MÉTODOS NUMÉRICOS

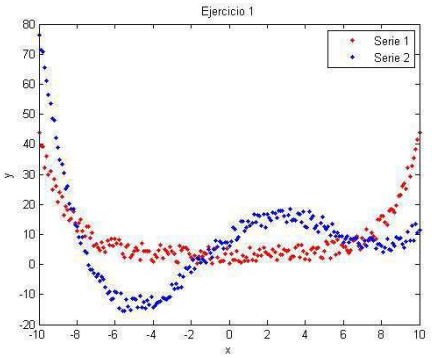
Trabajo integrador #2

Aproximación funcional y resolución no lineal de ecuaciones

Dominguez, Melina
Varela, Sofia
Año 2013

Parte #1

Consigna:
En el archivo adjunto, *datos.txt*, se encuentran dos series de puntos. La primer serie está contenida en la primer y segunda columna del archivo, mientras que la segunda serie está en la tercera y cuarta columna. Al realizar un gráfico de ambas series en conjunto, se observa la siguiente figura:



Se pide:
a) Ajustar la Serie#1 a la función, $f(x) = A\cosh(Bx)$, utilizando el comando MatLab *lsqcurvefit*.
b) Ajustar la Serie#2 con un *polinomio* $w(x)$, de grado 4.
c) Graficar ambas *series* de puntos y sus correspondientes *ajustes funcionales*.
d) Encontrar los puntos en los que $w(x) = f(x)$ con los métodos de resolución no lineal de funciones: *Bolzano*, *Falsa Posición*, *Secante* y *Newton–Rapson*, empleando tolerancias de error 10-6, 10-9 y 10-12.
e) Con el fin de comprar los métodos en cuanto a su performance (en cantidad de iteraciones efectuadas hasta alcanzar el error querido), completar la siguiente tabla con las coordenadas de alguna de las 3 raíces encontradas y cantidad de iteraciones efectuadas. Tenga en cuenta que para que la comparación tenga sentido, los algoritmos no deben cortar por número máximo de iteraciones, sino por la tolerancia al error pedido.
f) Mostrar en pantalla la raíz obtenida en los diferentes casos (solo para uno de los puntos de cruce). Tener en cuenta que para permitir una correcta comparación de los diferentes métodos, la cantidad de dígitos visibles debe corresponderse con la tolerancia seleccionada. Para ello, investigue el uso del comando MatLab *sprintf*.
g) ¿Qué conclusión se desprende de los datos y de la figura?

h) ¿Cuál método es el más rápido de los 4?

Resolución:

Para recolectar los valores guardados en el archivo de texto, el cual contenía 4 columnas de datos, se utilizó el comando *load*:

```
datos=load('datos1.txt');

serie1_x=datos(:,1);
serie1_y=datos(:,2);
serie2_x=datos(:,3);
serie2_y=datos(:,4);
```

La serie 1 se ajustó a la función $f(x) = A\cosh Bx$. Los parámetros A y B fueron conseguidos mediante el comando *lsqcurvefit.* Para ello se necesitó realizar una función externa que transforma, utilizando un coseno hiperbólico, un valor de x y un vector de coeficientes en un valor de y:

```
function y=Funcion_Ajuste_Serie1(coef,x)
%coef=[A B]
y=coef(1)*cosh(coef(2)*x);
```

Como el comando lsqcurvefit solo permite ingresar vectores *xdata* e *ydata* cuyos elementos sean números enteros, primero se realizaron dos series de datos auxiliares redondeando los valores de x e y de la serie (comando *round*).
A la función lsqcurvefit se le pasaron como parámetros: la función externa generada anteriormente, un valor inicial de coeficientes A y B (1 y 0) y los vectores *xdata* e *ydata* con valores enteros. La misma devolvió un vector con los valores de los coeficientes $E = [A\ B]$, que permitió generar el vector Ajuste_serie1:

```
%Ajuste
E=lsqcurvefit('Funcion_Ajuste_Serie1',[1 0],serie1_xbis,serie1_ybis);
Ajuste_serie1=E(1)*cosh(E(2)*serie1_x);
```

La función ajusta a la serie de datos como muestra el siguiente gráfico (Figura 1):

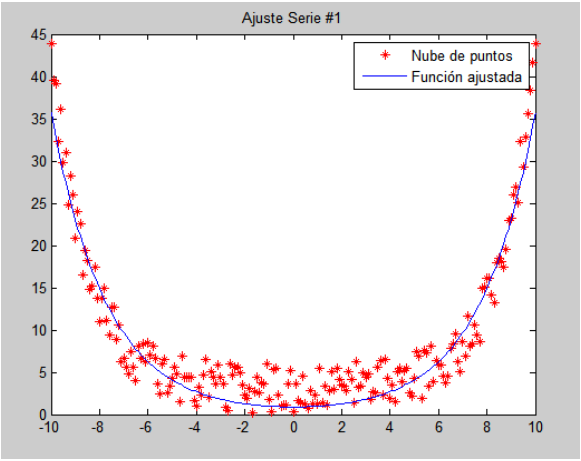


Figura 1: Ajuste con Acosh(Bx)

La segunda serie se ajustó con un polinomio de grado 4. Se utilizó la función externa Ajuste_Polinomico, que recibe como parámetros un vector de valores de abscisas, un vector de valores de ordenadas y el orden del polinomio, y devuelve un vector de coeficientes ordenados decrecientemente. La función utiliza otras dos funciones externas: triangulación gausseana y sustitución hacia atrás:

```
function C=Ajuste_Polinomico(x,y,M)
%M: orden del polinomio a aproximar

N=length(x');

C=zeros(M+1,1); %Vector de coeficientes (en orden creciente)

D=zeros(N,M+1);

for j=1:M+1
    D(:,j)=(x').^(j-1);
end

A=D'*D;
B=D'*y;

[A_n,B_n]=triang_gauss(A,B);
C=sust_atras(A_n,B_n);

C=flipud(C);    %Se entregan en orden decreciente
```

Con el valor de los 5 coeficientes del polinomio, se generó un vector de ordenadas:

```
C=Ajuste_Polinomico(serie2_x,serie2_y,4);
Ajuste_serie2=C(1)*serie2_x.^4+C(2)*serie2_x.^3+C(3)*serie2_x.^2+C(4)*serie2_x+C(5);
```

La correlación de la nube de datos y la función de ajuste se muestra en el gráfico de la figura 2.

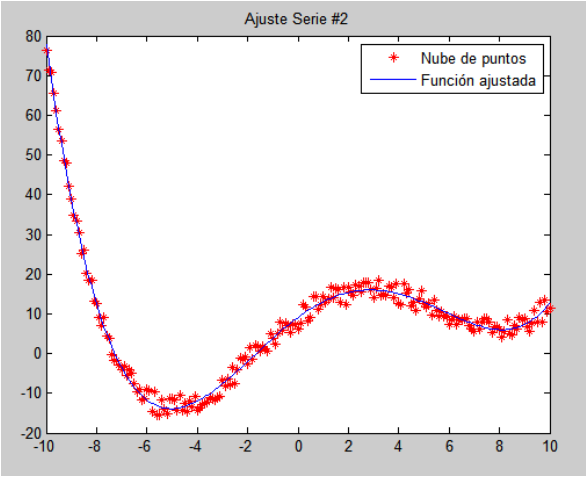


Figura 2: Ajuste con polinomio de grado 4

Para encontrar los puntos de intersección entre ambas funciones de ajustes se utilizó como técnica restar ambas funciones y encontrar las raíces de la función resultante. Cómo los métodos

de resolución de ecuaciones no lineales utilizan la función *feval*, se realizó una función externa auxiliar que, al recibir un valor de *x* y los vectores de coeficientes de ambas funciones de ajuste, resta las funciones y devuelve el valor de *y*:

```
function y=Funcion_Auxiliar(x,E,C)

 y=E(1)*cosh(E(2)*x)-(C(1)*x^4+C(2)*x^3+C(3)*x^2+C(4)*x+C(5));


end
```

Los métodos de Bolzano, Falsa Posición, Secante y Newton-Rapson fueron modificados para que reciban, además de los demás parámetros necesarios, los vectores de coeficientes de las funciones y para que los mismos sean enviados a la función *feval* cada vez que la misma sea solicitada.

La técnica mencionada permitió encontrar los puntos en el eje de abscisas en donde se intersecan ambas funciones.
Para encontrar el valor de los puntos en el eje de ordenadas, se buscó las ubicaciones de dichos valores en el rango x de alguna de las dos series (con una tolerancia de error de 0,1) y se reemplazó el índice correspondiente encontrado en la serie de datos en y:

```
%Vector r1x: Valores de x de primera intersección (Para distntos métodos y
%tolerancias)
%Vector r2x: Valores de y de segunda intersección (Para distntos métodos y
%tolerancias)
%Vector r3x: Valores de y de tercera intersección (Para distntos métodos y
%tolerancias)

%Calculo de punto en eje Y
for k=1:12
   for i=1:n
       if(abs(serie1_x(i)-r1x(k))<0.1)
           r1y(k)=serie1_y(i);
       end

       if(abs(serie1_x(i)-r2x(k))<0.1)
           r2y(k)=serie1_y(i);
       end

       if(abs(serie1_x(i)-r3x(k))<0.1)
           r3y(k)=serie1_y(i);
       end
   end
end
```

Todos los métodos aplicados con las distintas tolerancias de error ($10^{-6}, 10^{-9}, 10^{-12}$) coincidieron en que los puntos de intersección se encontraban en las coordenadas:

#1 Punto (-8.1538 ; 13.8142)
#2 Punto (-1.4854 ; 2.61)
#3 Punto (6.6328 ; 9.5023)

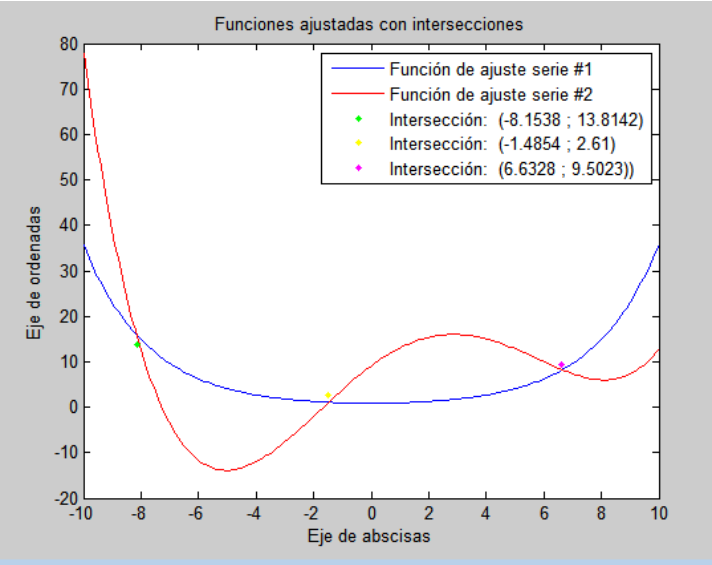


Figura 3: Funciones de ajuste con puntos de intersección

La diferencia entre los métodos utilizados no radicó en la exactitud de los valores obtenidos, sino en el tiempo de ejecución.

Para comparar los métodos se utilizó como referencia la búsqueda de la posición de la segunda intersección. Se modificaron los archivos externos de las funciones de los métodos para que todos finalicen la ejecución al llegar al valor de error deseado, y para que muestren la cantidad de iteraciones que realizaron para encontrar el valor de la raíz.

En la siguiente tabla se muestra la comparación:

	Tolerancia								
Método	10^{-6}			10^{-9}			10^{-12}		
	x	y	#N	x	y	#N	X	y	#N
Bolzano	-1.485420	2.610027	24	-1.485420412	2.610026600	32	-1.485420412276	2.610026600000	44
Falsa Posición	-1.485420	2.610027	4	-1.485420412	2.610026600	5	-1.485420412276	2.610026600000	5
Secante	-1.485420	2.610027	3	-1.485420412	2.610026600	3	-1.485420412276	2.610026600000	4
Newton-Rapson	-1.485420	2.610027	2	-1.485420412	2.610026600	2	-1.485420412276	2.610026600000	3

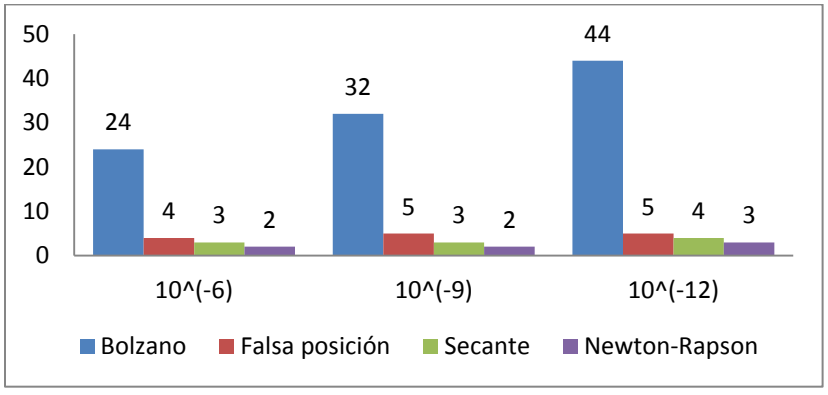


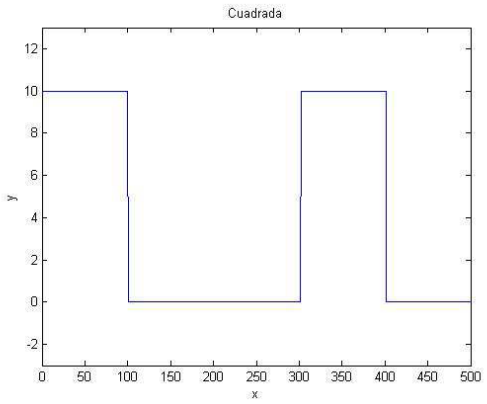
Figura 4: Tolerancia al error (abscisas) vs. cantidad de iteraciones (ordenadas)

Se concluye que la cantidad de iteraciones que realizan los métodos depende de la tolerancia al error que se elija.
El método de resolución más rápido es el de Newton-Rapson.

Parte #2

Consigna:

Realizar un algoritmo que utilice la *Serie Trigonométrica de Fourier* para realizar un video en el que se muestre como dicha serie va convergiendo a la señal periódica, a medida que se incrementa el orden de la serie.
Puntos a ajustar: Puntos de una señal cuadrada



Resolución:

Para realizar el ajuste se generó un vector X, con el valor del eje de abscisas para dos ciclos, y un vector Y correspondiente al vector X. El valor de Y(i) era de 10 cuando el valor de X(i) se encontraba entre 0 y 100 o entre 300 y 400, y de 0 para el resto de los casos.

La función de ajuste Serie_Trig_Fourier recibe como parámetros el vector de valores de abscisas de la función a ajustar, el vector de valores de ordenadas (y) y el orden (M), y devuelve el vector de abscisas y una nueva función ajustada (Y):

$$Y(k) = \frac{a_0}{2} + \sum_{i=1}^{M} a_i \cos\frac{2\pi ki}{N} + b_i \sin\frac{2\pi ki}{N}$$

$$a_i = \frac{2}{N}\sum_{m=1}^{N-1} y(m)\cos\frac{2\pi mi}{N}$$

$$b_i = \frac{2}{N}\sum_{m=1}^{N-1} y(m)\sin\frac{2\pi mi}{N}$$

(1)

Donde N es el largo de los vectores de datos (*length*).

Para visualizar como la serie ajustada con Fourier converge a la función cuadrada original a medida que se aumenta el orden, se generó un "gráfico variable" ingresado dentro de un ciclo *for* variando el valor de un índice k. Dentro del loop se ajusta la función utilizando el orden indicado por el índice k, se dibuja y se espera un tiempo determinado utilizando el comando *pause()*. Luego se incrementa el valor del índice y se vuelve a dibujar en la misma figura, sobrescribiendo el gráfico del índice anterior.

Si se incrementa el índice desde 1 hasta $\frac{N}{2-1}$ se observa, a modo de video, como la señal converge a la señal cuadrada inicial.

La variable de incremento del *for* también se utilizó para variar lo indicado por el comando *legend*: A medida que el orden del ajuste aumenta, el recuadro de leyenda indica en qué orden se está dibujando la función.

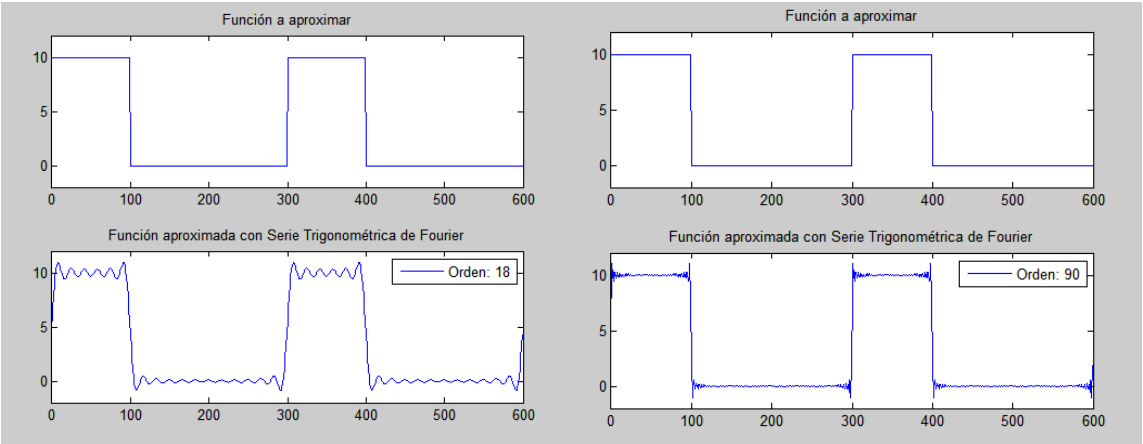


Figura 5: Ejemplos de visualización (orden 18 y orden 90)

Se concluye que el método numérico realizado a partir de la serie de Fourier discretizada (ecuación 1) es correcto, ya que la función aplicada en este ejercicio converge cómo lo indica la resolución teórica.