Informe Boletín 1 C++

M2i UDC (Grupo 1)
José García Quelle
Martín Zapata Martínez
Pedro Siaba Rodríguez
Adrián Sanjurjo García
Mónica Loureiro Varela

Febrero 2021

Índice

1. Instrucciones de Compilación

En cada uno de los archivos que contienen el main de cada ejercicio se incluye al principio en una línea comentada el código de compilación. Para mayor facilidad se incluye un fichero makefile con la línea de compilación para cada ejercicio. De esta manera escribiendo en la terminal situada en el directorio donde estan todos los archivos se escribe **make ej3**, o **make ej4** o el que se quiera. Se generará un archivo **b1_3.exe** o **b1_4.exe** por ejemplo.

2. Ejercicio 1

Dada una serie $\sum_{k=0}^{\infty} a_k$ se llama suma parcial S_n a la suma de los n primeros términos de dicha serie, $S_n = \sum_{k=0}^{n} a_k$. Para la serie

$$S_n = \sum_{k=0}^{\infty} \frac{3^k}{3 + k!}, \tag{1}$$

implementa un código que pida por teclado el número de términos ,n, que queremos sumar, y que muestre por pantalla el valor de suma parcial S_n.

Para la elaboración de este problema, se plantea un programa de nombre **b1_1.cpp** que contiene la estructura principal del cálculo. Se utiliza además otro programa auxiliar llamado **checkers.cpp** junto con un header **checkers.h**, los cuales comentaremos y se utilizarán además en otros códigos. Se pide por pantalla la cantidad de términos de la suma parcial n, y a continuación se llama a la función *check* que se encuentra en el programa **checkers.cpp**. Vemos que en dicho fichero contiene dos funciones de mismo nombre: La primera se ejecuta si lo que se pasa por pantalla es un entero, mientras que la segunda lo hace en el caso de que se pase un número real, pero en este apartado se trata de introducir un número entero. Ambas funciones *check* tratan de comprobar que lo que se está introduciendo por pantalla es un número del tipo especificado. En caso de no serlo, la función nos pide por pantalla que probemos otra vez a introducir un número, y en caso de que otra vez lo que se introduzca no es un número, repite esta pregunta por pantalla mf veces, siendo mf un valor introducido como parámetro de entrada en dichas funciones.

Ahora bien, una vez introducido el término correctamente, se puede calcular cualquier término de la suma parcial. Para comprobar los resultados de dicho programa, mostramos en la siguiente tabla varios cálculos realizados.

n	1	2	5	10	50	100
S_n	0.2500	1.0000	8.8000	12.4345	12.4566	12.4566

Cuadro 1: Cálculos de la suma parcial de la serie 1 mostrando 4 dígitos decimales

Como podemos comprobar, la suma converge a un valor aproximado menor que 12.5. Esto se podía esperar, puesto que se puede comprobar la convergencia de dicha serie bajo el criterio de d'Alembert para series:

$$\lim_{k\to\infty} \frac{a_{k+1}}{a_k} = \lim_{k\to\infty} \frac{\frac{3^{k+1}}{3+(k+1)!}}{\frac{3^k}{3+k!}} \simeq \lim_{k\to\infty} \frac{3}{k+1} = 0 \tag{2}$$

Sin embargo, para términos introducidos de n grande, cercano a 1000, veremos que la suma obtiene de resultado nan, puesto que el progama estará calculando valores como 1000!, lo que resulta imposible de calcular para la precisión de la máquina.

3. Ejercicio 2

Escribir un programa que aproxime el valor del número π generando puntos al azar. Consideremos un cículo inscrito en un cuadrado de lado $2L$. La relación general entre ambas áreas es

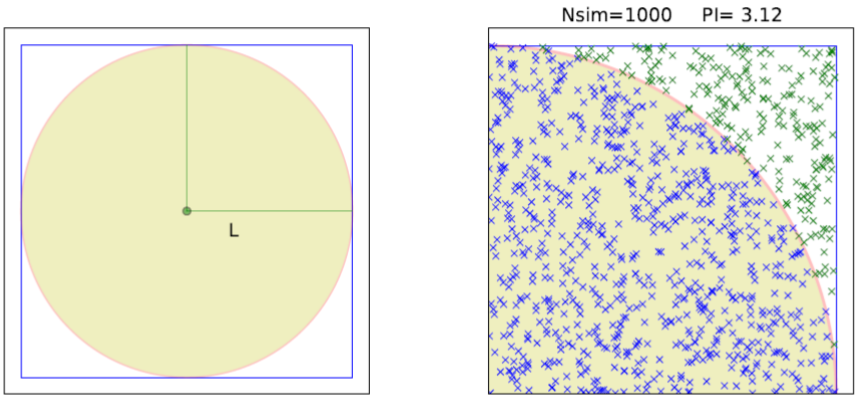

Figura 1: Relación entre el área del círculo y el cuadrado.

$$\frac{\text{Área Círculo}}{\text{Área Cuadrado}} = \frac{\pi L^2}{4L^2} \tag{3}$$

Por tanto, el valor de π es:

$$\pi = 4. * \frac{\text{Área Círculo}}{\text{Área Cuadrado}} \tag{4}$$

Podemos aproximar π empleando el método de Monte Carlo. La idea consiste en generar puntos al azar sobre el primer cuadrante (las coordenadas x, y de cada punto están comprendidas entre 0 y 1). De tal forma que contabilizaremos los puntos que están situados dentro del semicírculo, $N_{aciertos}$.

$$\pi = 4. * \frac{\text{Área Círculo}}{\text{Área Cuadrado}} \simeq \frac{N_{aciertos}}{N_{total}} \tag{5}$$

En este problema tratamos de computar el método de Montecarlo sobre el cuadrante indicado en 1. Para ello, en el código **b1_2.cpp** planteamos una estructura de programa basada en el cálculo mediante números aleatorios. Usamos principalmente dos librerias: **stdlib.h** y **time.h**. La primera nos permite hacer uso de la función rand(), mientras que la segunda nos genera una semilla aleatoria tomando como referencia el tiempo de la computadora. En caso de no incluirla, el programa toma siempre la misma semilla y generaria los mismos números aleatorios todo el rato.

Como se aprecia en la figura 1, el radio de la circunferencia y los lados de los cuadrados, los cuales toman valores entre cero y uno, cumplen el teorema de Pitágoras, por lo que establecemos la relación:

$$\sqrt{x^2 + y^2} = R \tag{6}$$

Entonces, una vez definidos x e y, se realizan N iteraciones, y en cada una se generan valores aleatorios entre 0 y 1 para x e y, y se añade una unidad al contador n en caso de que el módulo de dichos valores sea menor que uno.

Para comprobar el resultado de este programa, mostramos en una tabla el valor aproximado de π para distintos valores de N.

N	10	100	1000	10000	100000	
π	3.14062400	3.28000000	3.12000000	3.14120000	3.13832000	

Cuadro 2: Resultados aproximados para π con distintas iteraciones

Podemos observar que a mayor número de iteraciones, más decimales son mostrados en la aproximación.

4. Ejercicio 3

Construye un programa que realice las operaciones de suma, resta, multiplicación y división de dos números. El programa debe mostrar el siguiente menú:

Operación a realizar
1: Suma
2: Resta
3: Multiplicación
4: División
5: Salir

Una vez elegida la opción se solicitarán dos números y se mostrará el resultado correspondiente. El programa acabará cuando se elija la opción 5.

El programa consta de un bucle *switch* y un bucle *while*, de manera que la calculadora se ejecutará continuamente hasta que se reciba el comando de salida desde el menú principal[1]. La definición de los valores de entrada, que son los dos números que se operarán, se hará en cada uno de los bloques del *switch* en los que sea necesario, ya que en el caso de que se teclee el comando de finalización del programa no tendría sentido que se requirieran dos números para operar. Además, y a modo de control, se han añadido una respuesta *default* en el bucle *switch* para que, en caso de no insertar un valor recogido en el menú de posibles operaciones, el programa emita un aviso y retorne al usuario a este menú, y la función *check*[2] que ya ha sido empleada en ejercicios anteriores.

[1] Tal y como se especifica en el enunciado, en la tecla 5.

[2] Contenida en el archivo **checkers.cpp** y que, como ya se ha explicado en el primer ejercicio de este boletín, comprueba que los valores introducidos al programa son coherentes (esto es que pertenecen a la clase en la que han sido definidos).

5. Ejercicio 4

Implementa un programa que calcule el producto escalar de dos vectores de dimensión n >= 2. El programa debe leer los vectores desde un fichero.

El ejercicio 4 se realiza a través de un sólo archivo de código que lleva por nombre **b1_4.cpp**. Con la compilación de este único archivo tenemos un programa que calcula el producto escalar de dos vectores leídos desde otro archivo llamado **vector.txt**. Por tanto, modificando este último donde se introducen los datos de los vectores, podremos obtener el producto escalar de dos vectores cualquiera sin necesidad de volver a compilar.

Debemos comentar que para que el programa lea los vectores adecuadamente debemos estructurar los datos de la forma siguiente:

- La primera línea consta únicamente de la dimensión (tamaño) de los vectores.
- En la segunda línea se introducen las componentes del primer vector separadas por una tabulación.
- En la tercera línea se introducen las componentes del segundo vector separadas por una tabulación.

Como comentario adicional a los hechos en el propio código se quería recalcar que la lectura de los datos podría ser más eficiente si se introdujese de otra forma. Si por ejemplo leyésemos la primera componente de ambos vectores, posteriormente la segunda y así sucesivamente; podríamos realizarlo con un sólo bucle *for* pero parece que a la hora de introducir los datos en el **.txt** se hace más sencillo de la forma que se ha implementado.

Por último mostramos una serie de pruebas realizadas con el programa para asegurarnos de que realiza correctamente las operaciones:

- Producto escalar de vectores de dimensión $n = 2$

  Vector 1 = [1.5, 2.7]

  Vector 2 = [-5.3, 6.52]

  Resultado 9.654 (correcto)

- Producto escalar de vectores de dimensión $n = 3$

  Vector 1 = [1.5, 2.7, -4.27]

  Vector 2 = [-5.3, 6.52, -1.78]

  Resultado 17.2546 (correcto)

- Producto escalar de vectores de dimensión $n = 4$

  Vector 1 = [0.78, -1.21, 2.72, 8.09]

  Vector 2 = [5.73, -0.21, 1.56, -6.05]

  Resultado -39.9778 (correcto)

6. Ejercicio 5

Implementar en funciones los métodos de dicotomía y Newton-Raphson. Una vez que hayas comprobado su correcto funcionamiento aproximar el punto de intersección de las funciones $f_1(x) = e^{-x^2}$ y $f_2(x) = x$.

El ejercicio 5 se plantea en el programa llamado **b1_5.cpp** que proporciona la opción de elegir el método a utilizar. El código se complementa con otros dos auxiliares: el primero es el programa **checkers.cpp** con el header **checkers.h** ya explicado anteriormente, y el segundo **functions.cpp** también con su header **functions.h** correspondiente, dónde se definen las funciones para así no modificar el código principal cuándo necesitemos cambiar la función. Dicotomía funciona introduciendo los extremos del intervalo y el error de tipo epsilon, en cambio Newton funciona a partir de una semilla y un número máximo de iteracciones, sin atender a criterios de convergencia. El cálculo mediante Newton pedirá un valor para epsilon y para delta y si pasadas las N iteracciones no ha cumplido el test de parada concluirá con que no es convergente.

Se ha comprobado su correcto funcionamiento con diferentes funciones cómo se muestra a continuación:

- para el problema planteado en el enunciado se calculan las raices de:

$$f(x) = \exp(-x^2) - x$$

  1. Dicotomía: Intervalo (0,1), $\epsilon = 1 \cdot 10^{-10}$, resultado: **0.6529186405**
  2. Newton: semilla=5, N=100, $\epsilon = 1 \cdot 10^{-10}$, $\delta = 1 \cdot 10^{-10}$, resultado: **0.6529371688**

- para una ecuación con raíces múltiples:

$$f(x) = x^2 - 1$$

  1. Dicotomía: Intervalo (0.5,2), $\epsilon = 1 \cdot 10^{-10}$, resultado: **1.0000000000**
  2. Dicotomía: Intervalo (-3,3) , $\epsilon = 1 \cdot 10^{-5}$, resultado: **-1.0000019073** (arranca comprobando de -3 a 0, ve que ahí hai una raíz y la calcula aunque el intervalo incial no fuese adecuado)
  3. Dicotomía Intervalo (4,5), $\epsilon = 1 \cdot 10^{-5}$, resultado: no se cumple Bolzano en ese intervalo. (si no hay raíz ninguna entonces se detiene)
  4. Newton: semilla=100, N=1000, $\epsilon = 1 \cdot 10^{-8}$, $\delta = 1 \cdot 10^{-8}$, resultado: **1.0000714039** (vemos que con una semilla muy alejada, con un alto numero de iteracciones y con un error no muy exigente llega a la solución más cercana)
  5. Newton: semilla=-0.5, N=50, $\epsilon = 1 \cdot 10^{-10}$, $\delta = 1 \cdot 10^{-10}$, resultado: **-1.0000000465** (partiendo entre las dos raíces pero a la izquierda del cero llega a la solución negativa)
  6. Newton: semilla=0, N=500, $\epsilon = 1 \cdot 10^{-4}$, $\delta = 1 \cdot 10^{-4}$, resultado: No se ha alcanzado la convergencia (con un error bajo y muchas iteracciones no es capaz si se parte del cero, pues la derivada es nula y no va a poder calcular)