La serie de Taylor

Si la función *f* y sus primeras *n+1* derivadas son continuas, en un intervalo que contiene *a* y *x*, entonces el valor de la función esta dado por:

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

Con frecuencia es conveniente simplificar la serie de Taylor definiendo un paso $h = x_{i+1} - x_i$ expresando la serie de Taylor como:

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \ldots + \frac{f^n(x_i)}{n!}h^n$$

Uso de la expansión en serie de Taylor para aproximar una función con un número infinito de derivadas.

Utilizar los términos de la serie de Taylor con n= 0 hasta 6 para aproximar la función f(x) = cos(x) en $x_{i+1} = \pi/3$ y sus derivadas en $x_i = \pi/4$. Esto significa que $h = \pi/3 - \pi/4 = \pi/12$, los valores de las derivadas y el error de aproximación se presenta en la siguiente tabla.

Orden n	$f^n(x)$	$f^n(\pi/4)$	error (%)
0	cos(x)	0.707106781	-41.4
1	-sen(x)	0.521986659	-4.4
2	-cos(x)	0.497754491	0.449
3	sen(x)	0.499869147	2.62×10^{-2}
4	cos(x)	0.500007551	-1.51×10^{-3}
5	-sen(x)	0.500000304	-6.08×10^{-5}
6	-cos(x)	0.499999988	2.40×10^{-6}

Note, que a medida que se introducen más términos, la aproximación se vuelve más exacta y el porcentaje de error disminuye. En general podemos tener una aproximación polinomial de la función coseno, con sus derivadas en cero dada por

Orden n	$f^n(x)$	$f^n(0)$
0	cos(x)	1
1	-sen(x)	0
2	-cos(x)	-1
3	sen(x)	0
4	cos(x)	1
5	-sen(x)	0
6	-cos(x)	-1
7	sen(x)	0

8	cos(x)	1
9	-sen(x)	0
10	-cos(x)	-1

La aproximación polinomial final queda:

$$f(x) = 1 - \frac{1}{2}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \frac{1}{8!}x^8 - \frac{1}{10!}x^{10} + \ldots$$

La implementación en Java es:

```
class funciones
{

  public static double coseno(double x)
  {
    int i;
    double s = 0;
    int signo = 1;

    for(i=0; i<10; i+=2)
    {
      s += signo*pow(x, i) / factorial(i);
      signo *= -1;
    }

    return s;
  }

  public static double seno(double x)
  {
    int i;
    double s = 0;
    int signo = 1;

    for(i=1; i<10; i+=2)
    {
      s += signo*pow(x, i) / factorial(i);
      signo *= -1;
    }

    return s;
  }

  public static double factorial(int x)
  {
    int i;
    double fact = 1;

    for(i=1; i<=x; i++)
      fact *= i;

    return fact;
  }
```

```
  public static double pow(double x, int n)
  {

    int i;
    double pow =1;

    if(x==0) return 0;

    for(i=1; i<=n; i++)
      pow = pow*x;

    return pow;
  }
}
```

El programa principal para realizar el llamado es:

```
public class ej053 {
  public static void main(String[] args) {

    int i,N;

    double inicio = -3.1416, fin = 3.1416, incremento = 0.1;
    N = (int)((fin - inicio)/incremento) + 1;

    double x[] = new double [N];
    double y[] = new double [N];
    double z[] = new double [N];

    for(i=0;  i<N; i++)
    {
      x[i] = inicio + incremento *i;
      y[i] = funciones.seno(x[i]);
      z[i] = funciones.coseno(x[i]);
    }

    grafica g = new grafica("Funcion seno");
    g.Datos(x,y);
    g.Datos(x,z);
    g.show();
  }
}
```

Note en este último código, el uso de la función gráfica que permite desplegar una función dada como dos arreglos de datos.

Uso de la serie de Taylor para estimar errores de Truncamiento.

La serie de Taylor es muy útil para hacer la estimación de errores de truncamiento. Esta estimación ya la realizamos en los ejemplos anteriores. Recordemos que la serie de Taylor la podemos representar como:

Ahora, truncando la serie después del término con la primera derivada, se obtiene:

Despejando el valor de *v'*, tenemos:

El primer término de la ecuación represente la aproximación de la derivada y el segundo el error de truncamiento. Note que el error de truncamiento se hace más pequeño a medida que $t_{i+1} - t_i$ (incremento) se hace pequeño. Así que podemos hacer una buena aproximación de derivadas utilizando el primer término, siempre y cuando se utilicen incrementos pequeños.

Regresar.

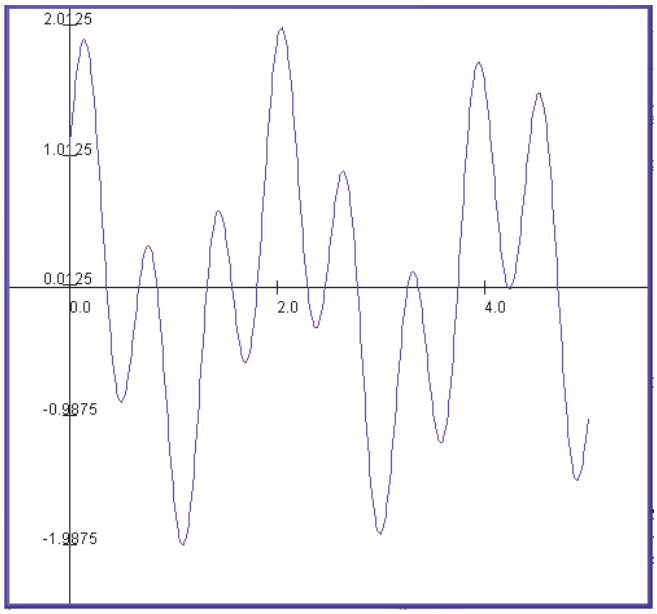
Raíces de Ecuaciones

Métodos Gráficos

Un método simple para obtener, visualmente la raíz de la ecuación *f(x)=0*, consiste en graficar la función y observar en donde cruza el eje x. Este punto que representa el valor de *x* para el cual *f(x)=0*, proporciona una aproximación inicial de la raíz.

Ejemplo.

Utilice el método gráfico para observar algunas de las raíces de la función *f(x) = sen 10x + cos 3x*, en el intervalo [0,5]



Note que la función cruza el eje x en varios puntos, los cuales son solución a *f(x)=0.* La implementación en Java para esta es:

```
public class ej056 {
  public static void main(String[] args) {
    int i, N;

    double inicio = 0, fin = 5, incremento = 0.01;
    N = (int) ( (fin - inicio) / incremento) + 1;

    double x[] = new double[N];
    double y[] = new double[N];

    for (i = 0; i < N; i++) {
      x[i] = inicio + incremento * i;
      y[i] = Math.sin(10.0*x[i]) + Math.cos(3.0*x[i]);
    }

    grafica g = new grafica("Funcion seno");
    g.Datos(x, y);
    g.show();
  }
}
```

Regresar.

El método de Bisección de Bolzano

Para este método debemos considerar una función continua dentro de un intervalo *[a,b]* tal que *f(a)* tenga diferente signo *f(a)*f(b) <0.*

El proceso de decisión para encontrar la raíz consiste en dividir el intervalo *[a, b]* a la mitad *c = (a+b)/2* y luego analizar las tres posibilidades que se pueden dar.

1.- Si *f(a)* y *f(c)* tienen signos opuestos, entonces hay un cero entre *[a,c]*.
2.- Si *f(c)* y *f(b)* tienen signos opuestos, entonces, hay un cero en *[a,b]*.
3.- Si *f(c)* es igual a cero, entonces c es un cero.

La implementación en Java es:

```
double biseccion()
{
  int i;
  double inicio =0, fin = 1, mitad;

  mitad = (fin+inicio)/2.0;
  while(fin - inicio > 0.0001)
  {
    System.out.println(mitad);
    pausa(300);
    mitad = (fin+inicio)/2.0;
    if(f(mitad) == 0) return mitad;
    if(f(inicio)*f(mitad) < 0 )
      fin = mitad;
    else
      inicio = mitad;
  }
  return mitad;
}
```

Ejemplo

Calcular los ceros de la función x-cos(x) utilizando el algoritmo de bisección en el intervalo [0,1].

iter	a	c	b	f(a)	f(c)	f(b)
0	0.0000	0.5000	1.0000	-1.0000	-0.3776	-1.0000
1	0.5000	0.7500	1.0000	-0.3776	0.0183	-0.3776
2	0.5000	0.6250	0.7500	-0.3776	-0.1860	-0.3776
3	0.6250	0.6875	0.7500	-0.1860	-0.0853	-0.1860
4	0.6875	0.7188	0.7500	-0.0853	-0.0339	-0.0853
5	0.7188	0.7344	0.7500	-0.0339	-0.0079	-0.0339
6	0.7344	0.7422	0.7500	-0.0079	0.0052	-0.0079
7	0.7344	0.7383	0.7422	-0.0079	-0.0013	-0.0079
8	0.7383	0.7402	0.7422	-0.0013	0.0019	-0.0013
9	0.7383	0.7393	0.7402	-0.0013	0.0003	-0.0013
10	0.7383	0.7388	0.7393	-0.0013	-0.0005	-0.0013
11	0.7388	0.7390	0.7393	-0.0005	-0.0001	-0.0005
12	0.7390	0.7391	0.7393	-0.0001	0.0001	-0.0001
13	0.7390	0.7391	0.7391	-0.0001	0.0000	-0.0001

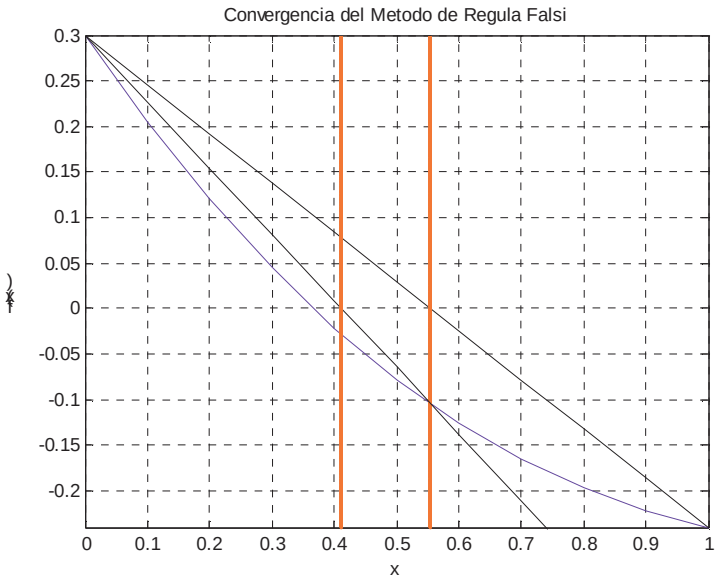
Su ejecución puede verse [aquí].

Regresar.

Método de Régula Falsi.

Una de las razones de la introducción de este método es que la velocidad de convergencia del método de biseccciones es bastante baja. En el método de bisección se usa el punto medo del intervalo *[a,b]* para llevar a cabo el siguiente paso. Suele conseguirse una mejor aproximación usando el punto *(c,0)* en el que la recta secante *L*, que pasa por los puntos *[a, f(a)]* y *[b, f(b)]*.

En la figura se puede ver como funciona el método. En esta figura en azul esta la función de la cual queremos calcular el cruce por cero y en negro dos líneas rectas que aproximan la solución.



Para calcula la ecuación de la línea secante hacemos

$$p_1 = [a, f(a)]$$
$$p_2 = [b,f(b)]$$

y sustituimos en la ecuación de la línea recta.

$$y - f(a) = (f(b) - f(a)) * (x - a)/(b - a)$$

El cruce por cero de esta ecuación esta en

$$c = a - f(a) * (b-a)/(f(b) - f(a))$$

Entonces el método de biseccciones puede ser modificado, en lugar de calcular c = (a+b)/2 hacemos $c = a - f(a) * (b-a)/(f(b) - f(a))$ y aplicamos las mismas tres reglas de la bisección.

Ejemplo

Calcular los ceros de la función x-cos(x) utilizando el algoritmo de regula falsi en el intervalo [0,1].

iter.	a	c	b	f(a)	f(c)	f(b)
0	0.0000	0.6851	1.0000	-1.0000	-0.0893	-1.0000
1	0.6851	0.7363	1.0000	-0.0893	-0.0047	-0.0893
2	0.7363	0.7389	1.0000	-0.0047	-0.0002	-0.0047
3	0.7389	0.7391	1.0000	-0.0002	0.0000	-0.0002
4	0.7391	0.7391	1.0000	0.0000	0.0000	0.0000
5	0.7391	0.7391	1.0000	0.0000	0.0000	0.0000
6	0.7391	0.7391	1.0000	0.0000	0.0000	0.0000

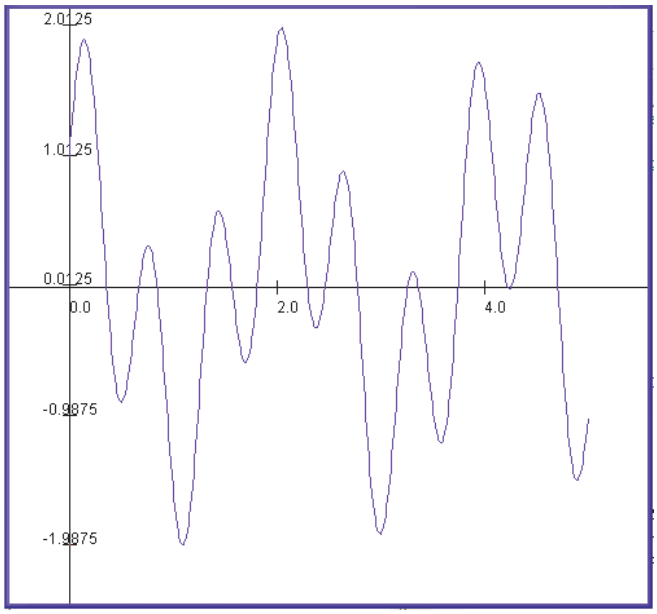
Ver ejemplo de convergencia.

Regresar.

Búsqueda con incrementos.

Como se ha mencionado, el graficar a función de la cual se desea calcular su raíz, es de mucha utilidad. Si queremos que esta tarea se realice de manera automática, una forma de hacerlo es por medio de incrementos.

Seleccionar un incremento no es tarea fácil, así por ejemplo, en la siguiente figura podemos ver, la gráfica de la función $f(x)=sen10x + cos\ 3x$



Así por ejemplo si tomamos dos puntos $x_1=0$ y $x_2 = 2$, las funciones valuadas en estos valores son $f(0) =0.841470985$ y $f(2)= 0.85780222$, podríamos suponer que la función

crece en la dirección positiva de x, cuando en realidad en este intervalo existen 6 raíces. Ahora si tomamos el intervalo $x_1=0$ y $x_2 = 1$ las funciones valuadas en estos valores son $f(0) =0.841470985$ y $f(2)= 0.402979862$, podríamos pensar que más adelante existe una raíz, cuando en realidad en este intervalo existen más de una.

Una manera de solucionar este problema es poner un punto inicial x_0 y un incremento h lo suficientemente pequeño para evitar las situaciones antes descritas. El algoritmo consiste en:

1.- Dado x_0 y h, hacemos $k = 1$
2.- Calculamos $x_k = x_{k-1} + h$
3.- Duplicamos el incremento $h = 2*h$
4.- Si $f(x_0) * f(x_k) < 0$ paramos, sino regresamos a 2.

Para nuestra función tenemos:

h	x_k	$f(x_k)$
	0.000	0.841
0.001	0.001	0.847
0.002	0.003	0.857
0.004	0.007	0.877
0.008	0.015	0.912
0.016	0.031	0.965
0.032	0.063	0.999
0.064	0.127	0.809
0.128	0.255	-0.129
0.256	0.511	-0.907
0.512	1.023	0.191
1.024	2.047	0.507

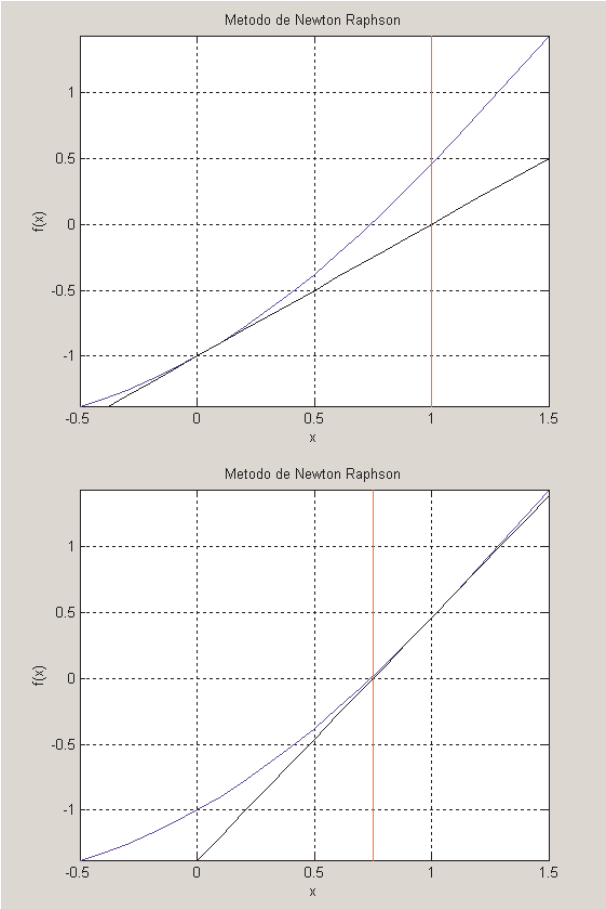
Podemos ver que con un incremento de 0.001, hemos localizado una raíz en el intervalo $x = [0.000, 0.255]$. Esta es la información que utilizamos para inicializar, ya sea, el método de Regula Falsi o Bisecciones.

Regresar.

El Método de Newton-Raphson.

Si tenemos una función $f(x)$ continua y cerca de una raíz p. Si la derivada $f'(x)$ existe, entonces puede utilizarse para desarrollar algoritmos que produzcan sucesiones $\{pk\}$ que converjan a p más rápidamente que los algoritmos de bisección.

Consideremos el caso de una función como la que se muestra en la figura.



Dos iteraciones del método de Newton.

En estas figuras se muestra dos iteraciones del método. En la figura de la izquierda mostramos la línea recta que es tangente a la función *f(x)* (en negro), note que la línea recta cruza el eje x en $x = 1$. A la derecha tenemos la segunda iteración tomando como valor inicial $x=1$. Note como poco a poco se acerca a la solución.

La sucesión que nos lleva a la solución esta dada por los puntos $\{p_0, p_1, p_2, \ldots, p_k\}$. La pendiente de la línea recta es

$$m = (0 - f(p_0))/(p_1 - p_0)$$

Por otro lado sabemos, del cálculo diferencial, que la pendiente de la línea tangente a una función es la primer derivada valuada en ese punto. Así:

$$m = f'(p_0)$$

Uniendo las ecuaciones tenemos

$$f'(p_0) = (0 - f(p_0))/(p_1 - p_0)$$

$$p_1 = p_0 - f(p_0)/f'(p_0)$$

De manera iterativa podemos hacer

$$p_2 = p_1 - f(p_1)/f'(p_1)$$
$$p_3 = p_2 - f(p_2)/f'(p_2)$$

$$p_{k+1} = p_k - f(p_k)/f'(p_k)$$

Ejemplo.

Hacer un algoritmo iterativo que permita hacer el cálculo de la raíz cuadrada de A.

Para este caso nuestra función a resolver es $f(x) = x^2 - A$. La solución cuando $f(x)=0$ es $x = A^{0.5}$.

$f(x) = x^2 - A$
$f'(x) = 2x$

$$p_{k+1} = p_k - f(p_k)/f'(p_k)$$
$$p_{k+1} = p_k - (p_k^2 - A))/(2\,p_k)$$
$$p_{k+1} = (p_k + A/p_k)/2$$

Los cálculos numéricos suponiendo A=5 son:

k	pk
0	2.0000
1	2.2500
2	2.2361
3	2.2361
4	2.2361
5	2.2361
6	2.2361
7	2.2361
8	2.2361
9	2.2361
10	2.2361

Ejemplo

Calcular los ceros de la función x-cos(x) utilizando el algoritmo de regula falsi en el intervalo [0,1].

k	pk
0	0.0000
1	1.0000

2	0.7504
3	0.7391
4	0.7391
5	0.7391
6	0.7391
7	0.7391

Método de Newton Raphson para sistemas no lineales

Consideremos el sistema

Utilizando la serie de Taylor podemos hacer una aproximación lineal

Si escribimos el sistema original como una función vectorial $V=F(X)$, entonces la matriz jacobiana $J(x,y)$ es el análogo bidimensional de la derivada. La aproximación lineal queda como:

donde

Entonces nuestra formulación bidimensional queda como:

y la actualización de la variable la hacemos:

Ejemplo.

Resolver el siguiente sistema de ecuaciones dado por

$$f_1(x,y) = x^2 - 2x - y + 0.5$$
$$f_2(x,y) = x^2 + 4y^2 - 4$$

El jacobiano es

Considerando como valores iniciales [0, 1] tenemos:

Primer iteración

cuya solución es $\Delta x = 0.25$ y $\Delta y = 0$

Segunda iteración

cuya solución es $\Delta x = -0.0274$ y $\Delta y = 0.0061$

Las demás iteraciones la resumimos es:

k	x	y
0	0.0000	1.0000
1	-0.2500	1.0000
2	-0.2260	0.9939
3	-0.2222	0.9938
4	-0.2222	0.9938
5	-0.2222	0.9938
6	-0.2222	0.9938
7	-0.2222	0.9938

La implementación en Java de este método es:

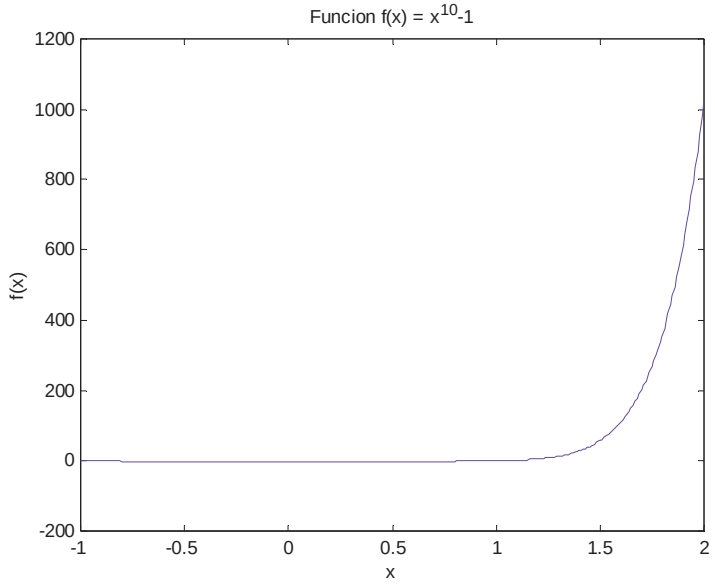
```
double Newton()
{
  int i;
  double m, b, x;

  x = 0;

  while(Math.abs(f(x)) > 0.0001)
  {
    m = df(x);
    b = - m*x + f(x);

    pausa(300);
    x = x - f(x)/m;
  }
  return x;
}

double f(double x)
{
  return (x-Math.cos(x));
}

double df(double x)
{
  return(1+Math.sin(x));
}
```

Desventajas del Método de Newton.

Aunque el método de Newton-Raphson en general es muy eficiente, hay situaciones en que se comporta en forma deficiente. Un caso especial, raíces múltiples.

Ejemplo.

Determinar la raíz de la función $f(x) = x^{10} - 1$.



La solución utilizando el método de Newton queda:

Y la solución numérica es:

x	f(x)	df(x)
0.5000	-0.9990	0.0195
51.6500	135114904483914000.0000	26159710451871000.0000
46.4850	47111654129711500.0000	10134807815362300.0000
41.8365	16426818072478500.0000	3926432199748670.0000
37.6529	5727677301318310.0000	1521180282851980.0000
33.8876	1997117586819850.0000	589336409039672.0000
30.4988	696351844868619.0000	228320999775654.0000
27.4489	242802875029547.0000	88456233382052.8000
24.7040	84660127717097.5000	34269757191973.2000
22.2336	29519161271064.1000	13276806089225.7000
20.0103	10292695105054.7000	5143706707446.1600
18.0092	3588840873655.1100	1992777367871.5700
16.2083	1251351437592.9200	772042782329.1500
14.5875	436319267276.5290	299105192259.1190
13.1287	152135121499.2910	115879479847.7330
11.8159	53046236848.5329	44894084747.9692
10.6343	18496079117.2577	17392888266.5936
9.5708	6449184014.3077	6738361277.7304
8.6138	2248691421.7628	2610579221.6818
7.7524	784070216.9426	1011391879.0870

Su ejecución puede verse [aquí].

Regresar.

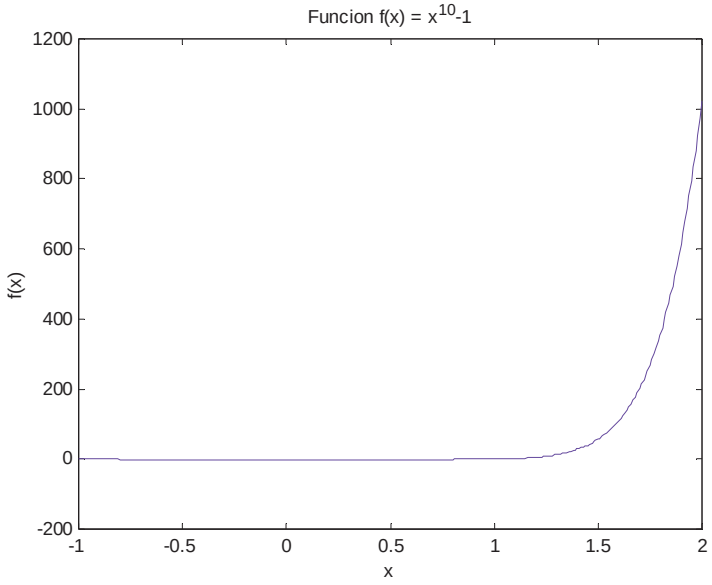
Método de la Secante.

Un problema en la implementación del método de Newton-Raphson es el de la evaluación de la derivada. Aunque esto no es un inconveniente para los polinomios y para muchas otras funciones, existen algunas funciones cuyas derivadas pueden ser en extremo difíciles de evaluar. En estos casos, la derivada se puede aproximar mediante una diferencia finita.

Sustituyendo esta aproximación en la formula de Newton, tenemos la fórmula del método de la secante.

Ejemplo.

Encontrar la solución utilizando el método de la secante para la función $f(x) = x^{10} - 1$.



x0	x1	f(x0)	f(x1)

0.5000	0.9000	-0.9990	-0.6513
0.9000	1.6493	-0.6513	147.9235
1.6493	0.9033	147.9235	-0.6384
0.9033	0.9065	-0.6384	-0.6253
0.9065	1.0602	-0.6253	0.7945
1.0602	0.9742	0.7945	-0.2301
0.9742	0.9935	-0.2301	-0.0630
0.9935	1.0008	-0.0630	0.0080
1.0008	1.0000	0.0080	-0.0002

Note, que para este problema, el método de la secante si lleva a una solución. Este hecho se debe a que calcula su aproximación apoyada de dos puntos, no de uno solo.

La implementación de este método en Java es :

```
double Secante()
{
  int i;

  double m, x, x0 = 0, h = 0.5;

  x      = inicio + h;

  while(Math.abs(f(x)) > 0.0001)
  {
    m = (f(x) - f(x0))/(x - x0);
    x0 = x;

    x = x - f(x)/m;
  }
  return x;
}
```

Su ejecución puede verse [aquí].

Regresar.

Raíces Múltiples.

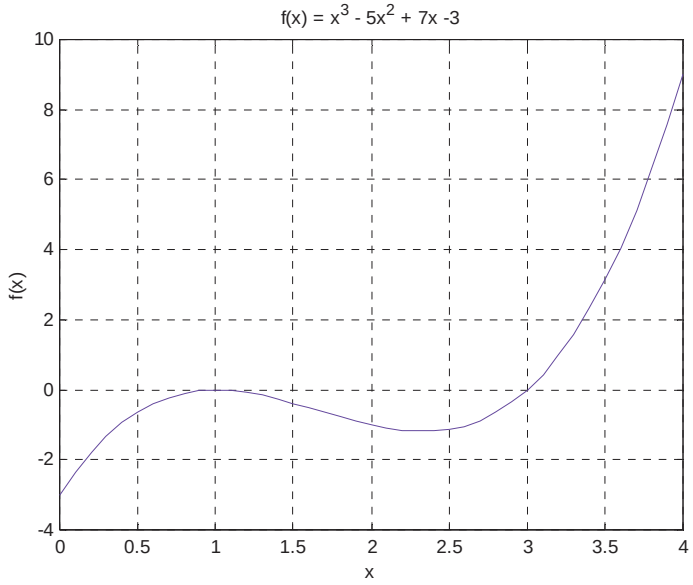
Una raíz múltiple corresponde a un punto donde una función es tangente al eje x. Por ejemplo, una raíz doble resulta de

$$f(x) = (x-3)(x-1)(x-1)$$

multiplicando términos este polinomio luce como

$$f(x) = x^3 - 5x^2 + 7x - 3$$

En la siguiente figura podemos ver como la función toca tangencialmente el eje de la x, en el punto donde existe la raíz doble.



De la figura podemos ver algunos de los problemas asociados con raíces múltiples, dichos problemas son:

Dado que la función no cambia de signo, utilizar métodos basados en intervalos, como son el método de bisecciones, regula Falsi, etc. Otro problema es que cerca de la solución, la derivada tiende a cero, lo cual provoca que en el algoritmo de Newton-Raphson tenga problemas de convergencia al tener una división por cero.

Ralston y Rabinowitz (1978) proponen que se haga un pequeño cambio en la formulación para que retorne la convergencia, así la formulación para el método de Newton-Raphson es

en donde *m*, es la multiplicidad de la raíz. Para este caso será necesario conocer a priori el número de raíces multiples.

Otra alternativa propuesta por Ralston y Rabinowitz (1978) es la de definir una nueva función u(x), que es el cociente de la función y su derivada

Se puede mostrar esta función tiene las mismas raíces que *f(x)* y que la multiplicidad de raíces no afectará. La formulación del método de Newton-Raphson es:

La derivada de *u(x)* es:

Sustituyendo esta, tenemos la formulación final del Método de Newton-Raphson modificado.

Ejemplo.

Hacer una comparación entre el método de Newton-Raphson y el método de Newton-Raphson modificado, para encontrar las raíces del polinomio $f(x) = x^3 - 5x^2 + 7x - 3$.

Para nuestro calculo requerimos de:

$$f(x) = x^3 - 5x^2 + 7x - 3$$
$$f'(x) = 3x^2 - 10x + 7$$
$$f''(x) = 6x - 10$$

Primeramente resolvemos con $x_0 = 0$.

Newton-Raphson

k	xk	f(xk)
0	0	-3
1	0.42857143	-0.83965015
2	0.68571429	-0.22859475
3	0.8328654	-0.06053668
4	0.91332989	-0.01567446
5	0.95578329	-0.00399668
6	0.9776551	-0.00100975
7	0.98876617	-2.54E-04
8	0.99436744	-6.36E-05
9	0.99717977	-1.59E-05
10	0.99858889	-3.99E-06

Newton-Raphson modificado

k	xk	f(xk)
0	0	-3
1	1.10526316	-0.02099431
2	1.00308166	-1.8964E-05
3	1.00000238	-1.1343E-11

11	0.9992942	-9.97E-07
12	0.99964704	-2.49E-07
13	0.9998235	-6.23E-08
14	0.99991175	-1.56E-08
15	0.99995587	-3.89E-09
16	0.99997794	-9.74E-10
17	0.99998897	-2.43E-10
18	0.99999448	-6.09E-11

La solución con $x_0 = 4$

Newton-Raphson

k	xk	f(xk)
0	4	9
1	3.4	2.304
2	3.1	0.441
3	3.00869565	0.03508572
4	3.00007464	2.99E-04
5	3.00000001	2.23E-08
6	3	-1.07E-14

Newton-Raphson modificado

k	xk	f(xk)
0	4	9
1	2.63636364	-0.97370398
2	2.82022472	-0.5956347
3	2.96172821	-0.1472843
4	2.99847872	-6.08E-03
5	2.99999768	-9.27E-06
6	3	-2.15E-11

Note que cuando existe una raíz múltiple (en el caso de $x=1$), el algoritmo de Newton Raphson modificado, tiene mejor comportamiento, que cuando no es el caso de raíz múltiple ($x=3$).

Implementación en Java.

```
/**
 * <p>Title: Metodo de Newton Raphson Modificado</p>
 * <p>Description: Compara los métodos de Newton</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: UMSH</p>
 * @author Dr. Felix Calderon Solorio.
 * @version 1.0
 */

public class ej059 {

  public static void main(String[] args) {
   Newton_Raphson(0, 1e-10);
   Newton_Raphson_Modificado(0, 1e-10);
  }

  public static double Newton_Raphson(double x0, double T)
  {
    int i =0;
    double x = x0;


    while(Math.abs(f(x)) > T)
    {
```

```
      System.out.println("Iteracion " + i++ + " " + x + " " + f(x));
      x = x - f(x)/df(x);
    }
    System.out.println("Iteracion " + i++ + " " + x + " " + f(x));
    return x;
  }

  public static double Newton_Raphson_Modificado(double x0, double T)
  {
    int i =0;
    double x = x0;


    while(Math.abs(f(x)) > T)
    {
      System.out.println("Iteracion " + i++ + " " + x + " " + f(x));
      x = x - (f(x)*df(x))/(df(x)*df(x) - f(x)*ddf(x));
    }
    System.out.println("Iteracion " + i++ + " " + x + " " + f(x));
    return x;
  }

  public static double f(double x)
  {
    return (x*x*x - 5.0*x*x + 7.0*x - 3.0);
  }

  public static double df(double x)
  {
    return (3.0*x*x - 10.0*x + 7.0);
  }

  public static double ddf(double x)
  {
    return (6.0*x - 10.0);
  }

}
```

Regresar.

Raíz de polinomios

Es esta sección se analizará los métodos para encontrar las raíces de ecuaciones polinomiales de la forma general

$$f_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n.$$

Evaluación de polinomios

Consideremos como ejemplo el siguiente polinomio.

$$f_3(x) = a_0 + a_1x + a_2x^2 + a_3x^3.$$

En este caso los coeficientes del polinomio los podemos poner en un vector de datos dado como $a = [a_0, a_1, a_2, a_3]$, y realizar la evaluación polinomial utilizando ciclos. Adicionalmente necesitamos de una función que nos permita calcular la potencias de x.

Manipulando el polinomio, podemos dar una forma eficiente de realizar la evaluación. Así el mismo polinomio, lo podemos representar como:

$$f_3(x) = a_0 + x(a_1 + x(a_2 + xa_3))$$

Note que ahora ya no tenemos potencia de x y su implementación resulta mas eficiente.

Ejemplo.

Consideremos el siguiente polinomio $f_3(x) = 1+2x+3x^2$, el cual deseamos evaluar en $x = 20$. Para implementarlo hacemos

n	p = p*x + a[i];	
2	p = 0*20+3	p = 3
1	p = 3*20+2	p = 62
0	p = 62*20+1	p = 1241

Cálculo de derivadas.

Cuando se buscan los ceros de una función, como es el caso del método de Newton, es necesario no solo hacer la evaluación del polinomio sino calcular también su derivada. En este caso la derivada de cualquier polinomio la podemos calcular como:

$$f_n(x) = a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2x^2 + a_1x + a_0$$

$$f'_n(x) = na_nx^{n-1} + (n-1)a_{n-1}x^{n-2} + (n-2)a_{n-2}x^{n-3} + \ldots + 2a_2x + a_1$$

Note que el polinomio $f'_n(x)$ es un polinomio de menor grado. Así por ejemplo considerando el polinomio $f_3(x) = 1+2x+3x^2$, tenemos que su derivada es:

n	df = df*x + a[i+1]*(i+1)	df
1	df = 0*20+3*2	6
0	pf = 6*20+2*1	122

Deflación polinomial.

Suponga que conoce, una de las raíces de un polinomio, podemos realizar la división de este polinomio para obtener un polinomio de grado menor. Así por ejemplo si tenemos

$$f_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n.$$

Y una de sus raíces es *s*, entonces podemos escribir

$$f_n(x) = (x-s)*(a'_0 + a'_1x + a'_2x^2 + \ldots + a'_{n-1}x^{n-1}).$$

En este caso el residuo de la división es cero y podemos calcular un polinomio de grado *n-1*. Para un polinomio de orden 3 tenemos que

	a_3x^2	$(a_2-a_3s)\,x$	$a_1-(a_2-a_3s)s$	
$x+s$	a_3x^3	$+a_2x^2$	$+\,a_1x$	$+a_0$
	$-a_3x^2$	$-\,a_3\,s\,x^2$		
	0	$(a_2-a_3s)\,x^2$	$+\,a_1x$	$+a_0$
		$-(a_2-a_3s)\,x^2$	$-(a_2-a_3s)sx$	
		0	$(a_1-(a_2-a_3s)s)x$	$+a_0$
			$-(a_1-(a_2-a_3s)s)x$	$-(a_1-(a_2-a_3s)s)s$
				$a_0-(a_1-(a_2-a_3s)s)s$

La división sintética, es una manera de hacer lo mismo, pero de forma compacta. Así tenemos:

$x+s$	a_3	a_2	a_1	a_0
		$-\,a_3\,s$	$-(a_2-\,a_3\,s)s$	$-(a_1-(a_2-s\,a_3)s)s$
	a_3	$a_2-\,a_3\,s$	$a_1-(a_2-\,a_3\,s)s$	$a_0-(a_1-(a_2-\,a_3\,s)s)s$

Ejemplo.

Dado el polinomio $f_5(x) = x^5 - 7x^4 - 3x^3 + 79x^2 - 46x - 120$ encontrar la división sintética con el monomio $(x-4)$.

x-4	1	-7	-3	79	-46	-120
		4	-12	-60	76	120
	1	-3	-15	19	30	0

El polinomio resultante, en este caso, es $f_4(x) = x^4 - 3x^3 - 15x^2 + 19x + 30$. Note que el residuo es cero.

Implementación en Java.

```
/**
 * <p>Title: Evaluacion polinomial</p>
 *  <p>Description: Realiza la evaluacion de polinomios asi como el
calculo de
 * derivadas y division sintetica</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: UMSNH</p>
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
```

```
 */

public class ej057 {
  public static void main(String[] args) {

    double a[] = {-120, -46, 79, -3, -7, 1};
    double b[] = new double [a.length];

    System.out.println(EvaluaPolinomio(a, 1));
    System.out.println(EvaluaDerivada(a, 1));
    DivisionSintetica(a, b, -4);
  }

  public static double EvaluaPolinomio(double a[], double x)
  {
    int n = a.length, i;
    double p = 0;

    for(i=n-1; i>=0; i--)
      p = p*x + a[i];

    return p;
  }
  public static double EvaluaDerivada(double a[], double x)
  {
    int n = a.length, i;
    double df = 0;

    for(i=n-2; i>=0; i--)
      df = df*x + a[i+1]*(i+1);

    return df;
  }
  public static void DivisionSintetica(double a[], double b[], double s)
  {
    int n = a.length, i;

    b[n-1] = a[n-1];

    for(i=n-2; i>=0; i--)
      b[i] = a[i] - b[i+1] * s;

    for(i=n-1; i>0; i--)
      System.out.print(b[i] + " ");

    System.out.println( "residuo = " +  b[0]);

  }
}
```

Regresar.

Método de Müller.

El método de la secante obtiene raíces de una función estimando una proyección de una línea recta en el eje de las x, a través de los valores de la función. El método de Müller, trabaja de manera similar, pero en lugar de hacer la proyección de una recta utilizando dos puntos, requiere de tres puntos para calcular una parábola.

Para esto necesitaremos de tres puntos [x0, f(x0)], [x1, f(x1)] y [x2, f(x2)]. La aproximación la podemos escribir como:

$$f_2(x) = A(x - x_2)^2 + B(x - x_2) + C$$

Los coeficientes de la parábola los calculamos resolviendo el siguiente sistema de ecuaciones.

$$f_2(x_0) = A(x_0 - x_2)^2 + B(x_0 - x_2) + C$$
$$f_2(x_1) = A(x_1 - x_2)^2 + B(x_1 - x_2) + C$$
$$f_2(x_2) = A(x_2 - x_2)^2 + B(x_2 - x_2) + C$$

De la última ecuación podemos ver que el calor de $C = f_2(x_2)$. Sustituyendo los valores de C en las otras dos ecuaciones tenemos

$$f_2(x_0) - f_2(x_2) = A(x_0 - x_2)^2 + B(x_0 - x_2)$$
$$f_2(x_1) - f_2(x_2) = A(x_1 - x_2)^2 + B(x_1 - x_2)$$

Si definimos

$$h_0 = x_1 - x_0$$
$$h_1 = x_2 - x_1$$
$$d_0 = [f(x_1) - f(x_0)]/[x_1 - x_0]$$
$$d_1 = [f(x_2) - f(x_1)]/[x_2 - x_1]$$

Sustituyendo en las ecuaciones tenemos

$$-(d_0 * h_0 + d_1 * h_1) = A(h_1 + h_0)^2 - B(h_1 + h_0)$$
$$-d_1 * h_1 = A(h_1)^2 - Bh_1$$

La solución de este sistema de ecuaciones es:

$$A = (d_1 - d_0)/(h_1 + h_0)$$
$$B = Ah_1 + d_1$$
$$C = f(x_2)$$

Ahora para calcular la raíz del polinomio de segundo grado, podemos aplicar la formula general. Sin embargo, debido al error potencial de redondeo, usaremos una formulación alternativa.

Ejemplo.

Use el método de Müller con los valores iniciales de 4.5, 5.5 y 5 para determinar la raíz de la ecuación $f(x) = x^3 - 13x - 12$.

x0	x1	x2	f(x0)	f(x1)	f(x2)	x3
4.50000	5.50000	5.00000	20.62500	82.87500	48.00000	3.97649
5.50000	5.00000	3.97649	82.87500	48.00000	-0.81633	4.00105
5.00000	3.97649	4.00105	48.00000	-0.81633	0.03678	4.00000
3.97649	4.00105	4.00000	-0.81633	0.03678	0.00002	4.00000

Implementación en Java.

```
/**
 * <p>Title: Metodo de Muller</p>
 *  <p>Description:  Resuelve  un  ecuación  haciendo  una  aproximacion
cuadratica</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: UMSNH</p>
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

public class ej058 {
  public static void main(String[] args) {
    Muller();
  }

  static public void Muller()
  {
    double x0 = 4.5, x1 = 5.5, x2 = 5.0, x3;
    double h0, h1, d0, d1, A, B, C;
    double den, raiz;

    do
    {

      h0 = x1 - x0;
      h1 = x2 - x1;
      d0 = (f(x1) - f(x0)) / h0;
      d1 = (f(x2) - f(x1)) / h1;

      A = (d1 - d0) / (h1 + h0);
      B = A * h1 + d1;
      C = f(x2);
```

```
      raiz = Math.sqrt(B * B - 4.0 * A * C);

      if (Math.abs(B + raiz) > Math.abs(B - raiz))
        den = B + raiz;
      else
        den = B - raiz;

      x3 = x2 - 2 * C / den;
      System.out.println(" x = " + x3 + " " + f(x3));

      x0 = x1;
      x1 = x2;
      x2 = x3;
    }while (Math.abs(f(x3)) > 0.000001);
  }
  static public double f(double x)
  {
    return(x*x*x - 13*x -12);
  }
}
```

Regresar.

Método de Bairstow.

El método de Bairstow es un método iterativo, basado en el método de Müller y de Newton Raphson. Dado un polinonio $f_n(x)$ se encuentran dos factores, un polinomio cuadrático $f_2(x) = x^2 - rx - s$ y $f_{n-2}(x)$. El procedimiento general para el método de Bairstow es:

1. Dado $f_n(x)$ y r_0 y s_0
2. Utilizando el método de NR calculamos $f_2(x) = x^2 - r_0x - s_0$ y $f_{n-2}(x)$, tal que, el residuo de $f_n(x)/f_2(x)$ sea igual a cero.
3. Se determinan la raíces $f_2(x)$, utilizando la formula general.
4. Se calcula $f_{n-2}(x) = f_n(x)/f_2(x)$.
5. Hacemos $f_n(x) = f_{n-2}(x)$
6. Si el grado del polinomio es mayor que tres regresamos al paso 2
7. Si no terminamos

La principal diferencia de este método, respecto a otros, es que permite calcular todas las raíces de un polinomio (reales e imaginarias).

Para calcular la división de polinomios, hacemos uso de la división sintética. Así dado

$$f_n(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x + a_0$$

Al dividir entre $f_2(x) = x^2 - rx - s$, tenemos como resultado el siguiente polinomio

$$f_{n-2}(x) = b_nx^{n-2} + b_{n-1}x^{n-3} + \ldots + b_3x + b_2$$

con un residuo $R = b_1(x-r) + b_0$, el residuo será cero solo si b_1 y b_0 lo son.

Los términos *b*, los calculamos utilizamos división sintética, la cual puede resolverse utilizando la siguiente relación de recurrencia

$$b_n = a_n$$
$$b_{n-1} = a_{n-1} + rb_n$$
$$b_i = a_i + rb_{i+1} + sb_{i+2}$$

Una manera de determinar los valores de r y s que hacen cero el residuo es utilizar el Método de Newton-Raphson. Para ello necesitamos una aproximación lineal de b_1 y b_0 respecto a r y s la cual calculamos utilizando la serie de Taylor

donde los valores de *r* y *s* están dados y calculamos los incrementos *dr* y *ds* que hacen a $b_1(r+dr, s+ds)$ y $b_0(r+dr, s+dr)$ igual a cero. El sistema de ecuaciones que tenemos que resolver es:

Bairtow muestra que las derivadas parciales pueden obtener haciendo un procedimiento similar a la división sintética, así

$$c_n = b_n$$
$$c_{n-1} = b_{n-1} + rc_n$$
$$c_i = b_i + rc_{i+1} + sc_{i+2}$$

donde

Sustituyendo término

Ejemplo 1

Dado el polinomio $f_5(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25$, determinar los valores de r y s que hacen el resido igual a cero. Considere $r_0 = -1$ y $s_0 = 2$.

Solución.

Iteración 1.

La división sintética con el polinomio $f_2(x) = x^2 - x + 2.0$ da como resultado

$f3(x) = x^3 - 4.5x^2 + 9.25x - 16.125$ $\quad$ $Residuo = \{30.75, -61.75\}$

Aplicando el método de Newton tenemos

-43.875	16.75	dr	-30.75
108.125	-43.875	ds	61.75

de donde

$r_1 = -1.0 + 2.7636812508572213 = 1.7636812508572213$
$s_1 = 2.0 + 5.403374022767796 = 7.403374022767796$

Iteración 2.

La división sintética con el polinomio $f_2(x) = x^2 - 1.7636812508572213x - 7.403374022767796$ da como resultado

$f3(x) = x^3 - 1.7363187491427787x^2 + 7.091061199392814x - 1.776754563401905$

$Residuo = \{51.75640698828836, 105.68578319650365\}$

Aplicando el método de Newton tenemos

27.628006	14.542693	dr	-51.75640

208.148405	27.62800	ds	-105.68578

de donde

$r_2 = 1.7636812508572213 - 0.04728019113442016 = 1.7164010597228012$
$s_2 = 7.403374022767796 - 3.469106187802152 = 3.934267834965644$

Iteración 3.

La división sintética con el polinomio $f_2(x) = x^2 - 1.7164010597228012x - 3.934267834965644$ da como resultado

$f3(x) = x^3 - 1.7835989402771988x^2 + 3.622896723753395x + 1.3261878347051992$

Residuo = {12.654716254544885, 28.1881465309956}

Aplicando el método de Newton tenemos

13.83497	7.44182	dr	-12.65471
65.679212	13.83497	ds	-28.18814

de donde

$r_3 = 1.7164010597228012 - 0.11666951305731528 = 1.599731546665486$
$s_3 = 3.934267834965644 - 1.4835870659929915 = 2.4506807689726524$

En resumen

k	r	s	Residuo	
0	-1	2	30.75	-61.75
1	1.76368	7.403374	51.756406	105.68578
2	1.71640	3.93426	12.65471	28.18814
3	1.599731	2.450680	2.89958	8.15467
4	1.33354	2.18666	0.760122	2.522228
5	1.11826	2.11302	0.271940	0.607688
6	1.02705	2.02317	0.04313	0.11185
7	1.00165	2.00153	0.00277	0.00634
8	1.00000	2.00000	1.13930E-5	2.67534E-5

La solución es:

$f_3(x) = x^3 - 2.53x^2 + 2.25x - 0.625$ y $f_2(x) = x^2 - x - 2$

Las raíces *de* $f_2(x) = x^2 - x - 2$, son

x1 = 2
x2 = -1

Si repetimos el ejemplo pero ahora considerando el polinomio $f_3(x) = x^3 - 2.53x^2 + 2.25x - 0.625$, podemos calcular el total de las raíces del polinomio original.

Ejemplo 2

Dado el polinomio $f_5(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25$, determinar las raíces de este polinomio. Considere $r_0 = -1$ y $s_0 = -1$.

Paso 1.

$f_5(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25$

$f_5(x) =(x^3 - 4x^2 + 5.25x - 2.5)*(x^2 +0.5x - 0.5)$

Las raíces de $x^2 +0.5x - 0.5=0$ son

$x_1 = 0.5$
$x_2 =-1.0$

Paso 2.

$f_3(x) = x^3 - 4x^2 + 5.25x - 2.5$

$f_3(x) =(x - 2)*(x^2 - 2x +1.25)$

Las raíces de $x^2 - 2x +1.25=0$ son

$x_3 = 1.0 + j0.5$
$x_4 =-1.0 - j0.5$

Paso 3

$f_1(x) =(x - 2)$

La raíz de este polinomio es

$x_5 = 2;$

Todas la raíces de $f_5(x)$ son $x = [0.5, 1.0, (1.0 + j0.5), (1 - j0.5), 2]$

Implementación en Java.

```
package ejemplos;
```

```
/**
 * <p>Title: Meétodo de Baritow </p>
 *  <p>Description: Calcula todas las raices de un polinomio de grado
n</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: UMSNH</p>
 * @author Dr. Felix Calderon Solorio
 * @version 1.0
 */

public class ej065 {
  public static void main(String[] args) {
    //double a[] = {-2.5, 5.25, -4, 1};
    double a[] ={1.25, -3.875, 2.125, 2.75, -3.5, 1};
    int n = a.length;
    double re[] = new double[n];
    double im[] = new double[n];

    Bairstow(a, -1, -1, re, im);
    //Bairstow(a, -1, 2, re, im);
  }

  static public void Bairstow(double a[], double r0, double s0, double
re[], double im[])
  {
    int n = a.length, iter =0;
    double b[] = new double[n], c[] = new double[n];
    double ea1 = 1, ea2 = 1, T = 0.00001;
    double r=r0, s=s0,det, ds, dr;
    int MaxIter = 100, i;

   for(iter=0; iter< MaxIter && n>3; iter++)
    {
      do {
        Division_Derivada(a, b, c, r, s, n);

/*
        System.out.println("Solucionando ");
        System.out.print("fn(x) = "); imprime(a, n);
        System.out.print("fn-2(x) = "); imprime2(b, n);
        System.out.println("f2(x) = x2 + " + r + "x + " + s);
        System.out.println(c[2] + " " + c[3] + " = " + -b[1]);
        System.out.println(c[1] + " " + c[2] + " = " + -b[0]);
*/
        det = c[2]*c[2] - c[3]*c[1];
        if(det!=0)
        {
          dr = (-b[1]*c[2] + b[0]*c[3])/det;
          ds = (-b[0]*c[2] + b[1]*c[1])/det;
          /*
          System.out.println("*********************************");
          System.out.println(r+dr +" = " + r + " + " + dr);
          System.out.println(s+ds +" = " + s + " + " + ds);
          System.out.println("---------------------------------");*/
          r = r+dr;
          s = s+ds;
```

```
          if(r!=0) ea1 = Math.abs(dr/r)*100.0;
          if(s!=0) ea2 = Math.abs(ds/s)*100.0;
        }
        else
        {
          r = 5*r+1;
          s = s+1;
          iter = 0;
        }
      }
      while ((ea1 > T) && (ea2 > T));
      raices(r, s, re, im, n);
      System.out.println("iter " +iter);
      System.out.print("fn(x) = "); imprime(a, n);
      System.out.print("fn-2(x) = "); imprime2(b, n);
      System.out.println("f2(x) = x2 -(" +r + ")x -(" +s +")");
      n = n-2;
      for(i=0; i<n; i++)
        a[i] = b[i+2];
      if (n < 4) break;
    }
    if(n==3)
    {
      System.out.println("n = " + n);
      r = -a[1]/a[2];
      s = -a[0]/a[2];
      imprime(a, n);
      raices(r, s, re, im, n);
    }
    else
    {
      re[n-1] = -a[0]/a[1];
      im[n-1] = 0;
    }
    for(i=1; i<re.length; i++)
      System.out.println( "X["+i+"]= " + re[i] + " j " + im[i]);
  }

  public static void Division_Derivada(double a[], double b[], double
c[], double r, double s, int n)
  {
    int i;

    b[n-1] = a[n-1];
    b[n-2] = a[n-2] + r*b[n-1];

    c[n-1] = b[n-1];
    c[n-2] = b[n-2] + r*c[n-1];

    for(i=n-3; i>=0; i--)
    {
      b[i] = a[i] + r*b[i+1] + s*b[i+2];
      c[i] = b[i] + r*c[i+1] + s*c[i+2];
    }

  }
```

```
  public static void imprime(double x[], int n)
  {
    int i;

    for (i = n - 1; i >= 0; i--)
      if(x[i] > 0) System.out.print("+ " +x[i] + "x"+i+" ");
      else System.out.print("- " + -x[i] + "x"+i+" ");
    System.out.println("");
  }

  public static void imprime2(double x[], int n)
  {
    int i;

    for (i = n - 1; i >= 2; i--)
      if(x[i] > 0) System.out.print("+ " +x[i] + "x"+(i-2)+" ");
      else System.out.print("- " + -x[i] + "x"+(i-2)+" ");
    System.out.println("Residuo = {"+ x[1]+ ", " + x[0] + "}");

  }

  public static void raices(double r, double s, double re[], double im[],
int n)
  {
    double d = r*r + 4*s;
    if(d > 0)
    {
      re[n-1] = (r + Math.sqrt(d))/2.0;
      re[n-2] = (r - Math.sqrt(d))/2.0;
      im[n-1] = 0;
      im[n-2] = 0;
    }
    else
    {
      re[n-1] = r/2.0;
      re[n-2] = re[n-1];
      im[n-1] = Math.sqrt(-d)/2.0;
      im[n-2] = -im[n-1];
    }
  }
}
```

Regresar.

Eliminación Gaussiana

Consideremos que tenemos un sistema lineal Ax=b, donde la matriz A no tiene las condiciones de ser triangular superior.

a_{11}	a_{12}	a_{13}		x_1		b_1
a_{21}	a_{22}	a_{23}		x_2	=	b_2
a_{31}	a_{32}	a_{33}		x_3		b_3

Comenzaremos por despejar de la x_1 de la ecuación (I)

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}}$$

y sustituimos en las ecuaciones II

$$a_{21}\frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} + a_{22}x_2 + a_{23}x_3 = b_2$$

agrupando términos semejantes tenemos:

$$(a_{22} - a_{21}\, a_{12}/a_{11})x_2 + (a_{23} - a_{21}\, a_{13}/a_{11})\, x_3 = (b_2 - a_{21}\, b_1/a_{11})$$

Ahora sustituimos en la ecuación III

$$a_{31}\frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} + a_{32}x_2 + a_{33}x_3 = b_3$$

$$(a_{32} - a_{31}\, a_{12}/a_{11})x_2 + (a_{33} - a_{31}\, a_{13}/a_{11})\, x_3 = (b_3 - a_{31}\, b_1/a_{11})$$

Lo cual nos da un nuevo sistema simplificado dada por

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \quad (I) \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \quad (II') \\ a'_{32}x_2 + a'_{33}x_3 &= b'_3 \quad (III') \end{aligned}$$

donde los valores de a'_{ij} los calculamos como:

$$a'_{ij} = a_{ij} - a_{ik}\, a_{kj/}\, a_{kk}$$

$$b'_i = b_i - a_{ik}\, b_{k/}\, a_{kk}$$

Si repetimos el procedimiento tendremos un sistema dado como:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \quad (I) \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \quad (II') \\ a''_{33}x_3 &= b'_3 \quad (III'') \end{aligned}$$

que corresponde a un sistema triangular superior que podemos solucionar utilizando sustitución hacia atrás.

Ejemplo.

Calcular el sistema triangular superior utilizando eliminación gaussiana.

$$5x_1 + 2x_2 + 1x_3 = 3$$

$$2x_1 + 3x_2 - 3x_3 = -10$$
$$1x_1 - 3x_2 + 2x_3 = 4$$

Primer paso

$$5x + 2y + 1z = 3$$
$$0 + (11/5)y - (17/5)z = -(56/5)$$
$$0 - (17/5)y + (9/5)z = (17/5)$$

Segundo paso

$$5x + 2y + 1z = 3$$
$$0x + (11/5)y - (17/5)z = -(56/5)$$
$$0x - 0y - (38/11)z = -(153/11)$$

Ejemplo.

Resolver el sistema de ecuaciones

3	-1	-1	x		0
-1	1	0	y	=	1
-1	0	1	z		1

Aplicando eliminación gaussiana nos queda.

3	-1	-1	x		0
0	2/3	-1/3	y	=	1
0	0	1/2	z		3/2

Ejemplo.

Un ejemplo de sistema donde es necesario hacer un cambio de renglón por renglón para que tenga solución es el siguiente sistema.

1	2	6
4	8	-1
-2	3	5

Aplicando el primer paso de la eliminación gaussiana tenemos:

1	2	6
0	0	-25
0	7	17

note, que aparece un cero en el elemento 22, lo cual nos da un sistema sin solución. Permutando los renglones 2 y 3 el sistema tiene solución.

1	2	6
-2	3	5
4	8	-1

Regresar.

Factorización triangular.

Supongamos que la matriz de coeficientes A de un sistema lineal $Ax=b$ admite una factorización triangular como la siguiente.

a_{00}	a_{01}	a_{02}	=	1	0	0	*	u_{00}	u_{01}	u_{02}
a_{10}	a_{11}	a_{12}		l_{10}	1	0		0	u_{11}	u_{12}
a_{20}	a_{21}	a_{22}		l_{20}	l_{21}	1		0	0	u_{22}

Entonces la solución la podemos plantear como

$[LU]x = b$

El cual se puede solucionar resolviendo primero $Ly=b$ y luego el sistema $Ux=y$. Estos sistemas pueden resolverse utilizando sustitución hacia atrás y sustitución hacia delante.

Sustitución hacia delante.

La formula para calcular la sustitución hacia delante es:

La implementación en Java es:

```
void sustitucion_hacia_delante(double A[][], double y[], double b[]) {
    int k, i, j, n;
    double suma;

    n = A.length;
    for (k = 0; k < n; k++) {
      suma = 0;
      for (j = 0; j < k; j++) {
        suma += A[k][j] * y[j];
      }
      y[k] = b[k] - suma;
    }
  }
```

Sustitución hacia atrás.

La formula para calcular la sustitución hacia atrás es:

La implementación en Java es:

```
static public void sustitucion_hacia_atras(double a[][], double x[], double b[])
{
  double suma;
  int k, j;
  int n = x.length;

  for (k = n - 1; k >= 0; k--) {
    suma = 0;
    for (j = k + 1; j < n; j++)
      suma += a[k][j] * x[j];
    x[k] = (b[k] - suma) / a[k][k];
  }
}
```

Ejemplo.

Resolver el sistema de ecuaciones

$x_0 + 2x_1 + 4x_2 + x_3 = 21$
$2x_0 + 8x_1 + 6x_2 + 4x_3 = 52$
$3x_0 + 10x_1 + 8x_2 + 8x_3 = 79$
$4x_0 + 12x_1 + 10x_2 + 6x_3 = 82$

$$
A = \begin{vmatrix} 1 & 2 & 4 & 1 \\ 2 & 8 & 6 & 4 \\ 3 & 10 & 8 & 8 \\ 4 & 12 & 10 & 6 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 \\ 4 & 1 & 2 & 1 \end{vmatrix} * \begin{vmatrix} 1 & 2 & 4 & 1 \\ 0 & 4 & -2 & 2 \\ 0 & 0 & -2 & 3 \\ 0 & 0 & 0 & -6 \end{vmatrix}
$$

Primero resolvemos el sistema de ecuaciones

$$
\begin{aligned}
y_0 &= 21 \\
2y_0 + y_1 &= 52 \\
3y_0 + y_1 + y_2 &= 79 \\
4y_0 + y_1 + 2y_2 + y_3 &= 82
\end{aligned}
$$

El cual corresponde a un sistema triangular inferior

$y_0 = 21$
$y_1 = 10$
$y_2 = 6$
$y_3 = -24$

En general

$x_k = (b_k - S_{j=1,k-1}\ a_{kj}\ x_j)/a_{kk}$

y posteriormente solucionamos

$$
\begin{aligned}
x_0 + 2x_1 + 4x_2 + x_3 &= 21 \\
4x_1 - 2x_2 + 2x_3 &= 10 \\
-2x_2 + 3x_3 &= 6 \\
-6x_3 &= -24
\end{aligned}
$$

Utilizando sustitución hacia atrás tenemos

$X = [1,2,3,4]^T$

Implementación utilizando eliminación Gaussiana..

Como vimos una matriz factorizada, fácilmente puede ser resuelta utilizando los métodos de sustitución hacia adelante y sustitución hacia atrás. ¿Pero como hacemos la factorización? Comenzaremos por escribir el sistema

a_{00}	a_{01}	a_{02}
a_{10}	a_{11}	a_{12}
a_{20}	a_{21}	a_{22}

$=$

1	0	0
l_{10}	1	0
l_{20}	l_{21}	1

$*$

u_{00}	u_{01}	u_{02}
0	u_{11}	u_{12}
0	0	u_{22}

$$A = LU$$

Si multiplicamos el primer renglón de L por la primera columna de U tenemos

$$u_{00} = a_{00}$$
$$u_{01} = a_{01}$$
$$u_{02} = a_{02}$$

Ahora hacemos lo mismo para el segundo renglón de L

$$l_{10}\, u_{00} = a_{10} \qquad l_{10} = a_{10}/u_{00}$$
$$l_{10}\, u_{01} + u_{11} = a_{11} \qquad u_{11} = a_{11} - l_{10}\, u_{01}$$
$$l_{10}\, u_{02} + u_{12} = a_{12} \qquad u_{12} = a_{12} - l_{10}\, u_{02}$$

Multiplicando el tercer renglón de L

$$l_{20}\, u_{00} = a_{20} \qquad l_{20} = a_{20}/u_{00}$$
$$l_{20}\, u_{01} + l_{21}\, u_{11} = a_{21} \qquad l_{21} = (a_{21} - l_{20}\, u_{01})/a_{11}$$
$$l_{20}\, u_{02} + l_{21}\, u_{12} + u_{22} = a_{22} \qquad u_{22} = a_{22} - l_{20}\, u_{02} - l_{21}\, u_{12}$$

Podemos notar lo siguiente, en este caso la triangulación la podemos hacer en dos pasos.

Paso I. Dado la matriz A hacemos

a_{00}	a_{01}	a_{02}
a_{10}/a_{00}	$a_{11} - a_{10}\, a_{10}/a_{00}$	$a_{12} - a_{10}\, a_{02}/a_{00}$
a_{20}/a_{00}	$a_{21} - a_{20}\, a_{01}/a_{00}$	$a_{22} - a_{20}\, a_{02}/a_{00}$

a_{00}	a_{01}	a_{02}
a'_{10}	a'_{11}	a'_{12}
a'_{20}	a'_{21}	a'_{22}

Paso II. Dado A modificada

a_{00}	a_{01}	a_{02}
a'_{10}	a'_{11}	a'_{12}
a'_{20}	a'_{21}/ a'_{11}	$a'_{22} - a'_{21}a'_{12}/a'_{11}$

a_{00}	a_{01}	a_{02}
a'_{10}	a'_{11}	a'_{12}
a'_{20}	a''_{21}	a''_{22}

Note que para calcular la matriz triangular superior se podría aplicar, el método de eliminación Gaussiana y la matriz diagonal inferior es simplemente los cocientes

$$a_{jk} = a_{jk}/a_{kk}$$

La implementación en Java es:

```
static public void LU_EG(double a[][])
{
  int n = a.length;
  double factor;

  for (int k = 0; k <= n - 2; k++) {
    for (int i =  k+1; i < n; i++) {
      factor = a[i][k]/a[k][k];
      a[i][k] = factor;
      for (int j = k+1; j < n; j++)
        a[i][j] -= factor * a[k][j];
    }
  }
}
```

Factorización de Crout.

Consideremos un sistema más grande, por ejemplo de 4 ecuaciones con cuatro incógnitas.

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

=

1	0	0	0
l_{10}	1	0	0
l_{20}	l_{21}	1	1
l_{30}	l_{31}	l_{32}	1

*

u_{00}	u_{01}	u_{02}	u_{03}
0	u_{11}	u_{12}	u_{13}
0	0	u_{22}	u_{23}
0	0	0	u_{33}

Si hacemos las multiplicaciones indicadas y dejamos los valores en la misma matriz tenemos:

Para el primer renglón de la matriz *(i=0)*

arriba de la diagonal

$j=0$ $a_{(0,0)} = a_{(0,0)}$

$j=1$ $a_{(0,1)} = a_{(0,1)}$

$j=2$ $a_{(0,2)} = a_{(0,2)}$

$j=3$ $a_{(0,3)} = a_{(0,3)}$

Para el segundo renglón *(i=1)*

bajo la diagonal

$j=0$ $a_{(1,0)} = a_{(1,0)} / a_{(0,0)}$

arriba de la diagonal

$j=1$ $a_{(1,1)} = a_{(1,1)} - a_{(1,0)}a_{(0,1)}$

$j=2$ $a_{(1,2)} = a_{(1,2)} - a_{(1,0)}a_{(0,2)}$

$j=3$ $a_{(1,3)} = a_{(1,3)} - a_{(1,0)}a_{(0,3)}$

Tercer renglón *(i=2)*

bajo la diagonal

$j=0$ $a_{(2,0)} = a_{(2,0)} / a_{(0,0)}$

$j=1$ $a_{(2,1)} = (a_{(2,1)} - a_{(2,0)}a_{(0,1)}) / a_{(1,1)}$

arriba de la diagonal

$j=2$ $a_{(2,2)} = a_{(2,2)} - a_{(2,0)}\, a_{(0,2)} - a_{(2,1)}a_{(1,2)}$

$j=3$ $a_{(2,3)} = a_{(2,3)} - a_{(2,0)}a_{(0,3)} - a_{(2,1)}a_{(1,3)}$

Y finalmente para el cuarto renglón *(i=3)*

bajo la diagonal

$j=0$ $a_{(3,0)} = a_{(3,0)} / a_{(0,0)}$

$j=1$ $a_{(3,1)} = (a_{(3,1)} - a_{(3,0)}\, a_{(0,1)}) / a_{(1,1)}$

$j=2$ $a_{(3,2)} = (a_{(3,2)} - a_{(3,0)}\, a_{(0,2)} - a_{(3,1)}\, a_{(1,2)}) / a_{(2,2)}$

arriba de la diagonal

$j=3$ $a_{(3,3)} = a_{(3,3)} - a_{(3,0)}\, a_{(0,3)} - a_{(3,1)}\, a_{(1,3)} - a_{(3,2)}a_{(2,3)}$

Podemos ver, que cualquier elemento bajo la diagonal se calcula como:

para todo $i=0,...,n-1$ y $j = 0,...,i-1$.

Para los términos arriba de la diagonal

para todo $i=0,...,n-1$ y $j = i,...,n-1$.

La implementación en Java es:

```
static public void LU(double a[][])
{
  int n = a.length;
  int i, j, k;
  double suma;

  for (i = 0; i<n; i++) {

    for (j = 0; j <= i-1; j++) {
      suma = 0;
      for (k = 0; k <= j-1; k++)
        suma += a[i][k] * a[k][j];
      a[i][j] = (a[i][j] - suma)/ a[j][j];
    }

    for (j = i; j < n; j++) {
      suma = 0;
      for (k = 0; k <= i-1; k++)
        suma += a[i][k] * a[k][j];
      a[i][j] = a[i][j] - suma;
    }
  }
}
```

Ejemplo:

Hacer la factorización LU de la siguiente matriz, utilizando eliminación Gaussiana. Corroborar utilizando Factorización de Crout.

1	2	4	1
2	8	6	4
3	10	8	8
4	12	10	6

Paso 1.

1	2	4	1
2	4	-2	2
3	4	-4	5
4	4	-6	2

Paso 2.

1	2	4	1
2	4	-2	2
3	1	-2	3
4	1	-4	0

Paso 3.

1	2	4	1
2	4	-2	2
3	1	-2	3
4	1	-2	6

Regresar.

Calculo de matriz inversa.

Inversa de Shiplay.

Consideremos el siguiente sistema de ecuaciones

a_{11}	a_{12}	a_{13}		x_1		b_1
a_{21}	a_{22}	a_{23}		x_2	=	b_2
a_{31}	a_{32}	a_{33}		x_3		b_3

de la ecuación I despejamos el valor de la variable x_1

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$$

Reorganizando la ecuación tenemos

$$(b_1/a_{11}) - (a_{12}/a_{11})x_2 - (a_{13}/a_{11})x_3 = x_1$$

Sustituimos el valor de x_1 en la ecuación II:

$$(b_1 - a_{12}x_2 - a_{13}x_3)*a_{21}/a_{11} + a_{22}x_2 + a_{23}x_3 = b_2$$

$$(a_{21}/a_{11})b_1 + (a_{22} - a_{21}\,a_{12}/a_{11})x_2 + (a_{23} - a_{21}\,a_{13}/a_{11})x_3 = b_2$$

y finalmente en III tenemos :

$$(a_{31}/a_{11})b_1 + (a_{32} - a_{31}\,a_{12}/a_{11})x_2 + (a_{33} - a_{31}\,a_{13}/a_{11})x_3 = b_3$$

En forma matricial estas ecuaciones lucen como:

$1/a_{11}$	$-a_{12}/a_{11}$	$-a_{13}/a_{11}$		b_1		x_1
a_{21}/a_{11}	$a_{22}-a_{21}\,a_{12}/a_{11}$	$a_{23}-a_{21}\,a_{13}/a_{11}$		x_2	=	b_2
a_{31}/a_{11}	$a_{32}-a_{31}\,a_{12}/a_{11}$	$a_{33}-a_{31}\,a_{13}/a_{11}$		x_3		b_3

Lo que nos da un sistema equivalente como el siguiente.

a'_{11}	a'_{12}	a'_{13}		b_1		x_1
a'_{21}	a'_{22}	a'_{23}		x_2	=	b_2
a'_{31}	a'_{32}	a'_{33}		x_3		b_3

y los valores lo calculamos haciendo

Dado un valor de k al que llamamos pivote:

$a_{ij} = a_{ij} - a_{ik}\, a_{kj}/a_{kk}$ para todos los valores de i y j diferentes de k

$a_{ik} = a_{ik}/a_{kk}$ para todos los i diferentes de k

$a_{kj} = a_{kj}/a_{kk}$ para todos los j diferentes de k

$a_{kk} = 1/a_{kk}$

Si repetimos el procedimiento suponiendo un valor de k = 2 tendremos un sistema.

$a'_{11}-a'_{12}a'_{21}/a'_{22}$	a'_{12}/a'_{22}	$a'_{13}-a'_{12}a'_{23}/a'_{22}$		b_1		x_1
$-a'_{21}/a'_{22}$	$1/a'_{22}$	$-a'_{23}/a'_{22}$		b_2	=	x_2
$a'_{31}-a'_{32}a'_{21}/a'_{22}$	a'_{32}/a'_{22}	$a'_{33}-a'_{32}a'_{23}/a'_{22}$		x_3		b_3

Lo que nos da un sistema equivalente como el siguiente.

a''_{11}	a''_{12}	a''_{13}		b_1		x_1
a''_{21}	a''_{22}	a''_{23}		b_2	=	x_2
a''_{31}	a''_{32}	a''_{33}		x_3		b_3

Finalmente si volvemos a repetir el método tendremos

a'''_{11}	a'''_{12}	a'''_{13}		b_1		x_1
a'''_{21}	a'''_{22}	a'''_{23}		b_2	=	x_2
a'''_{31}	a'''_{32}	a'''_{33}		b_3		x_3

Ejemplo:

Determinar la matriz inversa de

3	-1	-1
-1	1	0
-1	0	1

Primer paso:

1/3	1/3	1/3

-1/3	2/3	-1/3
-1/3	-1/3	2/3

Segundo paso:

½	1/2	1/2
½	3/2	1/2
-1/2	-1/2	1/2

Tercer paso:

1	1	1
1	2	1
1	1	2

Regresar.

Matrices Especiales.

Una matriz bandada es una matriz cuadrada en la que todos sus elementos son cero, con excepción de una banda sobre la diagonal principal. Un sistema tridiagonal (es decir con un ancho de banda de 3) se puede expresar en forma general como:

$$\begin{bmatrix} f_0 & g_0 & & \\ e_1 & f_1 & g_1 & \\ & e_2 & f_2 & g_2 \\ & & e_3 & f_3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Basados en la descomposición LU podemos ver que el algoritmo de Thomas es:

La sustitución hacia adelante es

y hacia atrás es:

Ejemplo:

Resuelva el siguiente sistema tridiagonal por medio del algoritmo de Thomas.

2.04	*-1*			x_0		*40.8*
-1	*2.04*	*-1*		x_1	=	*0.8*
	-1	*2.04*	*-1*	x_2		*0.8*
		-1	*2.04*	x_3		*200.8*

La solución de la descomposición triangular es:

2.04	*-1*		
-0.49	*1.55*	*-1*	
	-0.645	*1.395*	*-1*
		-0.717	*1.323*

La solución del sistema es:

$$X = [65.970, 93.778, 124.538, 159.480]^T$$

Descomposición de Cholesky.

Este algoritmo se basa en el hecho de que una matriz simétrica se puede descomponer en $[A] = [L][L]^T$, dado que la matriz [A] es una matriz simétrica. En este caso aplicaremos la eliminación de Crout a la parte inferior de la matriz y la parte superior, simplemente tendrá los mismos valores.

Así tomando las ecuaciones para la factorización LU la podemos adaptar como:

Podemos ver, que cualquier elemento bajo la diagonal se calcula como:

$$a_{i,j} = \frac{a_{i,j} - \sum_{k=0}^{j-1} a_{i,k} a_{k,j}}{a_{j,j}}$$

para todo $i=0,...,n-1$ y $j = 0,...,i-1$.

Para los términos arriba de la diagonal, en este caso solo la diagonal

para todo *i=0,...,n-1*.

La implementación en Java es:

```
  static public void Cholesky(double A[][]) {
    int i, j, k, n, s;
    double fact, suma = 0;

    n = A.length;

    for (i = 0; i < n; i++) { //k = i
      for (j = 0; j <= i - 1; j++) { //i = j
        suma = 0;
        for (k = 0; k <= j - 1; k++) // j = k
          suma += A[i][k] * A[j][k];

        A[i][j] = (A[i][j] - suma) / A[j][j];
      }

      suma = 0;
      for (k = 0; k <= i - 1; k++)
        suma += A[i][k] * A[i][k];
      A[i][i] = Math.sqrt(A[i][i] - suma);
    }
  }
```

Regresar.

Métodos Iterativos para solucionar sistemas lineales.

Método iterativo de Jacobi.

Consideremos el siguiente sistema de ecuaciones.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Vamos a representar cada una de las variables en términos de ellas mismas.

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}}$$

$$x_2 = \frac{b_2 - a_{21}x_1 - a_{23}x_3}{a_{22}}$$

$$x_3 = \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{22}}$$

Lo cual nos sugiere el siguiente esquema iterativo de solución.

$$x_1(t+1) = \frac{b_1 - a_{12}x_2(t) - a_{13}x_3(t)}{a_{11}}$$

$$x_2(t+1) = \frac{b_2 - a_{21}x_1(t) - a_{23}x_3(t)}{a_{22}}$$

$$x_3(t+1) = \frac{b_1 - a_{31}x_1(t) - a_{32}x_2(t)}{a_{33}}$$

En general podemos escribir como

$$x_k(t+1) = \frac{b_1 - \Sigma_{i=1,\ i\ !=k,\ n}\ a_{ki}x_i(t)}{a_{kk}}$$

Algoritmo iterativo de Gauss-Seidel.

El cambio que debemos hacer respecto al de Jacobi, es que las variables nuevas son utilizadas una vez que se realiza el cálculo de ellas así, para un sistema de tres ecuaciones tendremos:

$$x_1(t+1) = \frac{b_1 - a_{12}x_2(t) - a_{13}x_3(t)}{a_{11}}$$

$$x_2(t+1) = \frac{b_2 - a_{21}x_1(t+1) - a_{23}x_3(t)}{a_{22}}$$

$$x_3(t+1) = \frac{b_1 - a_{31}x_1(t+1) - a_{32}x_2(t+1)}{a_{33}}$$

Ejemplo.

Resolver el siguiente sistema de ecuaciones utilizando el método de Jacobi y comparar con el método de Gauss-Seidel.

$$\begin{bmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ -21 \\ 15 \end{bmatrix}$$

Las primeras 20 iteraciones del algoritmo de Jacobi son :

k	x(1)	x(2)	x(3)
1	1.0000	2.0000	2.0000
2	1.7500	3.3750	3.0000
3	1.8438	3.8750	3.0250
4	1.9625	3.9250	2.9625
5	1.9906	3.9766	3.0000
6	1.9941	3.9953	3.0009
7	1.9986	3.9972	2.9986
8	1.9996	3.9991	3.0000
9	1.9998	3.9998	3.0000
10	1.9999	3.9999	2.9999
11	2.0000	4.0000	3.0000
12	2.0000	4.0000	3.0000
13	2.0000	4.0000	3.0000
14	2.0000	4.0000	3.0000
15	2.0000	4.0000	3.0000
16	2.0000	4.0000	3.0000
17	2.0000	4.0000	3.0000
18	2.0000	4.0000	3.0000
19	2.0000	4.0000	3.0000
20	2.0000	4.0000	3.0000

La solución utilizando Gauss-Seidel es :

k	x(0)	x(1)	x(3)
1	1.0000	2.0000	2.0000
2	1.7500	3.7500	2.9500
3	1.9500	3.9688	2.9863
4	1.9956	3.9961	2.9990
5	1.9993	3.9995	2.9998
6	1.9999	3.9999	3.0000
7	2.0000	4.0000	3.0000
8	2.0000	4.0000	3.0000
9	2.0000	4.0000	3.0000
10	2.0000	4.0000	3.0000
11	2.0000	4.0000	3.0000
12	2.0000	4.0000	3.0000
13	2.0000	4.0000	3.0000

14	2.0000	4.0000	3.0000
15	2.0000	4.0000	3.0000
16	2.0000	4.0000	3.0000
17	2.0000	4.0000	3.0000
18	2.0000	4.0000	3.0000
19	2.0000	4.0000	3.0000
20	2.0000	4.0000	3.0000

Note que Gauss-Seidel requiere de 7 iteraciones mientras Jacobi de 11, para convergir.

Regresar.

Búsqueda de la sección dorada

Al resolver para la raíz de sólo una ecuación no lineal, la meta fue la de encontrar la variable *x* que diera cero en la función *f(x)*. La optimización de una sola variable tiene como meta encontrar el valor de x que generará un extremo, ya sea un máximo o un mínimo de *f(x)*.

La búsqueda de la sección dorada es una técnica simple de búsqueda de una sola variable de propósito general. Es similar en esencia al enfoque de la bisección para localizar raíces

La clave para hacer eficiente este procedimiento es la mejor elección de los puntos intermedios. Esta meta se puede alcanzar al especificar que las siguientes dos condiciones se cumplan.

Sustituyendo la primer ecuación en la segunda tenemos:

Si tomamos el reciproco y $R = l_2/l_1$, se llega a

$$1 + R = 1/R$$
$$R^2 + R - 1=0$$

la cual se puede resolver para la raíz positiva

Algoritmo

1.- Dados dos puntos iniciales x_l y x_u, tales que xu > xl y exista un máximo.

2.- Se escogen dos puntos interiores x_1 y x_2 de acuerdo con la razón dorada,

$d = R(x_u - x_l)$
$x_1 = x_l + d$
$x_2 = x_u - d$

3.- La función se evalúa en los puntos interiores es decir x_1, x_2, x_u, y x_l

- Si $f(x_1)$ es mayor que $f(x_2)$ entonces hacemos $x_l = x_2$;
- Si no $x_u = x_1$;

4.- Repetir los pasos 2 y 3 hasta convergencia.

Ejemplo

Use la búsqueda de la sección dorada para encontrar el máximo de la función $f(x) = 2seno(x) - x^2/10$ en el intervalo $[0,4]$.

i	xL	fL	x2	f2	x1	f1	xU	fU	d
0	0.0000	0.0000	1.5279	1.7647	2.4721	0.6300	4.0000	-3.1136	2.4721
1	0.0000	0.0000	0.9443	1.5310	1.5279	1.7647	2.4721	0.6300	1.5279
2	0.9443	1.5310	1.5279	1.7647	1.8885	1.5432	2.4721	0.6300	0.9443
3	0.9443	1.5310	1.3050	1.7595	1.5279	1.7647	1.8885	1.5432	0.5836
4	1.3050	1.7595	1.5279	1.7647	1.6656	1.7136	1.8885	1.5432	0.3607
5	1.3050	1.7595	1.4427	1.7755	1.5279	1.7647	1.6656	1.7136	0.2229
6	1.3050	1.7595	1.3901	1.7742	1.4427	1.7755	1.5279	1.7647	0.1378
7	1.3901	1.7742	1.4427	1.7755	1.4752	1.7732	1.5279	1.7647	0.0851
8	1.3901	1.7742	1.4226	1.7757	1.4427	1.7755	1.4752	1.7732	0.0526
9	1.3901	1.7742	1.4102	1.7754	1.4226	1.7757	1.4427	1.7755	0.0325
10	1.4102	1.7754	1.4226	1.7757	1.4303	1.7757	1.4427	1.7755	0.0201
11	1.4226	1.7757	1.4303	1.7757	1.4350	1.7757	1.4427	1.7755	0.0124
12	1.4226	1.7757	1.4274	1.7757	1.4303	1.7757	1.4350	1.7757	0.0077
13	1.4226	1.7757	1.4256	1.7757	1.4274	1.7757	1.4303	1.7757	0.0047
14	1.4256	1.7757	1.4274	1.7757	1.4285	1.7757	1.4303	1.7757	0.0029
15	1.4256	1.7757	1.4267	1.7757	1.4274	1.7757	1.4285	1.7757	0.0018
16	1.4267	1.7757	1.4274	1.7757	1.4278	1.7757	1.4285	1.7757	0.0011
17	1.4267	1.7757	1.4271	1.7757	1.4274	1.7757	1.4278	1.7757	0.0007

La implementación en Java es:

```
void Seccion_Dorada()
{
  double xL, xU, x1, x2, fL, fU, f1, f2;
  double d, R = (Math.sqrt(5)-1.0)/2.0;
  int i = 0;

  xL = 0;
```

```
    xU = 4.0;

    System.out.println("i" + "\t" + "xL" + "\t" + "fL" + "\t" + "x2" +
"\t" + "f2" +
                              "\t" + "x1" + "\t" + "f1" + "\t" + "xU" +
"\t" + "fU" +
                              "\t" + "d");
    do
    {
      d  = R*(xU - xL);
      x1 = xL + d;
      x2 = xU - d;

      fU = funcion(xU);
      fL = funcion(xL);
      f1 = funcion(x1);
      f2 = funcion(x2);

      System.out.println(i++ + "\t" + xL + "\t" + fL + "\t" + x2 + "\t" +
f2 +
                                "\t" + x1 + "\t" + f1 + "\t" + xU + "\t" +
fU +
                                "\t" + d);

      if(f1 > f2) xL = x2;
      else xU = x1;


    }while(d > 0.001);

  }
```

Regresar.

Interpolación cuadrática

La interpolación cuadrática tiene la ventaja del hecho que un polinomio de segundo orden con frecuencia proporciona una buena aproximación a la forma de *f(x)* cercana un óptimo.

Este método al igual que el de Muller hacen una aproximación cuadrática de un polinomio, así $f(x) = Ax^2 + Bx + C$. Para esta función el mínimo o máximo se localiza en

Se puede mostrar que después de un manejo algebraico el valor de x_3 que maximiza la ecuación es:

Algoritmo

1.- Dados tres puntos x_0, x_1 y x_2 dentro de los cuales existe un máximo y $x_0 < x_1 < x_2$
2.- Calculamos la aproximación cuadrática y su máximo x_3 utilizando la ecuación anterior.
3.- Si $x_0 < x_3 < x_1$, los nuevos puntos de búsqueda son x_0, x_3 y x_1, si no el intervalo de búsqueda es x_1, x_3 y x_2
4.- Se repiten los pasos hasta que la diferencia entre x_0 y x_2 sea pequeña.

Ejemplo

Use la búsqueda de la sección dorada para encontrar el máximo de la función $f(x) = 2seno(x) - x^2/10$ con los valores iniciales $x_0 = 0$, $x_1 = 1$ y $x_2 = 4$.

i	x0	f0	x1	f1	x2	f2	x3	f3
0	0.0000	0.0000	1.0000	1.5829	4.0000	-3.1136	1.5055	1.7691
1	1.0000	1.5829	1.5055	1.7691	4.0000	-3.1136	1.4903	1.7714
2	1.0000	1.5829	1.4903	1.7714	1.5055	1.7691	1.4256	1.7757
3	1.0000	1.5829	1.4256	1.7757	1.4903	1.7714	1.4266	1.7757
4	1.4256	1.7757	1.4266	1.7757	1.4903	1.7714	1.4275	1.7757
5	1.4266	1.7757	1.4275	1.7757	1.4903	1.7714	1.4276	1.7757

Implementación en Java

```
  void Interpolacion_Cuadratica()
  {
    double x0, x1, x2, x3;
    double f0, f1, f2, num, den;

    int i = 0;

    x0 = 0;
    x1 = 1;
    x2 = 4;

    System.out.println("i" + "\t" + "x0" + "\t" + "f0" + "\t" + "x1" +
"\t" + "f1" +
                              "\t" + "x2" + "\t" + "f2" + "\t" + "x3" +
"\t" + "f3");
    do
    {
      f0 = funcion(x0);
      f1 = funcion(x1);
      f2 = funcion(x2);

      den = f0*(x1*x1 - x2*x2) + f1*(x2*x2 - x0*x0) + f2*(x0*x0 - x1*x1);
      num = f0*(x1 - x2) + f1*(x2 - x0) + f2*(x0 - x1);
      x3 = den/(2.0*num);


      System.out.println(i++ + "\t" + x0 + "\t" + f0 + "\t" + x1 + "\t" +
f1 +
```

```
                                    "\t" + x2 + "\t" + f2 + "\t" + x3 + "\t" +
funcion(x3) );
      if(x0 <= x3 && x3 <= x1)
      {
        x2 = x1;
        x1 = x3;
      }
      else
      {
        x0 = x1;
        x1 = x3;
      }
    }while(x3-x0 >= 0.0001);
  }
```

Regresar.

Método de Newton

El objetivo de este método es calcular el máximo o mínimo de una función, haciendo uso de una aproximación cuadrática dada por la serie de Taylor. Dicha aproximación cuadrática es:

El máximo o mínimo de la función *q(x)* lo calculamos haciendo *q'(x) = 0.* Haciendo esto obtenemos

Note que la ecuación resultante es muy similar a la utilizada en el método de Newton Raphson si la función *f(x)* es sustituida por *f'(x)*.

Algoritmo

1.- Dado un valor inicial x_0 y una función cuyas primera y segunda derivada existan
2.- Calculamos la i-ésima aproximación utilizando la formula

3.- Repetimos el paso dos hasta convergencia.

Ejemplo

Use el método de Newton para encontrar el máximo de la función $f(x) = 2seno(x) - x^2/10$ con el valor inicial $x_0 = 2.5$.

i	x0	x1

1	2.5000	0.9951
2	0.9951	1.4690
3	1.4690	1.4276
4	1.4276	1.4276

Implementación en Java

```
void Newton()
{
  double x0 = 2.5, x;
  int i;

  System.out.println("i" + "\t" + "x0" + "\t" + "x1");
  for(i=1; i<1000; i++)
  {
    x = x0 - Df(x0)/DDf(x0);

    System.out.println(i + "\t" + x0 + "\t" + x);

    if(Math.abs(x-x0) <= 0.0001) break;
    else x0 = x;
  }
}

double Df(double x)
{
  return 2.0*Math.cos(x) - 2.0*x/10.0;
}

double DDf(double x)
{
  return -2.0*Math.sin(x) - 2.0/10.0;
}
```

Regresar.

Optimización Multidimensional sin restricciones

Búsqueda aleatoria

Este método evalúa en forma repetida la función mediante la selección aleatoria de valores de la variable independiente. Si un número suficiente de muestras se lleva a cabo, el óptimo será eventualmente localizado.

Dada una función $f(x,y)$ en el dominio acotado $x \in [x_l, x_u]$ y $y \in [y_l, y_u]$, se genera un número aleatorio r entre 0 y 1 y los valores de x y de y en el rango se calculan como:

$$x = x_l + (x_u - x_l) * r$$
$$y = y_l + (y_u - y_l) * r$$

Ejemplo

Encontrar el máximo de la función f(x,y) = y –x –$2x^2$ - 2xy - y^2 en el dominio acotado $x \in [-2, 2]$ y $y \in [1, 3]$.

k	x	y	f(x,y)
0	0.00000	0.00000	0.00000
1	-0.84276	1.19252	1.20270
2	-0.84276	1.19252	1.20270
3	-0.84276	1.19252	1.20270
4	-0.84276	1.19252	1.20270
5	-0.84276	1.19252	1.20270
6	-0.84276	1.19252	1.20270
7	-0.84276	1.19252	1.20270
8	-0.84276	1.19252	1.20270
9	-0.84276	1.19252	1.20270
167	-0.92622	1.28268	1.22395
183	-1.11212	1.49897	1.22463
465	-1.04210	1.66243	1.23375
524	-0.96660	1.36514	1.23859
626	-0.97539	1.48462	1.24931
999	-0.97539	1.48462	1.24931

Implementación en Java

```
void Busqueda_Aleatoria()
{
  int i;
  double r, x, y, max = 0, maxx = 0, maxy = 0, fx;
  double xl = -2, xu = 2, yl = 1, yu = 3;
  Random azar = new Random();


  for(i=0; i<1000; i++)
  {

    r = azar.nextDouble();
    x = xl + (xu - xl)*r;
    r = azar.nextDouble();
    y = yl + (yu - yl)*r;

    fx = f(x,y);

    if(fx > max)
    {
      max = fx;
      maxx = x;
      maxy = y;
    }
    System.out.println(i + "\t " +maxx + "\t " + maxy + "\t " + max );
  }
}
```

```
double f(double x, double y)
{
  return y - x - 2*x*x - 2*x*y - y*y;
}
```

Métodos basados en gradiente

Como su nombre lo indica, son métodos para calcular el óptimo de una función, basados en la información del gradiente.

El gradiente.

Dada una función en dos dimensiones *f(x,y)*, las derivada direccional, se define como

donde θ es la dirección sobre la cual se quiere calcular la derivada y u es un vector unitario con la información de la dirección de θ. Podemos escribir la derivada direccional como

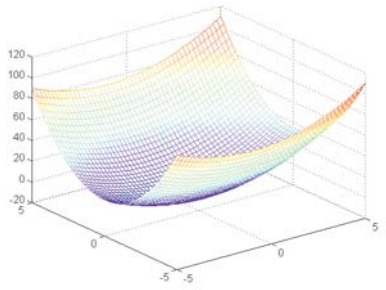
El gradiente se define como el vector de derivadas en la direcciones de x y y donde el cambio es máximo. La expresión de este esta dado por

donde i y j representan los vectores unitarios en la direcciones de x y y.

Utilizando la notación vectorial podemos representar al gradiente como

Ejemplo 1

Dada la función $f(x,y) = x^2 + 2x + 3y^2$, determinar el mínimo de la función y la dirección hacia donde se encuentra el mínimo en el punto de coordenadas [2,2].

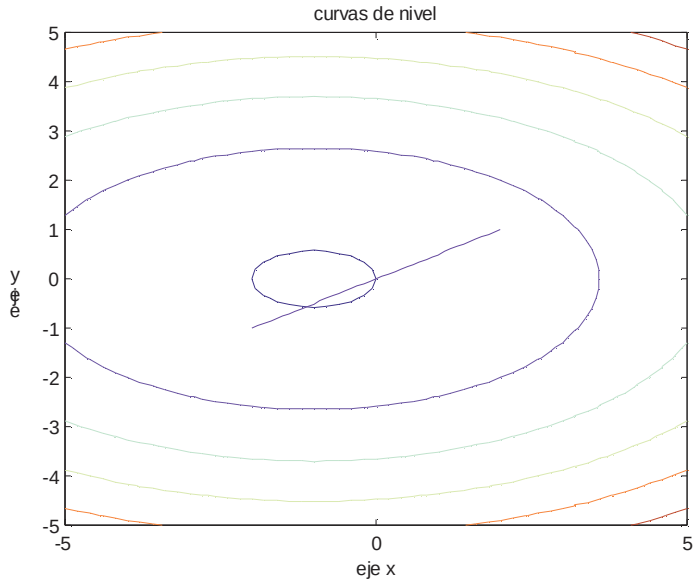


Si calculamos la derivada tenemos e igualamos a cero

$$f'_x = 2x+2 = 0$$
$$f'_y = 6y = 0$$

Note que el mínimo se localiza en el punto [0,0].

Ahora, si valuamos el gradiente en el punto [2,2] tenemos que es igual a [6, 12], de manera gráfica tenemos.



Note como del punto [2,2] puedo acercarme al mínimo, siguiendo la dirección opuesta al gradiente.

Demostración.

Dada la definición de la derivada direccional, podemos ver que esta, se puede escribir como el producto escalar de vectores

Si queremos determinar la dirección en la cual la función decrece, calculamos la derivada de *F*, con respecto a phi.

Note que el mínimo se encuentra cuando el ángulo es de 180 grado, lo cual significa que la función decrece en la dirección opuesta al gradiente y el máximo se encuentra en la dirección del gradiente.

Método de descenso de gradiente

Como hemos visto, el máximo de una función se encuentra en la dirección del gradiente, mientras que un mínimo en la dirección opuesta. Así podemos plantear una estrategia para alcanzar el mínimo de una función utilizando un punto inicial, el valor del gradiente y un tamaño de paso α con el cual avanzaremos.

Algoritmo

1.- Dado una posición inicial $r^{(0)}$ en un espacio n dimensional y una función *f(r)*

2.- Calculamos el valor del gradiente en

3.- Hacemos la actualización utilizando la sucesión

4.- Si la magnitud del gradiente es cercana a cero, terminamos, sino hacemos $k = k+1$ y regresamos a 2.

Implementación en Java

```
static public void Descenso_Gradiente()
{
  double r[] = {2,2};
  double u[] = new double[2];
  double alfa = 0.01;
```

```
  int k = 0;

  while(gradiente(u, r) > 0.01)
  {
    System.out.println(k + " " + r[0] + " " + r[1] + " " + f(r));
    gradiente(u, r);
    r[0] -= alfa*u[0];
    r[1] -= alfa*u[1];
    k ++;
  }
}

public static double f(double r[])
{
  return r[0]*r[0] +2.0*r[0] + 3.0*r[1]*r[1];
}

public static double gradiente(double u[], double r[])
{
  double mag;

  u[0] = 2.0*r[0] + 2.0;
  u[1] = 6.0*r[1];

  mag = Math.sqrt(u[0]*u[0] + u[1]*u[1]);

  u[0] /= mag;
  u[1] /= mag;

  return mag;
}
```

Problema para seleccionar el tamaño de paso

La solución del ejemplo 1 utilizando un tamaño de paso de 0.01 es:

k	x	y	f(x,y)
0	2.0000	2.0000	20.0000
1	1.9955	1.9911	19.8661
2	1.9910	1.9821	19.7327
3	1.9866	1.9732	19.5998
4	1.9820	1.9643	19.4675
5	1.9775	1.9553	19.3357
6	1.9730	1.9464	19.2044
7	1.9685	1.9375	19.0736
8	1.9639	1.9286	18.9433
9	1.9594	1.9197	18.8135
387	-0.9879	0.0000	-0.9999

Podríamos suponer que un tamaño de paso mas grande hará que el algoritmo converja mas rápido, así que para un tamaño de paso de 0.5 tenemos

k	x	y	f(x,y)
0	2.0000	2.0000	20.0000
1	1.7764	1.5528	13.9418
2	1.5204	1.1233	9.1378
3	1.2209	0.7229	5.5003
4	0.8632	0.3736	2.8902
5	0.4347	0.1159	1.0987
6	-0.0512	-0.0019	-0.0998
7	-0.5512	0.0011	-0.7986
8	-1.0512	-0.0025	-0.9974
9	-0.5566	0.0710	-0.7883
4680	-1.0000	-0.1456	-0.9364

Note que después de 4680 iteraciones no hemos convergido y que pasa si utilizamos un tamaño de paso de 0.005

k	x	y	f(x,y)
0	2.0000	2.0000	20.0000
1	1.9978	1.9955	19.9330
2	1.9955	1.9911	19.8661
3	1.9933	1.9866	19.7993
4	1.9910	1.9821	19.7327
5	1.9888	1.9777	19.6662
6	1.9865	1.9732	19.5998
7	1.9843	1.9687	19.5336
8	1.9820	1.9643	19.4675
9	1.9798	1.9598	19.4015
775	-0.9936	0.0000	-1.0000

Note que la convergencia es más lenta, pero la precisión obtenida es mejor. La selección del tamaño del paso es un proceso complicado donde este valor quedará sujeto a las características del problema y no existe regla general para seleccionarlo.

Hessiano

Cuando calculamos el gradiente de una función e igualamos este a cero, estamos calculando un punto que puede ser máximo o mínimo. Una manera de saber si estamos en un máximo o un mínimo es hacer una evaluación sobre la segunda derivada. Así estaremos en un mínimo si la segunda derivada es positiva y en un máximo si la segunda derivada es negativa.

En el caso de funciones de varias variables, dado que se tiene una función que depende de varias variables, es común representarla en una matriz de segundas derivadas a las que se les denomina Hessiano. El Hessiano de una matriz de dos variables se define como:

En este caso tendremos un mínimo local si el determinante de H es positivo o un máximo local si el determinante es negativo.

Ejemplo 2

Determinar para el ejemplo 1, si se trata de un mínimo o un máximo.

Calculamos la matriz de segundas derivadas y las evaluamos en cero

Note que el determinante es positivo, lo cual indica que en la posición [0,0] se tiene un mínimo.

El método de Newton

El método de Newton esta basado en la minimización de una función cuadrática. Esta función cuadrática es calculada a partir de la serie de Taylor

donde .

Si calculamos el máximo de q(X) tenemos

Despejando tenemos, que el mínimo de la aproximación cuadrática lo obtenemos resolviendo el siguiente sistema lineal de ecuaciones.

Algoritmo

1. Dado una función y un punto inicial $X^{(0)}$
2. Determinamos el Gradiente de la función $g(X^{(k)})$

3. Calculamos el valor del Hessiano de la función $H(X^{(k)})$
4. Resolvemos el sistema de ecuaciones $H(X^{(k)})dx = g(X^{(k)})$
5. Actualizamos el valor de $X^{(k+1)} = X^{(k)} - dx$
6. Si la magnitud del gradiente es pequeña terminamos, si no hacemos $k = k+1$ y regresamos a 2.

Ejemplo

Dada la función $f(x,y) = x^2 + 2x + 3y^2$, determinar el mínimo de la función dado el punto inicial $X^{(0)} = [2,2]$.

El gradiente de la función es

El Hessiano es

El sistema de ecuaciones que debemos resolver es

La solución del sistema es $dx = 3$ y $dy = 2$, con lo cual, los nuevos valores son $x=1$ y $y = 0$. Note que solo fue necesaria una iteración, esto se debe a que la aproximación cuadrática es exacta.

Ejemplo

Dada la función $f(x,y) = 100(x^2-y)^2 + (1-x)^2$ determinar el mínimo dado como valor inicial $X^{(0)} = [10, 10]$

El gradiente de la función es

El Hessiano es

$400(x^2 - y)*x - 2(1 - x)$
$-200\ (x^2 - y)$

El sistema de ecuaciones que debemos resolver es

$1200x^2 - 400y + 2.0$	-400x
-400x	200

Primera iteración

En nuestra primera iteración resolvemos el sistema de ecuaciones

116002.0	-4000.0	dx	=	360018.0
-4000.0	200	dy		-18000.0

Las actualizaciones son $x = 9.999500027776234$ y $y = 99.99000055552469$

Segunda iteración

Resolvemos el sistema de ecuaciones

79994.0007	-3999.8000	dx	=	17.9999
-3999.8000	200	dy		-4.9994E-5

Las actualizaciones son $x = 1.0004$ y $y = -79.9820$

Tercera iteración

Resolvemos el sistema de ecuaciones

33195.8812	-400.1799	dx	=	32407.7359
-400.1799	200	dy		-16196.5806

Las actualizaciones son $x = 1.0004$ y $y = 1.0008$

En resumen tenemos

k	x	y
1	9.99950	99.99000
2	1.00045	-79.98200
3	1.00045	1.00090
4	1.00000	1.00000
5	1.00000	1.00000
6	1.00000	1.00000
7	1.00000	1.00000
8	1.00000	1.00000
9	1.00000	1.00000
10	1.00000	1.00000

La implementación en Java de este ejemplo es:

```
public class ej033 {
  public static void main(String[] args) {
    Newton();
  }

  public static void Newton()
  {
    int n = 2;
    double x[]   = {10, 10};
    double g[]   = new double[n];
    double dx[]  = new double[n];
    double H[][] = new double[n][n];
    int i, k;

    for(k=1; k<20; k++)
    {
      gradiente(g, x);
      Hessiano(H, x);
      slineal.SolucionLU(H, dx, g);

      System.out.print(k + "\t");

      for(i=0; i<n; i++)
      {
        x[i] -= dx[i];
        System.out.print(x[i] + "\t");
      }

      System.out.println();
    }
  }

  public static void gradiente(double g[], double x[])
  {
    g[0] =  400.0*(x[0]*x[0] - x[1])*x[0] - 2.0*(1 - x[0]);
    g[1] = -200.0*(x[0]*x[0] - x[1]);
  }

  public static void Hessiano(double H[][], double x[])
  {
    H[0][0] =  1200.0*x[0]*x[0] - 400.0*x[1] + 2.0;
    H[0][1] = -400.0*x[0];
    H[1][0] = -400.0*x[0];
    H[1][1] =  200.0;
  }
}
```

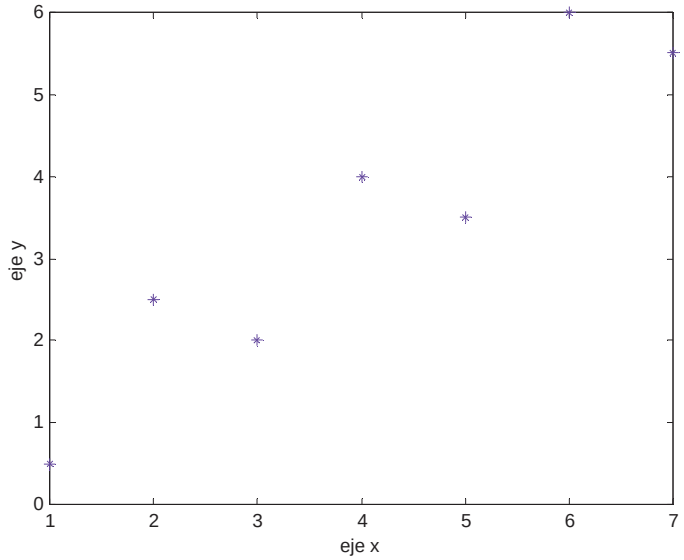
Regresar.

Regresión Lineal

El ejemplo más simple de una aproximación por mínimos cuadrados es mediante el ajuste de un conjunto de pares de observaciones: $(x_1, y_1), (x_2, y_2), (x_3, y_3),..., (x_n, y_n)$ a una línea recta. La expresión matemática de la línea recta es:

$$y = a_0 + a_1 x + e$$

donde a_0 y a_1, son los coeficientes que representan el cruce con el eje y y la pendiente de la línea y e representa el error de nuestra aproximación.



Una estrategia, para ajustar a la mejor línea, es minimizar la suma al cuadrado de los errores para todos los datos disponibles

Esta misma ecuación la podemos escribir en forma matricial como

El vector de error en forma matricial queda como

y en general

En forma compacta tenemos que $e = y - Ma$, donde M es la matriz de coordenadas en x y a el vector de parámetros.

Ajuste por mínimos cuadrados

Si queremos encontrar el vector de parámetros a que minimiza nuestra suma de cuadrados, tenemos que calcular la derivada de la función de error respecto al vector de parámetros e igualar a cero.

El valor del vector de parámetros a los calculamos resolviendo el siguiente sistema de ecuaciones

$$[M^T M]a = M^T y$$

Es común encontrar la solución de este sistema como:

Ejemplo

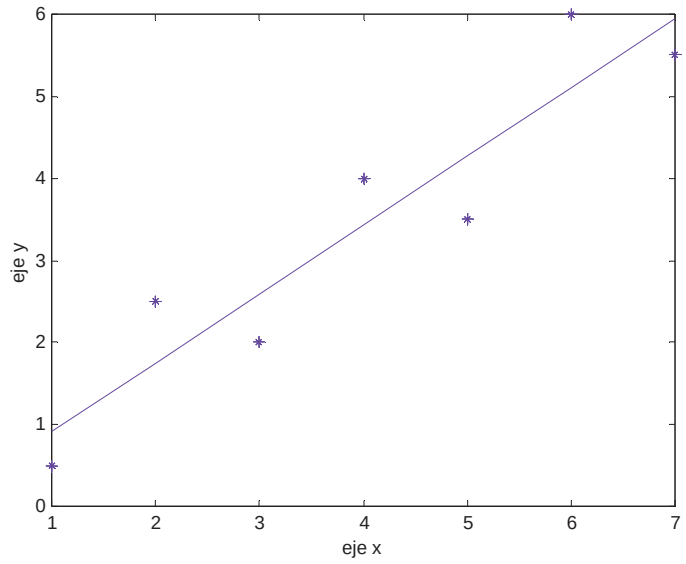
Hacer el ajuste a una línea recta de los siguientes valores

x	y
1.00	0.50
2.00	2.50
3.00	2.00
4.00	4.00
5.00	3.50
6.00	6.00
7.00	5.50

Aplicando las formulas anteriores, tenemos que el sistema de ecuaciones a resolver es

7	28.00	a0	24.00
28	784.00	a1	119.50

La solución es a = [0.07142, 0.8392] y en la siguiente figura se muestra el ajuste encontrado



Regresión polinomial

Podemos generalizar el caso de la regresión lineal y extenderla a cualquier polinomio de orden m.

$$[M^T M]a = M^T y$$

Así por ejemplo la solución para un polinomio de orden 3 es:

Note que cada uno de los elementos de la matriz M^TM son una sumatoria de todos los valores de x elevado a un exponente que resulta ser la suma del renglón y la columna donde se localiza. Adicionalmente la matriz es simétrica.

El vector de términos independientes es

Ejemplo

Ajustar a un polinomio de segundo orden los datos en la siguiente tabla.

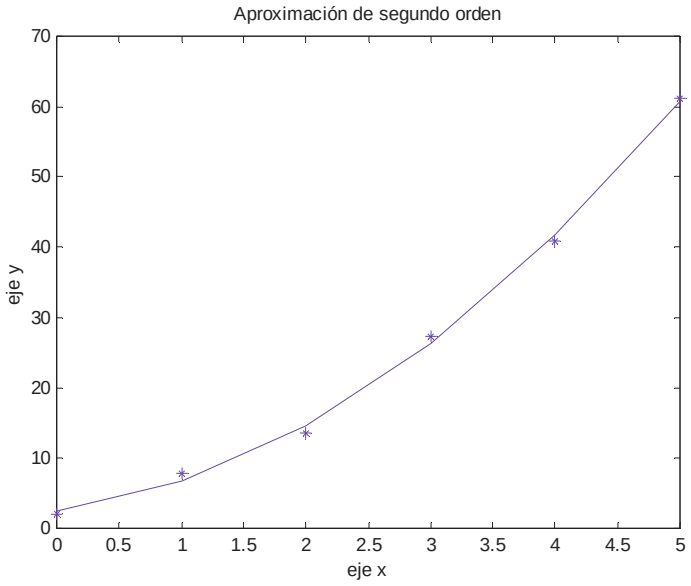
x	y
0.00	2.10
1.00	7.70
2.00	13.60
3.00	27.20

4.00	40.90
5.00	61.10

El sistema de ecuaciones que debemos resolver es:

6.00	15.00	55.00	a[0]		152.60
15.00	55.00	225.00	a[1]	=	585.60
55.00	225.00	979.00	a[2]		2488.80

El ajuste da como resultado el polinomio $p(x) = 2.4785 + 2.3592x + 1.8607x^2$. En la siguiente figura se muestra la aproximación a un polinomio de segundo orden.



Implementación en Java

```
public class ej034 {
  public static void main(String[] args) {
    double x[] = {0,     1,  2,    3,    4,    5};
    double y[] = {2.1, 7.7, 13.6, 27.2, 40.9, 61.1};
    double a[] = Regresion(x, y, 2);
    for(int i=0; i<a.length; i++)
      System.out.println("a["+i+"] = "+ a[i]);
    grafica(x, y, a);
  }

  public static double [] Regresion(double x[], double y[], int orden)
  {
    int i, j, n = orden +1;
    double a[]    = new double[n];
    double MM[][] = new double [n][n];
    double My[]   = new double[n];
```

```
  for(i=0; i<n; i++)
  {
    for (j = 0; j <= i; j++) {
      MM[i][j] = sumas(x, i + j);
      MM[j][i] = MM[i][j];
    }
    My[i] = sumas(x, y, i);
  }
  imprime(MM, My);
  slineal.Solucion(MM, a, My);
  return a;
}

static public void grafica(double x[], double y[], double a[])
{
  int i, j, n = x.length, m = a.length;
  double yc[] = new double[n];

  for(i=0; i<n; i++)
  {
    yc[i] =0;
    for(j=0; j<m; j++)
      yc[i] += a[j] * eleva(x[i], j);

  }

  grafica g = new grafica("salida");
  g.Datos(x, y);
  g.Datos(x, yc);
  g.show();
}

static public void imprime(double A[][], double b[])
{
  int n = b.length;
  int i, j;

  for(i=0; i<n; i++)
  {
    for (j = 0; j < n; j++)
      System.out.print(A[i][j] + "\t");

    System.out.println("\t" + "a["+i+"] \t" + b[i]);
  }
}

public static double sumas(double x[], int exp)
{
  double suma = 0;
  int n = x.length;
  for(int k=0; k<n; k++)
    suma += eleva(x[k], exp);

  return suma;
}

public static double sumas(double x[], double y[], int exp)
```

```
  {
    double suma = 0;
    int n = x.length;
    for(int k=0; k<n; k++)
      suma += eleva(x[k], exp)*y[k];

    return suma;
  }

  public static double eleva(double x, int exp)
  {
    double resul = 1;

    for(int i=1; i<=exp; i++)
      resul *= x;

    return resul;
  }
}
```

Regresar.