomial

Indice

· Objetivo de la Práctica

El objetivo principal de esta práctica es la comparación de distintos métodos de interpolación para una determinada función, más concretamente para la función:

$$f(x) = \frac{1}{1 + 12x^2}$$

Los métodos de interpolación polinomial que se implementarán serán: el de Lagrange y Neville, así como los algoritmos de diferencias divididas y diferencias progresivas o Newton.

Finalmente se realizará un estudio de los distintos errores cometidos por los métodos de interpolación anteriormente enumerados.

En esta práctica se utilizarán principalmente resultados del tipo gráfico, ya que resultan de comprensión más inmediata que las tablas de valores.

· Método de Lagrange

Fundamento teórico

Dados n+1 puntos (x_i,y_i), con i = 1, 2, 3, ..., n+1, de una función a interpolar, existe un único polinomio P de grado menor o igual a n tal que $p(x_i) = y_i$ para todo i = 1, 2, 3, ..., n+1, y dicho polinomio viene determinado mediante la fórmula de interpolación de Lagrange:

$$p(x) = \sum_{i=1}^{n+1} y_i \cdot L_i(x)$$

donde $L_i(x)$ es el denominado polinomio elemental de Lagrange para el nodo i-esimo, que se determina de la forma:

$$L_i(x) = \prod_{j=1,\, j \neq i}^{n+1} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_1) \cdot .... \cdot (x - x_{i-1}) \cdot (x - x_{i+1}) \cdot .... \cdot (x - x_{n+1})}{(x_i - x_1) \cdot .... \cdot (x_i - x_{i-1}) \cdot (x_i - x_{i+1}) \cdot .... \cdot (x_i - x_{n+1})}$$

Lógicamente el error cometido en la aproximación mediante interpolación depende de la función a interpolar, pero también de los nodos escogidos para el estudio, por lo cual será recomendable utilizar aquellos lodos en los cuales el error sea lo más pequeño posible, lo que implicaría una mejor aproximación a la función original.

Metodología

Para la implementación del método de interpolación de Lagrange he decidido dividir el funcionamiento del mismo en dos pequeños módulos, uno de ellos, denominado **lagrange.m**, se encarga de calcular el polinomio elemental i-esimo de Lagrange para una serie de nodos, siendo totalmente independiente del valor de la función en dichos nodos, es decir, que tan sólo necesita conocer las divisiones del intervalo de estudio, pero no necesita ninguna información de la función a estudiar.

El otro módulo denominado **poly_lag.m** se encarga de calcular el polinomio de interpolación propiamente dicho, a partir de una serie de nodos y valores de la función para dichos nodos, utilizando para ello al módulo de cálculo de los polinomios elementales. Concretamente la función que realiza es la aplicación

de la fórmula de interpolación de Lagrange, es decir, multiplica cada valor de la función en cada nodo por el polinomio elemental para dicho nodo.

El resultado obtenido mediante la aplicación de este método no es un valor numérico, sino un vector que contiene los coeficientes del polinomio interpolador, por lo tanto si deseamos conocer el valor de dicho polinomio en algún punto concreto deberemos hacer uso de la función de MATLAB *polival*, ya que se ha utilizado una salida compatible con la salida de la función *polyfit*.

Implementación y resultados

La implementación del módulo **poly_lag.m** no merece explicación alguna, pues simplemente se dedica a recorrer la matriz de nodos y valores de la función para dichos nodos, realizando el sumatorio de dichos valores por los polinomios elementales correspondientes, dando como resultado el polinomio de interpolación buscado.

En cambio el encargado del cálculo de los polinomios elementales, **lagrange.m**,si merece un poco de atención, ya que para tener como resultado un polinomio en lugar de un valor numérico, los cálculos difieren ligeramente de los expuestos en las clases de teoría. Realmente lo que se hace es y descomponiendo los cálculos en pequeños cálculos más sencillos, es decir, descomponer la multiplicación de los polinomios necesarios hasta lograr multiplicaciones entre un polinomio y un número, y después realizar la suma de todos los pequeños polinomio multiplicados, teniendo en cuenta el desplazamiento sufrido por cada grado, para obtener el polinomio final deseado.

La multiplicación entre polinomios se ha implementado en un módulo aparte, que será utilizado por prácticamente todos los métodos de la práctica, y es el que se encarga del trabajo sucio de descompone una multiplicación de dos polinomios genéricos en sumas de multiplicaciones de polinomios de grados menores.

```
function p=poly_lag(nodos_valores)
%  devuelve el polinomio de interpolación, para el vector de nodos-valores
%  utilizando los polinomios elementales de Lagrange.
%     L=poly_lag(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%              la 1ª son los nodos, y la 2ª los valores correspondientes

long=length(nodos_valores);
nodos=nodos_valores(:,1);      val=nodos_valores(:,2);
p=0;
while long~=0
   p=val(long)*lagrange(nodos,long)+p;
   long=long-1;
end
```

```
function L=lagrange(nodos,i)
%  devuelve el polinomio elemental de Lagrange para todos los nodos
%     L=lagrange(nodos,valores)
%           nodos  = vector fila
%           i         = polinomio i-esimo a calcular "Li"

resto=length(nodos);
temp=1; den01=1;
while resto~=0
   p01=[1 -nodos(resto)];
   if resto~=i
      den01=(nodos(i)-nodos(resto))*den01;
      temp=multi_p(p01,temp);
   end
   resto=resto-1;
end
L=temp/den01;
```

· Algoritmo de Neville

Fundamento teórico

Este algoritmo tan sólo es útil cuando no se necesita el cálculo del polinomio en sí, sino su valor en un punto determinado, pues la salida de este método que es un valor numérico.

Es un método interactivo que para de los polinomios de interpolación de grado cero[1] y va calculando progresivamente los polinomios de interpolación de grados superiores hasta llegar al grado n, para una interpolación de n+1 nodos.

El cálculo del valor de los polinomios de grado mayor de cero se obtiene mediante la siguiente fórmula:

$$P_{i,j}(x) = \frac{(x - x_i) \cdot P_{i+1,j}(x) - (x - x_j) \cdot P_{i,j-1}(x)}{(x_j - x_i)}, \qquad \text{con} \quad i < j$$

Metodología

Este algoritmo no necesita de ningún otro módulo para concluir su estudio, más concretamente se reduce a ir calculando las sucesivas iteraciones para los distintos polinomios, mediante la fórmula anteriormente expuesta, hasta llegar al polinomio de grado superior predefinido.

Implementación y resultados

La implementación realizada para este método, **neville.m**, se basa en la reiteración del cálculo del valor de los polinomios de los distintos grados, utilizando para ello los valores previamente calculados, es decir, los polinomios de grados inferiores, si bien el lugar de ir almacenando cada uno de dichos valores, en cada nueva iteración se irán destruyendo los valores previamente calculados, ya que no son necesarios para el cálculo de sucesivas iteraciones.

Por lo tanto la implementación queda reducida a dos bucles anidados dentro de los cuales se calcula que el valor del polinomio de grado i-esimo a partir de los de grado inferior.

```
function v=neville(nodos_valores,p)
%  devuelve el "valor" del polinomio de interpolación, para el vector nodos_valores
%   utilizando el lema de Aitken
%      v=neville(nodos_valores,punto)
%         nodos_valores = matriz de dos columnas,
%                 la 1ª son los nodos, y la 2ª los valores correspondientes
%         punto  = valor de "x" donde hallar el polinomio
long=length(nodos_valores);
x=nodos_valores(:,1);       y=nodos_valores(:,2);

for i=1:long-1
   for j=1:long-i
      y(j)=((p-x(j))*y(j+1)-(p-x(i+j))*y(j))/(x(i+j)-x(j));
   end
end
v=y(1)
```

[1] los valores de interpolación.

· Algoritmo de diferencias divididas

Fundamento teórico

Dados n+1 de la función original de la forma (x_i,y_i), el polinomio interpolación calculado mediante diferencias divididas viene determinado por:

$$p(x) = f[x_1] + f[x_1,x_2]\cdot(x - x_1) + f[x_1,x_2,x_3]\cdot(x - x_1)\cdot(x - x_2) + \ldots + f[x_1,x_2,\ldots,x_{n+1}]\cdot(x - x_1)\cdot(x - x_2)\cdot\ldots\cdot(x - x_n)$$

donde $f[x_1,x_2,\ldots,x_k]$ son las denominadas diferencias divididas para $k=1,2,\ldots,n+1$, las cuales pueden ser determinadas mediante el siguiente esquema recursivo:

$$\begin{cases} f[x_i] = y_i, & i = 1, 2, \ldots, n+1 \\ f[x_i, x_{i+1},\ldots, x_{i+k-1}, x_{i+k}] = \dfrac{f[x_{i+1},\ldots, x_{i+k-1}, x_{i+k}] - f[x_i, x_{i+1},\ldots, x_{i+k-1}]}{x_{i+k} - x_i} \end{cases}$$

Metodología

Nuevamente que decidido dividir el funcionamiento de este método en tres módulos, uno de ellos denominados **pol_ddif.m** tan sólo se utilizan para inicializar las variables de tipo global utilizadas, así como para realizar la primera llamada al módulo **difer.m** que se encargan de complementar la tabla de diferencias divididas deforma recursiva.

El tercer módulo que es el que realmente implementar el método, se denomina **dif_div.m** y es el encargado de ir construyendo recursivamente el polinomio interpolador, para lo cual hace uso tanto de **multi_p.m** como de **difer.m**, para la multiplicación de polinomios y para la obtención de las diferencias divididas, respectivamente.

Implementación y resultados

La implementación del lanzador del método no necesita explicación alguna, pues simplemente declarar como globales las variables que van a ser utilizadas por el resto de los módulos, lo cual no se puede hacer en dichos módulos porque ambos son de carácter recursivo, es decir, se llaman a sí mismo, lo cual impide la inició licitación de dichas variable. También el encargado de llamar por primera vez al módulo que calcula todas las diferencias divididas y por lo tanto el polinomio de interpolación.

El segundo módulo, **difer.m**, tiene dos formas de operación, cuando se le llama por primera vez calcula todas las diferencias divididas y las almacenan en una tabla de acceso global, y cuando se realizan sucesivas llamadas al mismo devuelve la diferencia divididas deseada mediante la consulta de dicha tabla.

Finalmente el módulo principal **dif_div.m** que es el que realmente calcular el polinomio interpolador, funciona de la siguiente manera:

- si el número de pares nodos-valores es tan sólo uno, devuelve como polinomio el valor asociado a dicho par.
- sino, calcular el coeficiente asociado a su correspondiente polinomio mediante diferencias divididas,
  y el denominador de dicho coeficiente como la resta del nodo i-esimo con el nodo inicial
- posteriormente realiza la multiplicación de todos los monomios a los cuales debe multiplicar el coeficiente calculado.
- finalmente realizar el sumatorio de dicha multiplicación con el resto de multiplicaciones (sumandos de la fórmula del polinomio de interpolación) obtenidas mediante sucesivas llamadas a sí mismo.

```
function pol=pol_ddif(nodos_valores)
%  lanzador para la inicializacion de las diferencias divididas
%     pol=pol_ddif(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%                la 1ª son los nodos, y la 2ª los valores correspondientes
tam=size(nodos_valores);
global tabla_dd,  global hechos_dd
tabla_dd=zeros(tam(1));    hechos_dd=zeros(tam(1));

pol=dif_div(nodos_valores,1);     % Lanza las diferencias divididas
```

```
function v=difer(nodos_valores,indice)
%  devuelve las diferencias divididas para una serie de nodos-valores
%     v=difer(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%                la 1ª son los nodos, y la 2ª los valores correspondientes
global tabla_dd,  global hechos_dd
nodos=nodos_valores(:,1);  long=length(nodos);
if long==1
   v=nodos_valores(1,2);
   tabla_dd(indice,long)=v;   hechos_dd(indice,long)=1;
   break
else
   if hechos_dd(indice+1,long-1)==0
      y1=difer(nodos_valores(2:long,:),indice+1);
   else
      y1=tabla_dd(indice+1,long-1);
   end
   if hechos_dd(indice,long-1)==0
      y0=difer(nodos_valores(1:long-1,:),indice);
   else
      y0=tabla_dd(indice,long-1);
   end
   coef=y1-y0;                den01=nodos(long)-nodos(1);
   v=coef/den01;
   tabla_dd(indice,long)=v;   hechos_dd(indice,long)=1;
end
```

```
function pol=dif_div(nodos_valores,indice)
%  devuelve el polinomio de interpolación, para el vector de nodos y valores
%  utilizando el algoritmo de diferencias divididas.
%     pol=dif_div(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%                la 1ª son los nodos, y la 2ª los valores correspondientes
nodos=nodos_valores(:,1);     long=length(nodos);
if long==1
   pol=nodos_valores(1,2);   break
else
   coef=difer(nodos_valores(2:long,:),indice+1)-
        difer(nodos_valores(1:long-1,:),indice);
   den01=nodos(long)-nodos(1);
end
resto=long-1;  temp=1;
while resto~=0
   p01=[1 -nodos(resto)];
   temp=multi_p(p01,temp);
   resto=resto-1;
end
ptemp=dif_div(nodos_valores(1:long-1,:),indice);
pol=(coef/den01)*temp+[0 ptemp];
```

· Fórmula de diferencias progresivas de Newton

Fundamento teórico

Si suponemos que los nodos están igualmente espaciados, entonces el polinomio de interpolación de los puntos (x_i,y_i) para $i = 1, 2, ..., n+1$ se puede calcular mediante:

$$p(x) = \sum_{k=0}^{n} \binom{s}{k} \cdot \Delta^k f(x_1)$$

El cálculo de las diferencias progresivas es semejante al de las diferencias divididas, si bien en este caso no existe ningún denominador que divida a la resta entre dos diferencias progresivas anteriores, es decir, que las diferencias progresivas están definidas mediante el siguiente esquema recursivo:

$$\begin{cases} \Delta^0 f(x_i) = y_i , & i = 1, 2, ..., n+1 \\ \Delta^k f(x_i) = \Delta^{k-1} f(x_{i+1}) - \Delta^{k-1} f(x_i) \end{cases}$$

Metodología

El método usado en este algoritmo es muy parecido al del algoritmo anterior, por lo tanto también lo será el diseño de ambos, por lo cual, otra vez he decidido dividir el funcionamiento de éste en cuatro módulos, uno de ellos denominados **pol_dpro.m** que tan sólo se utiliza para inicializar las variables de tipo global utilizadas, así como para realizar la primera llamada al módulo **difer_pr.m** que se encargan de complementar la tabla de diferencias progresivas de forma recursiva.

Otro de los módulos, **over.m**, se encarga de calcular un número sobre otro, más concretamente s sobre k, si bien en este caso tiene la particularidad de que s no es un valor numérico, sino un polinomio, y por lo tanto su resultado también será un polinomio.

El módulo principal que es el que realmente implementa el método, se denomina **dif_prog.m** y es el encargado de ir construyendo recursivamente el polinomio interpolador, para lo cual hace uso tanto de **multi_p.m** como de **difer_pr.m**, para la multiplicación de polinomios y para la obtención de las diferencias progresivas, respectivamente.

Implementación y resultados

La implementación del lanzador del método se resume como sigue: declara como globales las variables que van a ser utilizadas por el resto de los módulos, lo cual no se puede hacer en dichos módulos porque ambos son de carácter recursivo, lo cual impide la inicialización de dichas variable. También el encargado de llamar por primera vez al módulo que calcula todas las diferencias progresivas necesarias, con lo cual rellenamos la tabla en la cual se apoya el módulo principal.

El módulo auxiliar **over.m** sí entraña alguna pequeña dificultad, pues hasta momento todas las multiplicación es de polinomio se basaban en pequeñas multiplicaciones de monomios en los cuales el coeficiente de mayor grado era siempre igual a la unidad, por lo tanto era necesario eliminar del valor de s el coeficiente que divide a la variable X, lo cual ha sido resuelto realizando inicialmente el factorial del numerador de s y finalmente dividiendo dicho numerado entre el coeficiente de la variable X elevado al número de monomios que intervienen en dicho factorial.

El módulo **difer_pr.m**, tiene dos formas de operación, cuando se le llama por primera vez calcula todas las diferencias progresivas y las almacenan en una tabla de acceso global, y cuando se realizan sucesivas llamadas al mismo devuelve la diferencia progresiva deseada mediante la consulta de dicha tabla.

Finalmente el módulo principal **dif_prog.m** que es el que realmente calcular el polinomio interpolador, funciona de la siguiente manera:

- si el número de pares nodos-valores es tan sólo uno, devuelve como polinomio el valor asociado a dicho par.
- sino calcular el coeficiente asociado a su correspondiente polinomio mediante diferencias divididas.
- realiza múltiples llamadas a si mismo, hasta haber calculado todos los sumandos implicados en el polinomio de interpolación.
- y finalmente realiza el sumatorio de dichos sumandos con el resto, teniendo en cuenta que a cada sumando lo multiplica el resultado de s sobre, que también es un polinomio. dando como resultado el polinomio de interpolación completo.

```
function pol=pol_dpro(nodos_valores)
%  lanzador para la inicializacion de las diferencias divididas
%     pol=pol_ddif(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%              la 1ª son los nodos, y la 2ª los valores correspondientes
salir=0;     TOL=h/100;      n=2;
tam=size(nodos_valores);
xs=nodos_valores(:,1);      h=xs(2)-xs(1);
while n<tam(1)
   h1=xs(n+1)-xs(n);
   if (h<(h1-TOL)) | (h>(h1+TOL))
      disp('Diferencias progresivas debe usar nodos equidistantes');
      salir=1;
      break
   end
   n=n+1;
end
if ~salir
   global tabla_dp,  global hechos_dp
   tabla_dp=zeros(tam(1));     hechos_dp=zeros(tam(1));
   pol=dif_prog(nodos_valores,h,1);
end
```

```
function v=difer_pr(nodos_valores,indice)
%  devuelve las diferencias divididas para una serie de nodos-valores
%     v=difer_pr(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%              la 1ª son los nodos, y la 2ª los valores correspondientes
global tabla_dp,  global hechos_dp
nodos=nodos_valores(:,1);  long=length(nodos);
if long==1
   v=nodos_valores(1,2);
   tabla_dp(indice,long)=v;    hechos_dp(indice,long)=1;    break
else
   if hechos_dp(indice+1,long-1)==0
      y1=difer_pr(nodos_valores(2:long,:),indice+1);
   else
      y1=tabla_dp(indice+1,long-1);    end
   if hechos_dp(indice,long-1)==0
      y0=difer_pr(nodos_valores(1:long-1,:),indice);
   else
      y0=tabla_dp(indice,long-1);
   end
   v=y1-y0;
   tabla_dp(indice,long)=v;    hechos_dp(indice,long)=1;
end
```

```
function pol=dif_prog(nodos_valores,h,indice)
%  devuelve el polinomio de interpolación, para el vector de nodos y valores
%  utilizando diferencia progresiva de Newton
%     pol=dif_prog(nodos_valores)
%        nodos_valores = matriz de dos columnas,
%              la 1ª son los nodos, y la 2ª los valores correspondientes
%        indice = poner siempre con valor "1"
nodos=nodos_valores(:,1);  long=length(nodos);
if long==1
   pol=nodos_valores(1,2);
   break
else
   coef=difer_pr(nodos_valores(2:long,:),indice+1)-
        difer_pr(nodos_valores(1:long-1,:),indice);
end
ptemp=dif_prog(nodos_valores(1:long-1,:),h,indice);

s_sobre_k=over([1 -nodos(1)],long-1,h);
pol=(coef*s_sobre_k)+[0 ptemp];
```

Finalmente se lista a continuación el código de algunas de las funciones auxiliares, como **over.m**, utilizada por el mérodo de Newton para calcular un número sobre otro, o **multi_p.m**, empleado por la mayoría de los métodos para la multiplicación de polinomios:

```
function v=over(n,m,h)
%  calcula un numero sobre otro
%     ( s )
%     ( k )
%        El valor de "s" y "k" no debe incluir el denominador "h"
mn=m;     temp_m=1;
while m~=0
   temp_m=temp_m*m;    m=m-1;
end;
num=n(2);    count=mn;    temp_n=1;
while count~=0
   temp_n=multi_p(temp_n,[1 num]);
   num=num-h;    count=count-1;
end;
temp_n=temp_n/h^mn;
v=temp_n/temp_m;
```

```
function p=multi_p(p1,p2)
%  multiplica los polinomios p1 * p2
lp1=length(p1);       lp2=length(p2);
if (p1==1)&(lp1==1)
   p=p2;    break
end
if (p2==1)&(lp2==1)
   p=p1;    break
end
p=zeros(1,(lp1+lp2)-1);
if lp1~=1
   temp=multi_p(p1(1:lp1-1),p2);
   temp2=[temp 0];
   prod_pol=p1(lp1)*p2;
   long_c=length(temp2)-length(prod_pol);
   p=zeros(1,long_c);
   p=[p prod_pol];
   p=p+temp2;
else
   p=p1(lp1)*p2;
end
```

· Resolución de la práctica

Para la resolución de la práctica se implementados dos funciones que realizan automáticamente todos los pasos necesarios hasta la representación de la función original y de los distintos polinomios de interpolación para un número de nodos determinados como parámetros.

Estas funciones se encargan de calcular tanto los nodos como los valores asociados a los mismos, y de llamar al método deseados para el cálculo del polinomio de interpolación, así como de la representación gráfica de dicho polígono.

Las funciones se denominan **graf_eq.m** y **grf_cheb.m**, las cuales se encargan respectivamente de interpolación mediante nodos equidistantes y nodos de Chebyshev. con el fin de conseguir una implementación lo más general y modula posible, las funciones anteriores llaman a otras dos funciones que se encargan de calcular el polinomio propiamente dicho, y se denominan **equidist.m** y **chebyshe.m**, por lo cual cambiar el método empleado para dicho cálculo se reduce a cambiar una línea en estos ficheros.

Para el cálculo de los puntos de Chebyshev también sea implementado un pequeño fichero, denominado **cheby_ps.m**, el cual se encarga de calcular los puntos óptimos para la interpolación en un intervalo arbitrario, es decir, también se encarga de realizar la traslación del intervalo si fuese necesario.

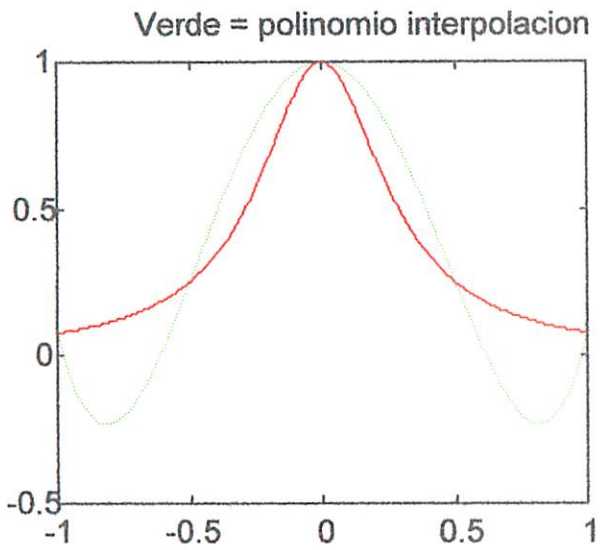
Para la estimación de los errores también sean implementados sendas funciones que automatizan todo el proceso, y se denominan **err_eq.m** y **err_cheb.m**, las cuales representan las gráficas de errores para los nodos equidistantes y los nodos de Chebyshev. ambas funciones se apoyan en otra función auxiliar denominada **err_max.m** que calcula el error máximo cometido por un determinado polinomio de interpolación, pasado como parámetro, dentro de un intervalo también pasado como parámetro.

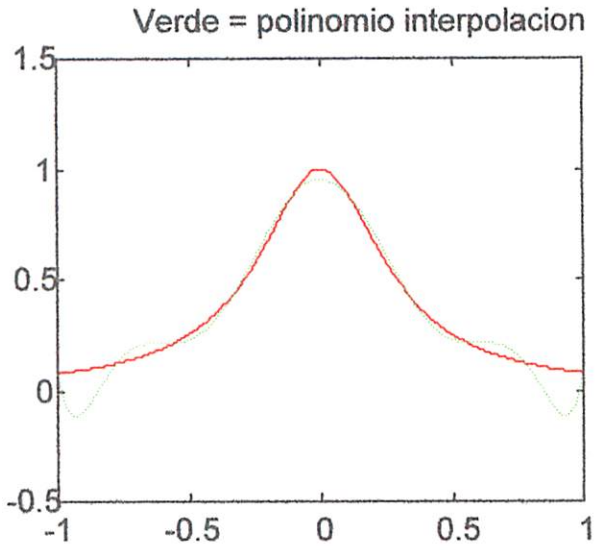
```
function graf_eq(xi,xf,vector_de_nodos)
%   >> graf_eq(xi,xf,vector_de_nodos)
%   dibuja graficas comparativas entre la funcion original func(x) y las
%   diferentes aproximaciones para unos determinados nodos, definidos mediante
%   un vector de longuitud par. %
%       Los nodos son calculados mediante la equidistancia entre los mismos.
%
%   p.e: vector_de_nodos=[6 10 16 24]
%          representa 4 graficas, con la funcion original y la aproximaciones
%          para 6, 10, 16 y 24 nodos.
nodos=vector_de_nodos;  n_grf=length(nodos);
if mod(n_grf,2)~=0
   break;
end
if n_grf==2
   g=21;
end
if n_grf==4
   g=22;
end
if n_grf==6
   g=32;
end
if n_grf==8
   g=42;
end
%  Numero de ptos usados para la representacion
nptos=500;  ptos=linspace(xi,xf,nptos);
for c=1:n_grf
   pol=equidist(xi,xf,nodos(c));
   subplot(10*g+c),plot(ptos,func(ptos),'r-',
                        ptos,polyval(pol,ptos),'g:'),
   title('Verde = polinomio interpolacion');
end
```

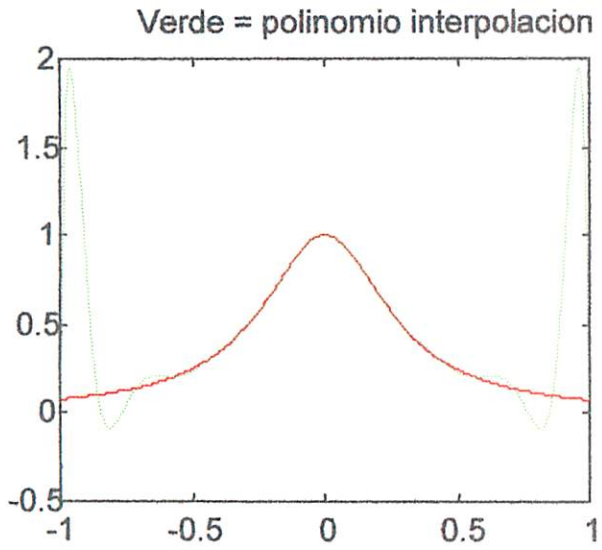
```
function pol=equidist(xi,xf,n_nodos)
%  devuelve el polinomio de interpolacion de grado "n"=n_nodos-1,
%     usando la funcion de Newton (diferencias progresivas)
xs=linspace(xi,xf,n_nodos);
ys=func(xs);
pol=pol_dpro([xs' ys'])
```

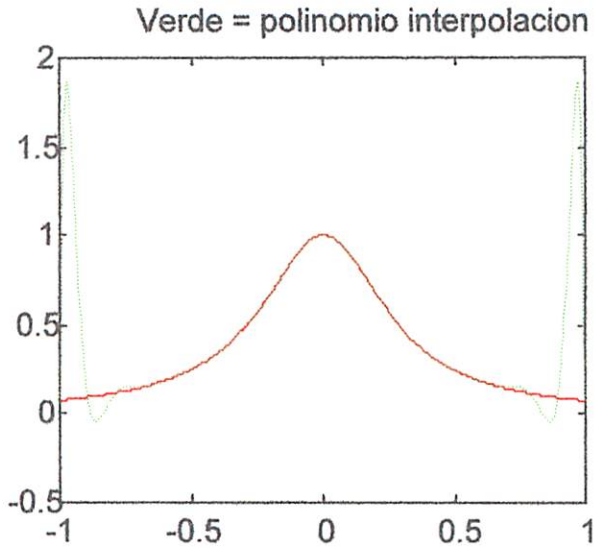
Por lo tanto, el realizar un gráfico comparativo en el intervalo [-1,1] entre la función original y el polinomio de interpolación para 5, 10, 15 y 20 nodos equidistantes se reduce a una orden del tipo:

```
» graf_eq(-1,1,[5 10 15 20])
```









Para el caso de nodos no equidistantes, es decir, para el caso de los nodos de Chebyshev, los programas que resuelven la práctica son:

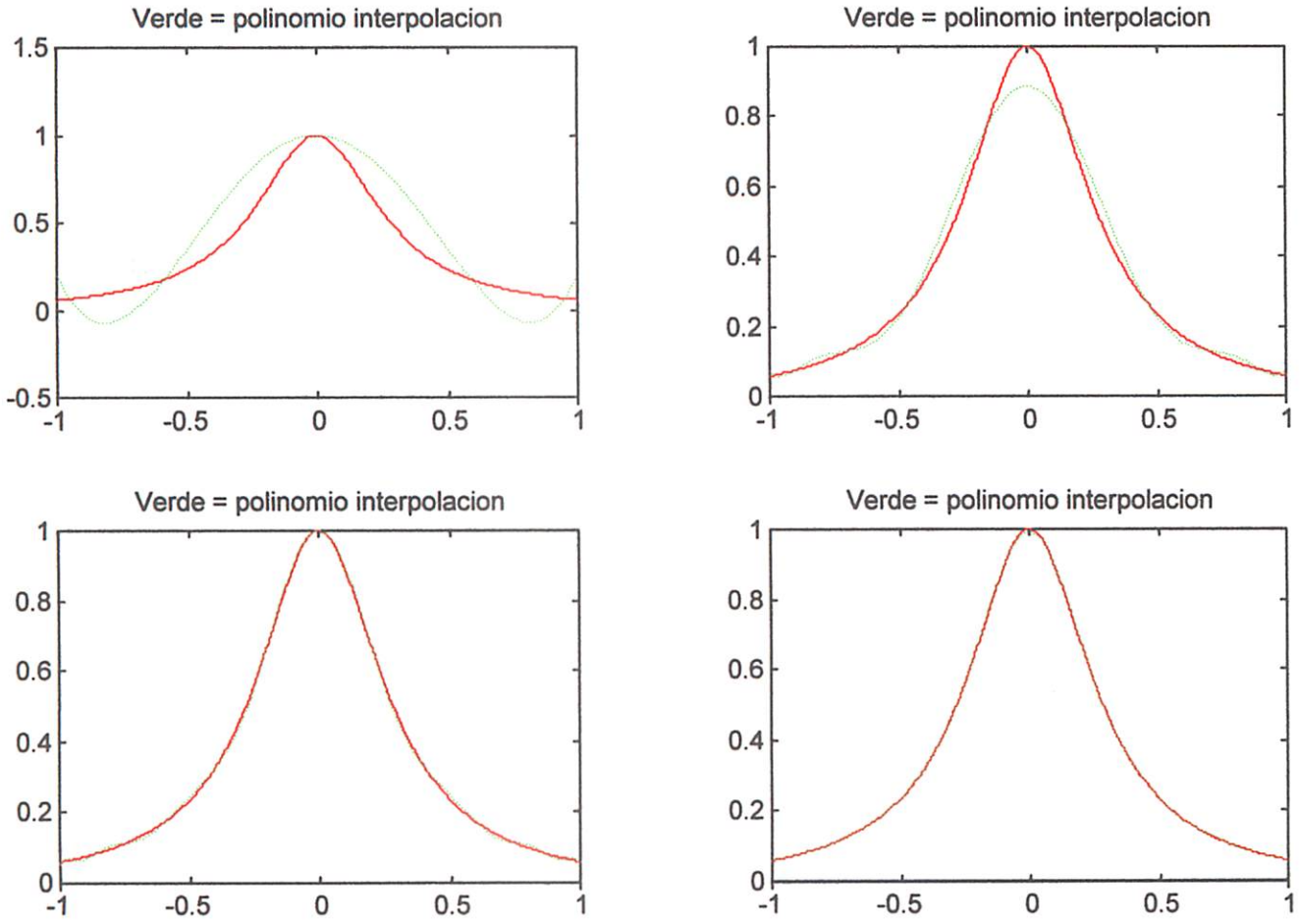
```
function grf_cheb(xi,xf,vector_de_nodos)
%  >> grf_cheb(xi,xf,vector_de_nodos)
%  dibuja graficas comparativas entre la funcion original func(x) y las
%  diferentes aproximaciones para unos determinados nodos, definidos mediante
%  un vector de longuitud par. %
%     Los nodos son calculados mediante las soluciones del polinomio de
%  Chebychev del grado correspondiente.
nodos=vector_de_nodos;  n_grf=length(nodos);
if mod(n_grf,2)~=0
   break;
end
if n_grf==2
   g=21;
end
if n_grf==4
   g=22;
end
if n_grf==6
   g=32;
end
if n_grf==8
   g=42;
end
%  Numero de ptos usados para la representacion
nptos=500;  ptos=linspace(xi,xf,nptos);
for c=1:n_grf
   pol=chebyshe(xi,xf,nodos(c));
   subplot(10*g+c),plot(ptos,func(ptos),'r-',
                        ptos,polyval(pol,ptos),'g:'),
   title('Verde = polinomio interpolacion');
end
```

```
function pol=chebyshe(xi,xf,n_nodos)
%  devuelve el polinomio de interpolacion de grado "n"=n_nodos-1,
%     usando las Diferencias divididas para los nodos de Chebyshev
xs=cheby_ps(xi,xf,n_nodos);
ys=func(xs);
pol=pol_ddif([xs' ys']);
```

```
function [ptos,M_xy]=cheby_ps(xi,xf,n_ptos)
%  ptos=cheby_ps(xi,xf,n_ptos)  devuelve n_ptos solución del polinomio
%     de Chebyshev para n_ptos, no limitado al intervalo [-1,1],
%     sino trasladado al intervalo [xi,xf]
n=(xf-xi)/2;   m=(xf+xi)/2;

x=zeros(1,n_ptos);   ptos=zeros(1,n_ptos);

den=2*(n_ptos);
for c=1:n_ptos
   x(c)=cos(((2*c-1)*pi)/den);
end
ptos=x*n+m;

M_xy=zeros(n_ptos,2);
M_xy(:,1)=ptos';
M_xy(:,2)=func(ptos)';
```

El resultado para 5, 10, 15 y 20 nodos de Chebyshev está representado en la serie de gráficas siguientes, considerando el intervalo de estudio [-1,1]:

```
» grf_cheb(-1,1,[5 10 15 20])
```

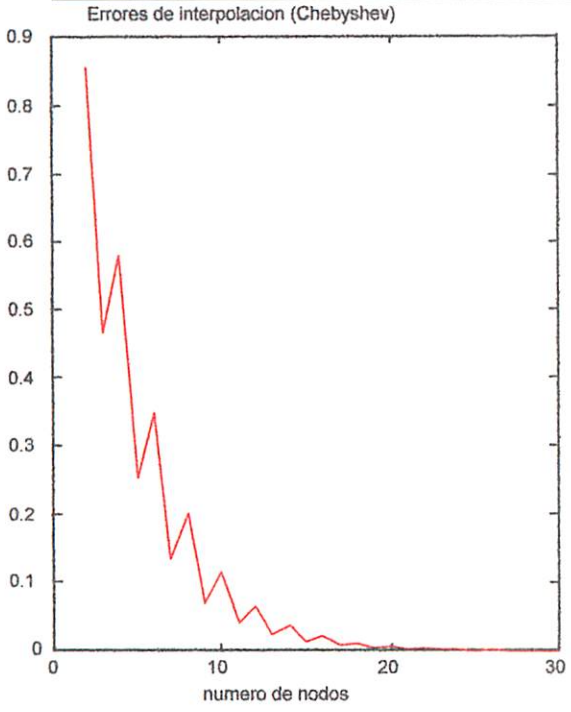


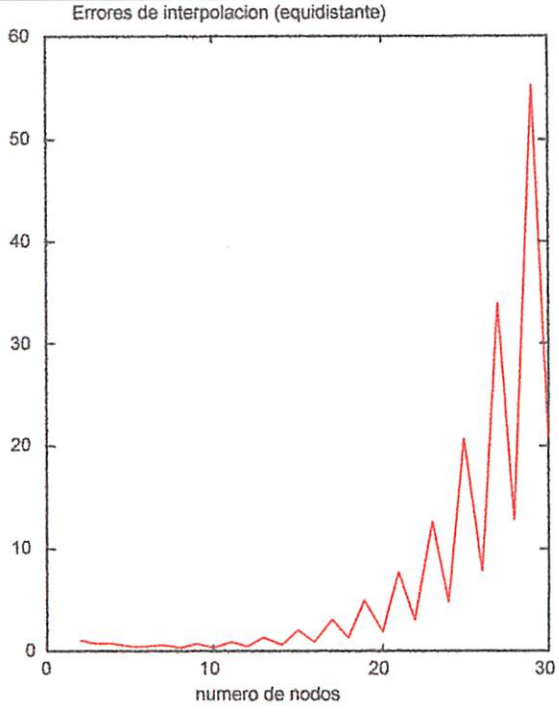
Para finalizar la resolución de la práctica, se presentan a continuación los códigos de las funciones de cálculo de errores para diferentes números de nodos, tanto para el caso de nodos equidistantes como Chebyshev:

```
function error=err_max(xi,xf,polinomio,n_ptos)
% devuelve el error MAX. del polinomio de interpolacion pasado como parametro
%      que debe estar definido como lo hace la funcion de MATLAB polyfit
inc=(xf-xi)/n_ptos;
error=0;
for c=xi:inc:xf
   err=(abs(func(c)-polyval(polinomio,c)));
   if err>error
      error=err;
   end
end
```

```
function err_cheb(xi,xf,vector_de_nodos)
%   >> err_cheb(xi,xf,vector_de_nodos)
% dibuja la grafica de errores maximos entre la funcion original func(x) y las
% diferentes aproximaciones para unos determinados nodos, definidos mediante
% un vector de longuitud par
nodos=vector_de_nodos;  it=length(nodos);
errores=zeros(1,it);
%  Numero de ptos usados para la representacion
nptos=200;
for c=1:it
   pol=chebyshe(xi,xf,nodos(c));
   errores(c)=err_max(xi,xf,pol,nptos);   xs_err(c)=nodos(c);
end
plot(xs_err,errores,'r-'),title('Errores de interpolacion'),
xlabel('numero de nodos');
```

```
function err_eq(xi,xf,vector_de_nodos)
%   >> err_eq(xi,xf,vector_de_nodos)
% dibuja la grafica de errores maximos entre la funcion original func(x) y las
% diferentes aproximaciones para unos determinados nodos, definidos mediante
% un vector de longuitud par.
nodos=vector_de_nodos;     it=length(nodos);
errores=zeros(1,it);
%  Numero de ptos usados para la representacion
nptos=200;
for c=1:it
   pol=equidist(xi,xf,nodos(c));
   errores(c)=err_max(xi,xf,pol,nptos);   xs_err(c)=nodos(c);
end
plot(xs_err,errores,'r-'),title('Errores de interpolacion'),
xlabel('numero de nodos');
```





Como se puede apreciar en las gráficas, mientras que para nodos equidistantes el error aumenta conforme se incrementa el número de nodos, debido al efecto en los extremos, para los nodos de Chebyshev el efecto es inverso, es decir, que el error cometido disminuye cuanto mayor es el número de nodos utiizados.

Análisis Numérico I:

Derivación e Integración aproximadas

Alumno:

Fernanso Sanz Clavijo

Asignatura:

Análisis Numérico I

Práctica 4:

Derivación e Integración aproximadas

Indice

· Enunciado de la Práctica

El objetivo de esta práctica es desarrollar una serie de métodos numéricos para el cálculo de las derivadas en un punto y de las integrales definidas o en un intervalo. Y lógicamente la aplicación de los métodos desarrollados a algunas funciones de ejemplo.

Se desarrollarán dos métodos, uno para el cálculo de las derivadas y otro para el cálculo de la integral, y a partir de estos métodos se extenderá mediante la extrapolación de los mismos, para obtener una forma más rápida de alcanzar los resultados.

· Derivación mediante Richardson

Fundamento teórico

Los métodos de derivación de Richardson se basan en los desarrollos de Taylor para funciones, más concretamente el método para la primera derivada se obtiene a partir de los siguientes desarrollos:

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2!} \cdot f''(x) + \frac{h^3}{3!} \cdot f'''(x) + \ldots\ldots$$

$$f(x-h) = f(x) - h \cdot f'(x) + \frac{h^2}{2!} \cdot f''(x) + \frac{h^3}{3!} \cdot f'''(x) + \ldots\ldots$$

$$f(x+h) - f(x-h) = 2h \cdot f'(x) + 2 \cdot \frac{h^3}{3!} \cdot f'''(x) + \ldots\ldots$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + k_1 \cdot h^2 + k_2 \cdot h^4 + k_3 \cdot h6$$

$$M = N(h) + Error$$

como se puede observar mediante este método obtenemos una aproximación de orden O(h2).

El cálculo de la segunda derivada mediante los desarrollos de Taylor se realiza de manera análoga:

□□ $f(x+h) = f(x) + h \cdot f'(x) +$ □ EC □□ EC \F(h2;2!) □ $\cdot f''(x) +$ □ EC □□ EC \F(h^3;3!) $\cdot f'''(x) + \ldots\ldots$

$$f(x-h) = f(x) - h \cdot f'(x) + \frac{h^2}{2!} \cdot f''(x) + \frac{h^3}{3!} \cdot f'''(x) + \ldots\ldots$$

$$f(x+h) + f(x-h) = 2 \cdot f(x) + h^2 \cdot f''(x) + \ldots\ldots$$

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + k_1 \cdot h^2 + k_2 \cdot h^4 + k_3 \cdot h^6$$

$$M = N(h) + Error$$

que como nuevamente podemos observar que es una aproximación de orden O(h2), pero con un término inicial distinto.

Metodología

Los métodos de derivación "puros" se basan como se ha dicho en los desarrollos previamente calculados, con lo cual sus algoritmos se apoyan en estas formulas generales, ya que decrementando el valor de "h" se reduce el error cometido. Por lo tanto se irán calculando sucesivas aproximaciones a la solución reduciendo el valor de "h" a la mitad en cada iteración.

Implementación

El algoritmo implementado, **richard.m,** recibe como parámetros principales el punto en el que desea calcular la derivada, así como un valor inicial de "h". Y realiza tantas iteraciones como sea necesario para alcanzar la tolerancia deseada, o sobrepasar las iteraciones máximas predefinidas.

Como criterio de paro se consideran en que dos iteraciones consecutivas se diferencien en menos que la tolerancia, o que se sobrepase el número máximo de iteraciones predefinidas.

```
function [s,it]=richard(p0,h,TOL,n_it)
%   Calcula tantas iteraciones como indica "n_it" del método de Richardson, o
%  hasta conseguir una tolerancia de "TOL" entre dos iteraciones consecutivas,
%  en el punto p0. Usando un incremento inicial = "h" (pequeño).
%   Version para la 1ª derivada:      [s,it]=richard(p0,h,TOL,n_it)
x=p0;
s(1)=(f(x+h)-f(x-h))/(2*h);

it=1; s1=s; no_salir=1;
while (it<n_it) & no_salir
   h=h/2;
   s2=(f(x+h)-f(x-h))/(2*h);
   it=it+1;
   if abs(s2-s1)<=TOL
      no_salir=0;
   end
   s(it)=s2;    s1=s2;
end
```

El algoritmo para segunda derivada, **richard2.m,** es idéntico al empleado anteriormente, con la única salvedad de que la fórmula general es diferente:

```
function [s,it]=richard2(p0,h,TOL,n_it)
%   Calcula tantas iteraciones como indica "n_it" del método de Richardson, o
%  hasta conseguir una tolerancia de "TOL" entre dos iteraciones consecutivas,
%  en el punto p0. Usando un incremento inicial = "h" (pequeño).
%   Version para la 2ª derivada:      [s,it]=richard2(p0,h,TOL,n_it)
x=p0;
s(1)=(f(x+h)+f(x-h)-2*f(x))/(h^2);

it=1; s1=s; no_salir=1;
while (it<n_it) & no_salir
   h=h/2;
   s2=(f(x+h)+f(x-h)-2*f(x))/h^2;
   it=it+1;
   if abs(s2-s1)<=TOL
      no_salir=0;
   end
   s(it)=s2;    s1=s2;
end
```

· Extrapolación de Richardson

Fundamento teórico

El fundamento de la extrapolación aplicada al método de Richardson, es el mismo que el de cualquier otra extrapolación, es decir, se basa en obtener una nueva aproximación a partir de dos aproximaciones previamente calculadas.

Concretamente para el caso de ambas derivadas, como el orden del error es el mismo e igual a los números pares comenzando desde el dos, el método de la extrapolación será el mismo y su desarrollo es el siguiente:

$$f'(x) = f''(x) = \frac{2^i \cdot N^k(h/2) - N^k(h)}{2^{i-1}}, \qquad i = 2, 4, 6, 8, ...$$

Metodología

La forma de actuación de algoritmo de extrapolación es la siguiente:

- inicialmente calcula una aproximación mediante el método de Richardson originario.
- a continuación calcula nuevas aproximaciones, por cada nuevo aproximación del método original
- calcular a tantas aproximaciones por extrapolación como pueda.
- finalmente devolverá la aproximación deseada o en su defecto la más cercana encontrada, para el número de iteraciones pedidos por el usuario.

Implementación

El algoritmo implementado, **xrichard.m**, se basa en una matriz cuya diagonal conteniendo los mejores valores de aproximación obtenido mediante extrapolación.

Inicialmente se definirá el polinomio de error para el método en concreto, en este caso es el mismo para la primera derivada y para la segunda, y se calcula el primer valor aproximación mediante la función adecuada, ya sea para la primera o segunda derivada.

Después se entrara un bucle en el cual se irá calculando una nueva fila, serie de aproximación, de la matriz por cada nuevo valor de "h", que en cada iteración se reduce a su mitad.

Como criterios de paro se han utilizado el número de iteraciones máximas deseadas, el que la diferencia de dos valores de la diagonal esté por debajo de la tolerancia deseada y por último se ha utilizado una especie de mecanismo de seguridad que memoriza la mejor aproximación encontrada, y si en algún momento las aproximaciones comienzan a diverger debido a los efectos de la cancelación, se concluirá con la ejecución y se devolverá el valor más aproximado antes de comenzar la divergencia.

```
function s=xrichard(p0,TOL,h)
%   Calcula la extrapolacion para el método de Richardson, en el punto p0.
%  Usando una tolerancia = TOL, y un incremento inicial = "h" (pequeño).
%
%       s=xrichard(p0,TOL,h)
pol_error=0:2:2*log(1/TOL);
%tabla(1,1)=richard(p0,h,TOL,1);      % primera derivada
tabla(1,1)=richard2(p0,h,TOL,1);      % segunda derivada
k=2; fin=0; aprox=1;
while ~fin
   h=h/2;
   %tabla(k,1)=richard(p0,h,TOL,1);      % primera derivada
   tabla(k,1)=richard2(p0,h,TOL,1);      % segunda derivada
   for j=2:k
      %coef=2^j;
      coef=2^pol_error(j);
      tabla(k,j)=tabla(k,j-1)+(tabla(k,j-1)-tabla(k-1,j-1))/(coef-1);
   end
```

· Integración mediante Romberg

Fundamento teórico

Del método para el cálculo de la integral aproximada de Romberg, se basa en la fórmula de los trapecios, es decir, en la aproximación del área encerrada por la curva mediante el área constituida por una suma de trapecio que se aproxima a la función en pequeños intervalos.

$$\int_{a}^{a+h} f(x)\cdot dx \approx \frac{h}{2}\cdot [f(a)+f(a+h)]$$

Teóricamente si dividimos el intervalo en el cual deseamos calcular la integral en infinitos trapecio, la suma del área de todos estos trapecios debería ser igual a la suma encerrada por la curva en este intervalo. Pero tendremos que conformarnos con dividirlo en n partes.

$$h=\frac{b-a}{n} \rightarrow \int_{a}^{b} f(x)\cdot dx \approx \frac{h}{2}\cdot [f(x_0)+2\cdot [f(x_1)+.....+f(x_{n-1})]+f(x_n)] + O(h^2)$$

$$\int_{a}^{b} f(x)\cdot dx \approx T(h) + \text{Error} : O(h^2)$$

Metodología

Puesto que lógicamente es imposible dividir el intervalo en infinitos valores, el método de integración aproximada mediante trapecios realiza la división de intervalo en un número arbitrario de valores y calcula el valor aproximan de la integral para dicho número de trapecios.

Para conseguir mejores aproximaciones al valor de la integral en dicho intervalo se va sucesivamente de duplicando el número de divisiones del intervalo de trabajo.

Como se puede de en algoritmo de integración posee un error de orden O(h2), más concretamente tiene el mismo polinomio de error que los métodos de derivación de Richardson.

Implementación

El algoritmo implementado, **romberg.m,** recibe como parámetros principales el intervalo en el que desea calcular la integral, así como un valor inicial del número de divisiones "n" de dicho intervalo. Y realiza tantas iteraciones, sea necesario para alcanzar la tolerancia deseada, o sobrepasar el número de iteraciones máximo predefinido.

Como criterio de paro se consideran en que dos iteraciones consecutivas se diferencien en menos que la tolerancia, o que las aproximaciones comiencen a diverger, utilizando el método de seguridad anteriormente comentado.

```
function [s,it]=romberg(p1,p2,n,TOL,n_it)
%  Calcula tantas iteraciones como indica "n_it" del método de Romberg, o
%  hasta conseguir una tolerancia de "TOL" entre dos iteraciones consecutivas,
%  para el intervalo [p1,p2].
%  Usando un incremento inicial  h = (p2-p1)/n
%  Version para la integral:      [s,it]=romberg(p1,p2,n,TOL,n_it)
h=(p2-p1)/n;
it=1; x=p1;

temp=f(x); x=x+h;
while (x<p2)
   temp=temp+2*f(x);    x=x+h;
end
temp=temp+f(x);
s1=(h/2)*temp;

s=s1; no_salir=1; aprox=1;
while (it<n_it) & no_salir
   h=h/2;
   it=it+1;

   x=p1; temp=f(x);  x=x+h;
   while (x<p2)
      temp=temp+2*f(x);       x=x+h;
   end
   temp=temp+f(x);
   s2=(h/2)*temp;

   s(it)=s2;
   aprox=min(abs(s2-s1),aprox);
   if abs(s2-s1)<=TOL
      no_salir=0;       s(it)=s2;
   else if (aprox<abs(s2-s1))
         no_salir=0;
         disp('No es posible aproximar tantos decimales, la solucion más proxima es:');
         s(it)=s1
        end
   end
   s1=s2;
end
```

· Extrapolación de Romberg

Fundamento teórico

El fundamento de la extrapolación aplicada al método de Romberg, al igual que en el caso anterior, es el mismo que el de cualquier otra extrapolación, es decir, se basa en obtener una nueva aproximación a partir de dos aproximaciones previamente calculadas.

Concretamente para el caso de la integral, como el orden del error es el mismo que en las derivadas, e incluso tienen el mismo polinomio de error, es decir, los números pares comenzando desde el 2, el método de la extrapolación es por tanto el mismo que anteriormente y su desarrollo es el siguiente:

$$\int_a^b f(x)\cdot dx \approx T(h) + + k_1\cdot h^2 + k_2\cdot h^4 + k_3\cdot h^6$$

$$\int_a^b f(x)\cdot dx \approx \frac{2^i\cdot N^{k^*}(h/2) - N^{k^*}(h)}{2^{i-1}}, \qquad i = 2, 4, 6, 8, ...$$

Metodología

la forma de actuación de algoritmo de extrapolación es la siguiente:

- inicialmente calcula una aproximación mediante el método de Romberg originario.
- a continuación calcular nuevas aproximaciones por Romberg, y por cada nueva aproximación calcula tantas aproximaciones por extrapolación como le sea posible hasta el momento.
- finalmente devolverá la aproximación deseada o en su defecto la más cercana encontrada para el número de iteraciones máximo permitido.

Implementación

El algoritmo implementado es exactamente el mismo al usado en la derivación, y como se dijo se basa en una matriz cuya diagonal continente las aproximaciones optimas por cada iteración del método original.

Inicialmente se definirá el polinomio de este error para el método en concreto y se calculará el primer valor de aproximación mediante la función de Romberg.

Después se entra en un bucle en el cual se irá calculando una nueva fila por cada nuevo valor de "n", que en cada iteración se duplica.

Como criterios de paro se han utilizado: el número de iteraciones máximas deseadas, el que la diferencia de dos valores de la diagonal esté por debajo de la tolerancia deseada y el mecanismo de seguridad de memorizar la mejor aproximación encontrada, para evitar la divergencia por cancelación.

```
function s=xromberg(p1,p2,TOL,n)
%  Calcula la extrapolacion para el método de Romberg, en el intervalo [p1,p2].
%  Usando una tolerancia = TOL, y tanto subintervalos como indique "n",
%  dando como resultado un incremento inicial "h",
%
%         h = (p2 - p1) / n
%
%      s=xromberg(p1,p2,TOL,n)

pol_error=0:2:2*log(1/TOL);
tabla(1,1)=romberg(p1,p2,n,TOL,1);      % Integral por Romberg

k=2; fin=0;aprox=1;
while ~fin
   n=2*n;
   tabla(k,1)=romberg(p1,p2,n,TOL,1);      % Integral por Romberg
   for j=2:k
      coef=2^pol_error(j);
      tabla(k,j)=tabla(k,j-1)+(tabla(k,j-1)-tabla(k-1,j-1))/(coef-1);
   end

   aprox=min(abs(tabla(k,j)-tabla(k-1,j-1)),aprox);
   if abs(tabla(k,j)-tabla(k-1,j-1))<=TOL
      fin=1;
      s=tabla(k,j);
   else if (abs(tabla(k,1)-tabla(k-1,1))<=TOL) |
           (aprox<abs(tabla(k,j)-tabla(k-1,j-1)))
         fin=1;
         disp('No es posible aproximar tantos decimales, la solucion más proxima es:');
         s=tabla(k-1,j-1)
        end
   end

   k=k+1;
end
```

· Ejemplos

Utilizando como función original f = sin (x), podemos pedir por ejemplo su derivada y su segunda derivada en π / 6, mediante:

```
» [solucion,it]=richard(pi/6,0.1,10^-6,12)

      solucion = 0.86602531568776,        it = 8

» [solucion,it]=richard2(pi/6,0.1,10^-6,12)

      solucion = -0.49999989828393,       it = 7
```

O mediante los métodos de extrapolación:

```
» [solucion,it]=xrichard(pi/6,0.1,10^-6,12)

      solucion = 0.86602540378176,        it = 3

» [solucion,it]=richard2(pi/6,0.1,10^-6,12)

      solucion = -0.49999999999953,       it = 3
```

Igualmente para la integral utilizaremos como función original f = sin (x), y pediremos por ejemplo su integral entre [0, π / 4], [0, π / 2], [0, π], [0, 2·π], mediante:

```
» [solucion,it]=romberg(0,pi/4,10,10^-6,12)
      solucion = 0.29289307178279,        it = 6
» [solucion,it]=romberg(0,pi/2,10,10^-6,12)
      solucion = 0.99999799202304,        it = 7
» [solucion,it]=romberg(0,pi,10,10^-6,12)
      solucion = 1.99998393623754,        it = 7
» [solucion,it]=romberg(0,2*pi,10,10^-6,12)
      solucion = -0.12566885569729·10^-15,  it = 2
```

O mediante los métodos de extrapolación:

```
» [solucion,it]=xromberg(0,pi/4,10^-6,10)
      solucion = 0.29289321881342,        it = 3
» [solucion,it]=xromberg(0,pi/2,10^-6,10)
      solucion = 0.99999999999223,        it = 3
» [solucion,it]=xromberg(0,pi,10^-6,10)
      solucion = 2.00000000000003,        it = 4
» [solucion,it]=xromberg(0,2*pi,10^-6,10)
      solucion = -1.419103787650061·10^-16, it = 2
```

Análisis Numérico I:

Valores iniciales para ecuaciones diferenciales

Alumno: **Fernanso Sanz Clavijo**

Asignatura: **Análisis Numérico I**

Práctica 5: **Valores iniciales para ecuaciones diferenciales**

Indice

· Enunciado de la Práctica

El objetivo de esta práctica es desarrollar varios métodos para la resolución de ecuaciones diferenciales apatir de un valor inicial conocido de la ecuación primitiva, y logicamente conocida la ecuación diferencial.

Se desarrollarán varios métodos, que son los siguientes: Euler, Gragg, Runge-Kutta, Adam-Bashforth y Adam Moulton, así como la extrapolación de algunos de ellos, los de convergencia más lenta.

· Método de Euler

Fundamento teórico

Se trata de un método iterativo unipaso, que se basa en el desarrollo de Taylor de la función primitiva, y cuyo esquema podría resumirse en las siguientes reglas:

Valor inicial: $W_0 = \alpha$ (conocido)

Forma general: $W_{i+1} = W_i + h \cdot f(x_i, W_i)$

Este método aproximativo tiene un error de orden $O(h^1)$.

Metodología

El método de Euler "puro" se basa como se ha dicho en el esquema iterativo previamente comentado, con lo cual su algoritmo se apoya directamente en esta fórmula general, ya que decrementando el valor de "h" se reduce el error cometido. Por lo tanto se irán calculando sucesivas aproximaciones a la solución reduciendo el valor de "h" a la mitad en cada iteración, si bien es un método de convergencia muy lenta.

Implementación

El algoritmo implementado, **euler.m,** recibe como parámetros principales el punto conocido de la función primitiva buscada y el valor de "x" en el que se desea conocer dicha primitiva , así como un valor inicial de "h". Y realiza tantas iteraciones como sea necesario para alcanzar la tolerancia deseada, o sobrepasar el número de iteraciones máximos permitido.

También se considerará como criterio de paro se consideran en que dos iteraciones consecutivas se diferencien en menos que la tolerancia.

```
function [s,it]=euler(xa,ya,h,x0,TOL,n_it)
%   Calcula la ecuacion primitiva de la diferencial contenida en "fprima" en f(x0),
%  conocido el punto de la funcion original f(xa)=ya, y con un incremento inicial de "h".
%  El parametro "n_it" indica el numero de iteraciones para distintos valores de "h".
%  Y "TOL" expresa la tolerancia deseada en el calculo del valor.
%
%   [s,it]=euler(xa,ya,h,x0,TOL,n_it)

if mod(x0,h)~=0
   disp('el punto en el que hacer la aproximacion debe ser "multiplo" de h,');
   disp('es decir, x0= xa + N * h');    break;
end

it=0; no_salir_ext=1;   y_ant_euler=inf;
while (it<n_it) & no_salir_ext

   no_salir=1; y_ant=ya;   x_ant=xa;
   x=xa+h;
   while no_salir

      y=y_ant+h*fprima(x_ant,y_ant);

      y_ant=y;    x_ant=x;    x=x+h;

      if x_ant>(x0-h/2)
         no_salir=0;
      end
   end

   if abs(y-y_ant_euler)<=TOL
      no_salir_ext=0;
   end;

   h=h/2;
   it=it+1; y_ant_euler=y;   s(it)=y;
end
```

· Método de Gragg

Fundamento teórico

Se trata también de un método iterativo, si bien necesita del método de Euler para calcular el primer paso, su progresión es la siguiente:

Valor inicial:	W_0	$= \alpha$	(conocido)
1er paso:	W_1	$= W_0 + h \cdot f(x_0, W_0)$	(Euler)
Forma general:	W_{i+1}	$= W_{i-1} + 2h \cdot f(x_i, W_i)$	

Este método aproximativo mejora el error, ya que tiene un error de orden $O(h^2)$.

Metodología

El método de Gragg "puro" se basa en el esquema iterativo anterior, con lo cual su algoritmo se apoya en el de Euler y en la fórmula general, decrementando el valor de "h" para conseguir sucesivas reducciones del error. Por lo tanto se irán calculando aproximaciones a la solución reduciendo el valor de "h" a la mitad en cada iteración.

Implementación

El algoritmo implementado, **gragg.m,** recibe como parámetros principales el punto conocido de la función primitiva buscada y el valor de "x" en el que se desea conocer dicha primitiva , así como un valor inicial de "h".

Inicialmente calcula un primer valor de cada serie mediante Euler, y realiza tantas iteraciones de Gragg como sea necesario para alcanzar la tolerancia deseada, o sobrepasar el número de iteraciones máximos permitido.

También se considerará como criterio de paro se consideran en que dos iteraciones consecutivas se diferencien en menos que la tolerancia.

```
function [s,it]=gragg(xa,ya,h,X,TOL,n_it)
%   Calcula la ecuacion primitiva de la diferencial contenida en "fprima" en f(x0),
%  conocido el punto de la funcion original f(xa)=ya, y con un incremento inicial de "h".
%  El parametro "n_it" indica el numero de iteraciones para distintos valores de "h".
%  Y "TOL" expresa la tolerancia deseada en el calculo del valor.
%
%   [s,it]=gragg(xa,ya,h,x0,TOL,n_it)

if mod(X,h)~=0
   disp('el punto en el que hacer la aproximacion debe ser "multiplo" de h,');
   disp('es decir, x0= xa + N * h');
   break;
end

it=0; no_salir_ext=1;   y_ant_gragg=inf;
while (it<n_it) & no_salir_ext

   y0=ya;
   x1=xa+h;
   y1=euler(xa,ya,h,x1,TOL,1);    % w1 mediante Euler
   y=y1;

   no_salir=1;
   while no_salir & (x1<X)  % comprueba si necesita Gragg, o solo Euler

      y=y0+2*h*fprima(x1,y1);

      y0=y1;      y1=y;      x1=x1+h;

      if x1>(X-h/2)
         no_salir=0;
      end
   end

   if abs(y-y_ant_gragg)<=TOL
      no_salir_ext=0;
   end;


   h=h/2;
   it=it+1;   y_ant_gragg=y;   s(it)=y;
end

if abs(x1-(xa+h))<h/2        % no se ha usado Gragg, solo Euler
   s=y;
end
```

· Método de Runge-Kutta

Fundamento teórico

Se trata también de un método iterativo unipaso, de orden $O(h^4)$. Cuyo esquema iterativo es el siguiente:

Valor inicial: $W_0 = \alpha$ (conocido)

Forma general:

$$W_{i+1} = W_i + \frac{k_1 + 2\cdot k_2 + 2\cdot k_3 + k_4}{6}$$

$$k_1 = h \cdot f(x_i, W_i)$$

$$k_2 = h \cdot f(x_i + h/2, W_i + k_1/2)$$

$$k_3 = h \cdot f(x_i + h/2, W_i + k_2/2)$$

$$k_4 = h \cdot f(x_i + h, W_i + k_3)$$

Metodología

El método de Runge-Kutta "puro" se basa en el esquema iterativo anterior, con lo cual su algoritmo se apoya en la fórmula general descrita, decrementando el valor de "h" para conseguir sucesivas reducciones del error, más concretamente reduciendola a la mitad en cada iteración.

Implementación

El algoritmo implementado, **kutta.m,** recibe como parámetros principales el punto conocido de la función primitiva buscada y el valor de "x" en el que se desea conocer dicha primitiva , así como un valor inicial de "h".

Realiza tantas iteraciones como necesite para alcanzar la tolerancia deseada.
Considerará como criterio de paro el sobrepasar el número de iteraciones máximos permitido o bien que dos iteraciones consecutivas se diferencien en menos que la tolerancia prefijada.

```
function [s,it]=kutta(xa,ya,h,X,TOL,n_it)
   Calcula la ecuacion primitiva de la diferencial contenida en "fprima" en f(x0),
%  conocido el punto de la funcion original f(xa)=ya, y con un incremento inicial de "h".
%  El parametro "n_it" indica el numero de iteraciones para distintos valores de "h".
%  Y "TOL" expresa la tolerancia deseada en el calculo del valor.

%   [s,it]=kutta(xa,ya,h,x0,TOL,n_it)

if mod(X,h)~=0
   disp('el punto en el que hacer la aproximacion debe ser "multiplo" de h,');
   disp('es decir, x0= xa + N * h');    break;
end

it=0; no_salir_ext=1;   y_ant_kutta=inf;
while (it<n_it) & no_salir_ext

   y0=ya;   x0=xa;

   no_salir=1;
   while no_salir

      k1=h*fprima(x0,y0);              k2=h*fprima(x0+h/2,y0+k1/2);
      k3=h*fprima(x0+h/2,y0+k2/2);     k4=h*fprima(x0+h,y0+k3);

      y=y0+((k1+2*k2+2*k3+k4)/6);

      y0=y;    x0=x0+h;

      if x0>(X-h/2)
         no_salir=0;
      end
   end

   if abs(y-y_ant_kutta)<=TOL
      no_salir_ext=0;
   end;

   h=h/2;
   it=it+1;   y_ant_kutta=y;     s(it)=y;
end
```

· Método de Adam Bashforth

Fundamento teórico

Se trata también de un método iterativo de cuatro pasos, que utiliza como método de apoyo al de Runge-Kutta, ambos de orden $O(h^4)$. El esquema iterativo del método es el siguiente:

Valor inicial: $W_0 = \alpha$ (conocido)

Pasos iniciales: W_1, W_2, W_3 mediante Runge-Kutta

Forma general: $$W_{i+1} = W_i + \frac{h}{24} \cdot [\, 55 \cdot f_i - 59 \cdot f_{i-1} + 37 \cdot f_{i-2} - 9 \cdot f_{i-3} \,], \quad i = 3,4,\ldots n$$

$$f_j = f(x_j, W_j), \quad j = i,\ i-1,\ i-2,\ i-3$$

Metodología

El método de Bashforth "puro" se basa en el esquema iterativo anterior, con lo cual su algoritmo se apoya en la fórmula general descrita y el algoritmo de Runge-Kutta.

La forma de reducir el error, de orden $O(h^4)$, es decrementando el valor de "h", concretamente reduciendolo a la mitad en cada iteración.

Implementación

El algoritmo, **bash.m,** recibe como parámetros principales el punto conocido de la función primitiva buscada y el valor de "x" en el que se desea conocer dicha primitiva , así como un valor inicial de "h".

Inicialmente calcula las primeras iteraciones mediante Runge-Kutta, y posteriormente realiza tantas iteraciones del método en sí como necesite para alcanzar la tolerancia deseada.

Se considerarán como criterios de paro el sobrepasar el número de iteraciones máximos permitido o bien que dos iteraciones consecutivas se diferencien en menos que la tolerancia prefijada.

```
function [s,it]=bash(xa,ya,h,X,TOL,n_it)
%   Calcula la ecuacion primitiva de la diferencial contenida en "fprima" en f(x0),
%  conocido el punto de la funcion original f(xa)=ya, y con un incremento inicial de "h".
%  El parametro "n_it" indica el numero de iteraciones para distintos valores de "h".
%  Y "TOL" expresa la tolerancia deseada en el calculo del valor.
%   Empleando los metodos de Runge-Kutta para comenzar, y Adam-Bashforth para afinar.
%   [s,it]=bash(xa,ya,h,x0,TOL,n_it)
if mod(X,h)~=0
   disp('el punto en el que hacer la aproximacion debe ser "multiplo" de h,');
   disp('es decir, x0= xa + N * h');    break;
end
it=0; no_salir_ext=1;   y_ant_bash=inf;
while (it<n_it) & no_salir_ext
   x0=xa;      y0=ya;
   x1=x0+h;    y1=kutta(xa,ya,h,x1,TOL,1);      % w1 mediante Runge-Kutta
   x2=x1+h;    y2=kutta(xa,ya,h,x2,TOL,1);      % w2 mediante Runge-Kutta
   x3=x2+h;    y3=kutta(xa,ya,h,x3,TOL,1);      % w3 mediante Runge-Kutta
   if x1>=(X-h/2)          % la solucion ha sido encontrada por Runge-Kutta
      y=y1;
   else if x2>=(X-h/2)          y=y2;
        else if x3>=(X-h/2)     y=y3;   end
        end
   end
   if (x3>=X-h/2)        no_salir=0;
   else                  no_salir=1;    end

   while no_salir
      fi_3=fprima(x0,y0);          fi_2=fprima(x1,y1);
      fi_1=fprima(x2,y2);          fi = fprima(x3,y3);

      y=y3+((h/24)*(55*fi -59*fi_1 +37*fi_2 -9*fi_3));
      y0=y1;       y1=y2;       y2=y3;       y3=y;
      x0=x1;       x1=x2;       x2=x3;       x3=x3+h;

      if x3>(X-h/2)      no_salir=0;     end
   end

   if abs(y-y_ant_bash)<=TOL
      no_salir_ext=0;
   end;

   h=h/2;
   it=it+1;    y_ant_bash=y;    s(it)=y;
end
```

· Extrapolación de Euler, Gragg, Runge-Kutta y Adam Bashforth

Fundamento teórico

El fundamento de la extrapolación aplicada al los distintos métodos se basa en obtener una nueva aproximación a partir de dos aproximaciones previamente calculadas, como se puso de manifiesto en la práctica anterior.

Metodología

La forma de actuación de algoritmo de extrapolación para todos los métodos es la siguiente:

- · inicialmente calcula el polinomio de error correspodiente a cada método en concreto.
- · después calcula una primera aproximación mediante el método originario que se desea extrapolar.
- · a continuación calcula nuevas aproximaciones,
- · y por cada nueva aproximación del método original, calcular tantas aproximaciones por extrapolación como se puedan.
- · finalmente devolverá la aproximación deseada o en su defecto la más cercana encontrada, para el número de iteraciones pedidos por el usuario.

Implementación

Los algoritmo implementado, **xeuler.m, xgragg.m, xkutta.m** y **xbash.m**, se basan en una matriz cuya diagonal conteniene los mejores valores de aproximación obtenido mediante extrapolación para cada iteración.

Inicialmente se definirá el polinomio de error para el método en concreto y se calcula el primer valor de aproximación, a partir de punto inicial conocido, mediante la función adecuada. Después se entrara un bucle en el cual se irá calculando una nueva fila, o serie de aproximación, de la matriz por cada nuevo valor de "h", que en cada iteración se reduce a su mitad.

Como criterios de paro se han utilizado el número de iteraciones máximas deseadas, el que la diferencia de dos valores de la diagonal esté por debajo de la tolerancia deseada y por último se ha utilizado una especie de mecanismo de seguridad que memoriza la mejor aproximación encontrada, y si en algún momento las aproximaciones comienzan a diverger debido a los efectos de la cancelación, se concluirá con la ejecución y se devolverá el valor más aproximado antes de comenzar la divergencia.

Como muestra de los ficheros de extrapolación se lista el de Gragg, pero los cambios entre ellos son mínimos:

```
function [s,it]=xgragg(xa,ya,x0,TOL,h)
%   Calcula la extrapolacion para el método de Gragg, en el punto x=x0.
%  Usando una tolerancia = TOL, y un incremento inicial = "h" (pequeño).
%  Datos iniciales:  f("xa")="ya"
%       [s,it]=xgragg(xa,ya,x0,TOL,h)
pol_error=0:2:2*log(1/TOL);    fin=0;    k=2;  aprox=inf;
tabla(1,1)=gragg(xa,ya,h,x0,TOL,1);      % 1er valor mediante Gragg
while ~fin
   h=h/2;
   tabla(k,1)=gragg(xa,ya,h,x0,TOL,1);      % valores propios de Gragg
   for j=2:k
      coef=2^pol_error(j);
      tabla(k,j)=tabla(k,j-1)+(tabla(k,j-1)-tabla(k-1,j-1))/(coef-1);
   end
   aprox=min(abs(tabla(k,j)-tabla(k-1,j-1)),aprox);
   if abs(tabla(k,j)-tabla(k-1,j-1))<=TOL
      fin=1;    s=tabla(k,j);
   else if(abs(tabla(k,1)-tabla(k-1,1))<=TOL)|(aprox<abs(tabla(k,j)-tabla(k-1,j-1)))
         disp('No es posible aproximar tantos decimales, la solucion más proxima es:');
         fin=1;    s=tabla(k-1,j-1)
       end
   end
   k=k+1;    it=k-1;
end
```

· Método de Adam Moulton

Fundamento teórico

Se trata también de un método iterativo explícito de cuatro pasos, que utiliza como métodos de apoyo al de Runge-Kutta y al de Adam Bashforth, todos de orden $O(h^4)$.

El esquema iterativo de este método, predictor-corrector, es el que sigue:

Valor inicial: $W_0 = \alpha$ (conocido)

Pasos iniciales: W_1, W_2, W_3 mediante Runge-Kutta

Predicción: W_4^* mediante Adam Bashforth

Forma general: $$W_{i+1} = W_i + \frac{h}{24} \cdot [\, 9 \cdot f_{i+1} + 19 \cdot f_i - 5 \cdot f_{i-1} + 9 \cdot f_{i-2} \,], \qquad i = 3,4,\ldots n$$

$$f_j = f(x_j, W_j), \qquad j = i+1,\ i,\ i-1,\ i-2$$

Metodología

El método de Moulton "puro" se basa en el esquema iterativo anterior, con lo cual su algoritmo se apoya en la fórmula general descrita y en los algoritmos de Bashforth, como predictor, y de Runge-Kutta, para valores iniciales. La reducción del error, de orden $O(h^4)$, se realiza decrementando el valor de "h", más concretamente reduciendolo a la mitad en cada iteración.

Implementación

El algoritmo, **moulton.m,** recibe como parámetros principales el punto conocido de la función primitiva buscada y el valor de "x" en el que se desea conocer dicha primitiva , así como un valor inicial de "h".

Inicialmente calcula las primeras iteraciones mediante Runge-Kutta, y posteriormente realiza repetidas veces una predicción mediante Bashforth y su corrección mediante Adam Moulton, y así hasta tantas iteraciones del método en sí como necesite para alcanzar la tolerancia deseada.

Se considerarán como criterios de paro el sobrepasar el número de iteraciones máximos permitido o bien que dos iteraciones consecutivas se diferencien en menos que la tolerancia prefijada.

```
function [s,it]=moulton(xa,ya,h,X,TOL,n_it)
%   Calcula la ecuacion primitiva de la diferencial contenida en "fprima" en f(x0),
%  conocido el punto de la funcion original f(xa)=ya, y con un incremento inicial de "h".
%  El parametro "n_it" indica el numero de iteraciones para distintos valores de "h".
%  Y "TOL" expresa la tolerancia deseada en el calculo del valor.
%
%   Empleando los metodos de Runge-Kutta para comenzar, y
%  Adam-Bashforth para predecir y Adam-Moulton para correguir.
%
%   [s,it]=moulton(xa,ya,h,x0,TOL,n_it)
if mod(X,h)~=0
   disp('el punto en el que hacer la aproximacion debe ser "multiplo" de h,');
   disp('es decir, x0= xa + N * h');    break;
end

it=0; no_salir_ext=1;   y_ant_moulton=inf;
while (it<n_it) & no_salir_ext
   x0=xa;      y0=ya;
   x1=x0+h;    y1=kutta(xa,ya,h,x1,TOL,1);     % w1 mediante Runge-Kutta
   x2=x1+h;    y2=kutta(xa,ya,h,x2,TOL,1);     % w2 mediante Runge-Kutta
   x3=x2+h;    y3=kutta(xa,ya,h,x3,TOL,1);     % w3 mediante Runge-Kutta
   if x1>=(X-h/2)
      y=y1;
   else if x2>=(X-h/2)
           y=y2;
        else if x3>=(X-h/2)
                y=y3;                          end
        end
   end

   if (x3>=X-h/2)        no_salir=0;
   else                  no_salir=1;        end

   x4=x3+h;
   while no_salir

      % Prediccion mediante Bashforth (w4 >> explicito)
      y4=minibash(x0,y0,y1,y2,y3,h);

      fi_2=fprima(x1,y1);      fi_1=fprima(x2,y2);
      fi = fprima(x3,y3);      fiplus1=fprima(x4,y4);

      % Correccion de Adam-Moulton  (w4 >> implicito)
      y=y3+((h/24)*(9*fiplus1 +19*fi -5*fi_1 +fi_2));

      y0=y1;       y1=y2;       y2=y3;       y3=y;
      x0=x1;       x1=x2;       x2=x3;       x3=x4;
      x4=x4+h;
      if x3>(X-h/2)
         no_salir=0;
      end
   end

   if abs(y-y_ant_moulton)<=TOL
      no_salir_ext=0;
   end;

   it=it+1;
   h=h/2;
   y_ant_moulton=y;
   s(it)=y;
end
```

· Ejemplos

Se considerará como ecuación de ejemplo la propuesta en el ejercicio 6.1., es decir, que utilizaremos como ecuación diferencial a:

$$\begin{cases} y' = -y + t^2 + 1 \\ y(0) = 1 \\ 0 \le t \le 1 \end{cases}$$

```
» [s,it]=euler(0,1,0.1,0.6,10^-6,12)
      solucion no encontada para 12 iteraciones. (aprox=1.06237077560911)
» [s,it]=gragg(0,1,0.1,0.6,10^-6,12)
      solucion = 1.06237645983873,        it = 7
» [s,it]=kutta(0,1,0.1,0.6,10^-6,12)
      solucion = 1.06237675228038,        it = 2
» [s,it]=bash(0,1,0.1,0.6,10^-6,12
      solucion = 1.06237672245613,        it = 4
» [s,it]=moulton(0,1,0.1,0.6,10^-6,12)
      solucion = 1.06237673477933,        it = 3
```

Y ahora las extensiones mediante los métodos de extrapolación:

```
» [s,it]=xeuler(0,1,0.6,10^-6,0.1)
      solucion = 1.06237674667453,        it = 5
» [s,it]=xgragg(0,1,0.6,10^-6,0.1)
      solucion = 1.06237672781144,        it = 4
» [s,it]=xkutta(0,1,0.6,10^-6,0.1)
      solucion = 1.06237672783205,        it = 2
» [s,it]=xbash(0,1,0.6,10^-6,0.1)
      solucion = 1.06237672904745,        it = 3
```