Capítulo XXV: Cálculo Numérico[1]

Motivación

El computador representa los números con una cantidad finita de cifras (no tendríamos memoria si pusiéramos infinitas cifras).

En los enteros, el rago de trabajo va desde –2.000.000.000 y 2.000.000.000. En los reales (punto flotante) los números no solo tienen un rango definido, sino que además tienen una precisión finita también, por lo tanto, los número como 1/3 que no pueden ser representados con una base decimal finita, poseen un cierto "margen de error" en el computador.

¿Qué importancia tiene para el ingeniero?

Esta situación sirve como un antecedente que el ingeniero debe tener muy en cuenta al momento de realizar cálculos en el computador, ya que debe considerar que todos ellos poseen errores de aproximación. Por ejemplo:

```
double sexto = 1.0/6.0;
double uno = sexto + sexto + sexto + sexto + sexto + sexto;
if (uno == 1.0)
      System.out.println("Es 1.0");
```

Te darás cuenta que no lo hace, porque la variable uno posee un cierto error de aproximación. Para corregir esto deberíamos hacer:

```
final double tolerancia = 0.000001
double sexto = 1.0/6.0;
double uno = sexto + sexto + sexto + sexto + sexto + sexto;
if (Math.abs(uno - 1.0) < tolerancia)
      System.out.println("Es 1.0");
```

Con estas aproximaciones, el computador sirve para resolver problemas de métodos numéricos.

Definición

Existen muchos problemas de cálculo que el computador es capaz de resolver rápidamente. Estos problemas son considerados dentro de un área que se llama Cálculo Numérico y sale de la matemática normal porque utilizan aproximaciones para resolver los problemas más clásicos.

Veremos algunos problemas y los métodos (que muchos matemáticos han desarrollado y probado) que sirven para resolver a través de aproximaciones más y menos exactas un problema matemático.

[1] Gentileza del profesor Kurt Schwarze

CC10A – Introducción a la Computación – Capítulo XXV
Sección 06 – 2003

Raíz de una Función

La raíz de una función es el punto en donde esa función corta el eje X en el gráfico. Si uno sabe que la función en un x1 es positiva y en un x2 es negativa, podemos encontrar numéricamente la raíz mediante búsqueda binaria.

El método consiste en evaluar la función en la mitad del intervalo y comenzar a tomar ese valor junto con uno de los dos bordes del intervalo el cual es de distinto signo que la mitad. Esto quiere decir que si x (la mitad entre x1 y x2) es positivo, se toma como nuevo intervalo de búsqueda x y x2, de lo contrario tomar x1 y x, y seguir con la búsqueda.

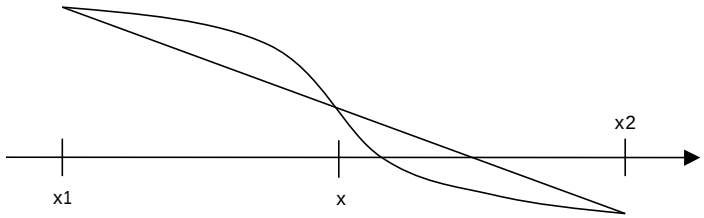
¿Hasta cuando?

Se define una cierta tolerancia o vecindad que se toma como aceptable entre x y x2 o x1 y x.

```
public class funcion {
      public double eval(double x) {
            ... // evaluación de la función
      }

      public double raiz (double x1, double x2, double epsilon) {
            // Si cumple con la vecindad
            if (Math.abs(x2 – x1) < espsilon)
                  return x1;

            // Sacamos el punto medio
            double x = (x1 + x2) / 2;
            boolean esNegativo = (this.eval(x1) < 0);

            // Vemos el siguiente intervalo
            if (esNegativo && this.eval(x) > 0)
                  return this.raiz(x1, x, epsilon);
            else if (!esNegativo && this.eval(x) < 0)
                  return this.raiz(x1, x, epsilon);
            else
                  return this.raiz(x, x2, epsilon);
      }
}
```

Esta versión recursiva de la búsqueda binaria nos muestra como numéricamente encontraríamos la raíz de una función.

Otro método más refinado es el método de la secante. En este caso, en vez de promediar x1 y x2 para sacar la mitad, se "pesan" los valores de f(x1) y f(x2) para iterar:



Otros métodos ya más elaborado fueron desarrollados por grandes matemáticos. Uno de ellos es el conocido Newton-Raphson, el cual se basa en calcular la sucesión:

$$x_{n+1} \leftarrow x_n - \frac{f(x_n)}{f'(x_n)} .$$

donde x_{n+1} es la aproximación que se obtiene para la raíz en la iteración n+1. Esto si lo llevamos a nuestra evaluación de la raíz quedaría como:

```
...
      public double eval(double x) {
            // evalua la función en x
      }

      public double deriv(double x) {
            // evalua la derivada de la función en x
      }

      public double raiz (double xn, double epsilon) {
            double x = xn - (eval(xn) / deriv(xn));

            // Verificamos la vecindad
            if (Math.abs(x - xn) < epsilon)
                  return x;
            else
                  return raiz(x, epsilon);
      }
...
```

Existe un caso particular llamada fórmula de Herón, y que sirve para calcular la raíz cuadrada de un número x. La sucesión es:

$$r \leftarrow \frac{r + x/r}{2} .$$

La solución iterativa (no recursiva) de esta sucesión es:

```
...
      public double raiz (double x, double epsilon) {
            double r = 1.0;

            while (true) {
                  r = (r + x/r) / 2.0;
                  if (Math.abs(r - x/r) < epsilon)
                        break;
            }
            return r;
      }
...
```


Aún cuando estas fórmulas sean "matemáticas", podría no haber convergencia, es decir, nunca encontrar una vecindad tan pequeña. ¿Qué se podría agregar a Newton-Raphson para que el computador no se cuelgue?.

Evaluación de Polinomios

Al hacer cálculos numéricos hay que fijarse en obtener una buena precisión con la menor cantidad de cálculos posibles. Hay veces que los métodos para el mismo problema pueden tomar tiempos distintos. Por ejemplo:

$$y(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

se puede simplificar haciendo:

$$y(x) = a_3 x \cdot x \cdot x + a_2 x \cdot x + a_1 x + a_0$$

Generalizando para el polinomio de orden n, se obtiene una cantidad de multiplicaciones proporcional a n^2 y una cantidad de sumas igual a n, en cambio, si el polinomio se evalúa astutamente usando la Regla de Horner, se hacen sólo n multiplicaciones y n sumas:

$$y(x) = ((a_3 x + a_2) x + a_1) x + a_0$$

En Java, la versión iterativa para el caso general quedaría:

```
public double horner (double[] a, int n, double x) {
      // a son los coeficientes y n el orden
      double suma = a[n];
      for (int i = n-1; i >= 0; i--)
            suma = suma * x + a[i];
      return suma;
}
```

También, podemos hacer un pequeño análisis más grande y decir que:

$$y(x) = (y_1(x)) x + a_0$$

Así tenemos que y_1 es un polinomio más pequeño (recursivo), para dejar el programa en Java, astutamente llamándolo como horner(a, 0, a.length, x):

```
public double horner (double[] a, int i, int n, double x) {
      if (n == 0)
            return a[i];
      return horner(a, i+1, n-1, x) * x + a[i];
}
```

Aproximación de Series Infinitas

Muchas funciones matemáticas pueden ser representadas por series infinitas de términos. Por ejemplo:

$$e^n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Y las fórmulas del seno y coseno, en donde x está expresado en radianes. En Java, veamos como quedaría el seno de x:

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

utilizando solo 7 términos de la serie original y forma iterativa quedaría:

```
public double seno ( double x ) {
      double xcuad = x * x;
      double term = x;
      double sine = x;
      for (int i=1; i<8; i++) {
            term = - (term * xcuad ) / (2 * i * ( 2 * i + 1 ));
            sine = sine – (term * xcuad) / (2 * i * ( 2 * i + 1 ));
      }
      return sine;
}
```

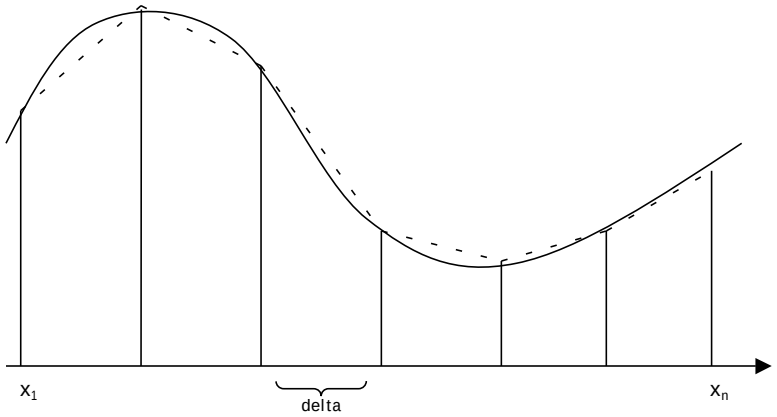
Y si optimizamos un poco para seleccionar dinámicamente la cantidad de términos (precisión) del seno:

```
public double seno ( double x, double n ) {
      double xcuad = x * x;
      double term = x;
      double sine = x;
      for (int i=1; i<n+1; i++) {
            term = - (term * xcuad ) / (2 * i * ( 2 * i + 1 ));
            sine = sine – (term * xcuad) / (2 * i * ( 2 * i + 1 ));
      }
      return sine;
}
```

Propuesto: Saca la forma general de la serie y resuélvela en forma recursiva, es decir, escribe el método seno(x, n) que sea recursivo y que haga lo mismo que lo anterior en forma muy astuta.

Integración Numérica

Existe un método para calcular la integral y es el Método de Los Trapecios. Supón que se tiene una curva y = f(x), y que se quiere encontrar el área entre la curva y el eje de las absisas entre las coordenadas $x = x_1$ y $x = x_n$.



Entonces, se divide el intervalo $[x_1, x_n]$ en intervalos de tamaño delta, tal que

$$delta = (x_n – x_1) / (n – 1)$$

Como muestra la figura, se puede aproximar el área bajo la curva dibujando los trapecios imaginariamente y sumándo sus propias áreas como:

$$area = (y_1 + y_2) * delta / 2 + (y_2 + y_3) * delta / 2 + \ldots + (y_{(n-1)} + y_n) * delta / 2$$

factorizando:

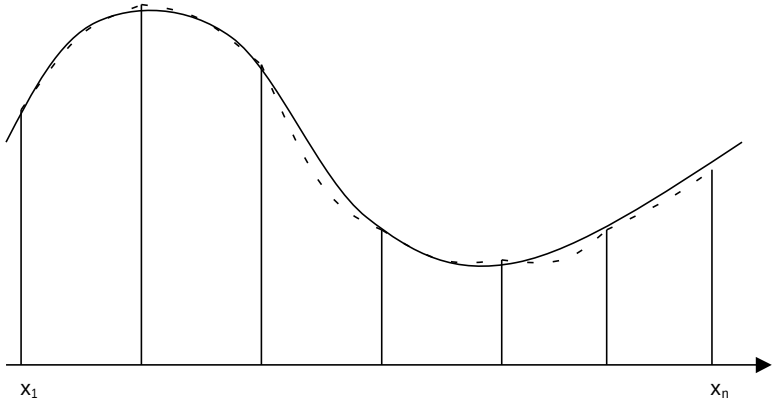
$$area = (y_1 / 2 + y_2 + y_3 + \ldots + y_{(n-1)} + y_n / 2) * delta$$
$$area = ((y_1 + y_n) / 2 + y_2 + y_3 + \ldots + y_{(n-1)}) * delta$$

Con esto, el programa en Java es tan sencillo como:

```
public double integral ( double y[], int n ) {
      double suma = (y[0] + y[n-1]) / 2;
      for (int i = 1; i < n-1; i++)
            suma += y[i];
      return suma * delta;
}
```

Entre más grande es el n (cantidad de intervalos) más preciso va a ser la aproximación del área y más real saldrá la integral.

Otro método para calcular el área de la curva es el Método de Simpson (y no es de Bart Simpson), en donde asume que cada par de tajadas adyacentes de la curva tiene la forma de una parábola.



Visualmente parece ser más exacto que el método de los trapecios, pero todo depende del tamaño delta que se elija.

El área de cada par de tajadas es delta por 1/3 de la suma de los valores de la función en los extremos más cuatro veces el valor del medio, es decir:

$$(f(x_0) + 4\ f(x_1) + f(x_2)) * delta / 3$$

Todo esto evaluado para 2 tajadas. Pero si tomásemos 4 tajadas quedaría:

$$(f(x_0) + 4\ f(x_1) + 2\ f(x_2) + 4\ f(x_3) + f(x_4)) * delta / 3$$

Esto se puede ir extendiendo de a pares, es decir, se van tomando pares de tajadas y se va calculando el área. Por ejemplo:

```
public double integral ( double y[], int n) {
      double suma = 0;
      for (int i = 1; i < n-1; i++)
            if ( i % 2 == 0 ) // par
            suma += 4 * y[i];
      else
            suma += 2 * y[i];
      return (y[0] + suma + y[n-1]) * delta / 3;
}
```

En este ejemplo genérico, cada término par es multiplicado por 4 y cada término impar multiplicado por 2.

Veamos un ejemplo de la diferencia de los métodos para el caso del sen entre 0 y pi:

```
public class Seno {
      private double[] evalua_intervalo ( int n ) {
            double y[] = new double[n+1];
            double delta = Math.PI / 6;
            for (int i=0; i<n+1; i++)
                  y[i] = Math.sin(i * delta);
            return y;
      }

      public double integral_trapecios ( double n ) {
            double y[] = evalua_intervalo(n);
            double suma = (y[0] + y[n-1]) / 2;
            for (int i = 1; i < n-1; i++)
                  suma += y[i];
            return suma * delta;
      }

      public double integral_simpson ( int n ) {
            double y[] = evalua_intervalo(n);
            double suma = 0;
            for (int i = 1; i < n-1; i++)
                  if ( i % 2 == 0 ) // par
                  suma += 4 * y[i];
            else
                  suma += 2 * y[i];
            return (y[0] + suma + y[n-1]) * delta / 3;
      }
}
```

Si utilizamos el siguiente programa:

```
public class Compara {
      static public void main(String args[]) {
            Seno s = new Seno();
            System.out.println( "Trapecios = " +
                  s.integral_trapecios(6) );
            System.out.println("Simpson = " +
                  s.integral_simpson(6) );
      }
}
```

La salida estántas mostrará:

```
Trapecios = 1.954097
Simpson = 2.000863
```

Es decir, en este caso el error con el método de Simpson es 53 veces menos que con el de los Trapecios.

Sistemas de Ecuaciones Lineales

También es sencillo resolver sistemas de ecuaciones lineales del estilo Ax = b a través del computador. Uno puede guardar un sistema de ecuaciones en 3 arreglos: una matriz con los coeficientes A, un vector con la incógnita x y otra matriz con los coeficientes del vector solución b.

Por ejemplo, un sistema general de ecuaciones lineales quedaría como:

$$
\begin{aligned}
A_{1,1}\,x_1 + A_{1,2}\,x_2 + A_{1,3}\,x_3 + \ldots + A_{1,n}\,x_n &= b_1 \\
A_{2,1}\,x_1 + A_{2,2}\,x_2 + A_{2,3}\,x_3 + \ldots + A_{2,n}\,x_n &= b_2 \\
A_{3,1}\,x_1 + A_{3,2}\,x_2 + A_{3,3}\,x_3 + \ldots + A_{3,n}\,x_n &= b_3 \\
&\ldots \\
A_{n,1}\,x_1 + A_{n,2}\,x_2 + A_{n,3}\,x_3 + \ldots + A_{n,n}\,x_n &= b_n
\end{aligned}
$$

Basta que la matriz A sea convertida en una triangular superior:

$$
\begin{pmatrix}
A'_{1,1} & A'_{1,2} & A'_{1,3} & \ldots & A'_{1,n} \\
0 & A'_{2,2} & A'_{2,3} & \ldots & A'_{2,n} \\
0 & 0 & A'_{3,3} & \ldots & A'_{3,n} \\
 & & & \ldots & \\
0 & 0 & 0 & \ldots & A'_{n,n}
\end{pmatrix}
$$

Con esta nueva matriz, el sistema se reduce al problema de resolver:

$$
\begin{aligned}
A'_{1,1}\,x_1 + A'_{1,2}\,x_2 + A'_{1,3}\,x_3 + \ldots + A'_{1,n}\,x_n &= b_1 \\
A'_{2,2}\,x_2 + A'_{2,3}\,x_3 + \ldots + A'_{2,n}\,x_n &= b_2 \\
A'_{3,3}\,x_3 + \ldots + A'_{3,n}\,x_n &= b_3 \\
\ldots & \\
A'_{n,n}\,x_n &= b_n
\end{aligned}
$$

Con la última incógnitica x_n ya resuelta. Luego esta incóginita se va reemplazando en las distintas ecuaciones para luego terminar con la primera para sacar x_1.

Propuesto: Escriba un programa que permita resolver este tipo de ecuaciones pasando por los siguientes pasos:

(a) Escriba un método que dada una matriz cuadrada, retorne la matriz triangular superior.
(b) Escriba un método que reciba A y b, y que luego de triangular A retorne el vector x de resultados.