Implementación multihebra de aproximación numérica de una integral

Alumno: Miguel Ángel Fernández Gutiérrez

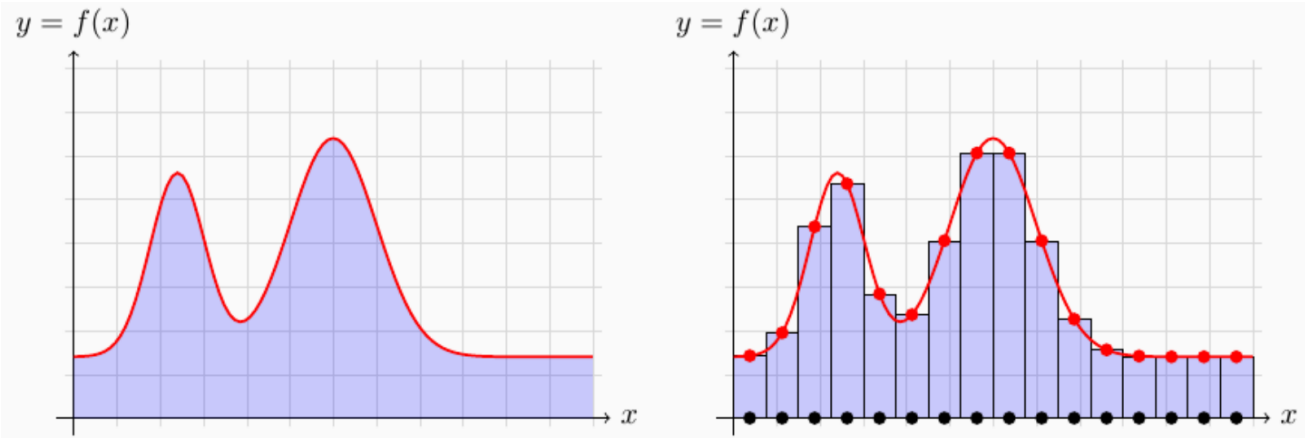
Introducción

En esta actividad, pretenderemos aproximar el valor de π, mediante la integral de la función arcotangente:

$$\pi = 4 \int_0^1 \arctan(x) \mathrm{d}x$$

Lo aproximaremos mediante la suma de las áreas de los rectángulos que quedan bajo la curva, variando en precisión. Podemos, en general, decir que:

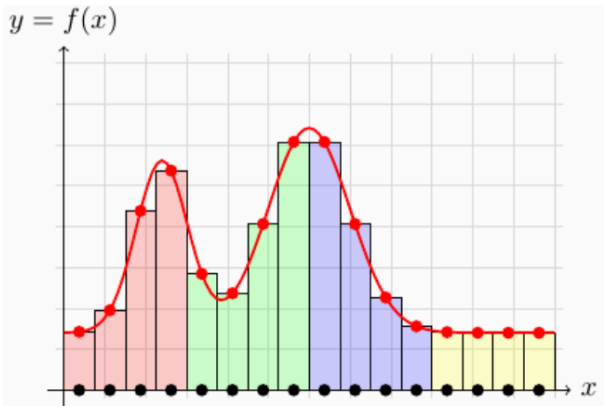
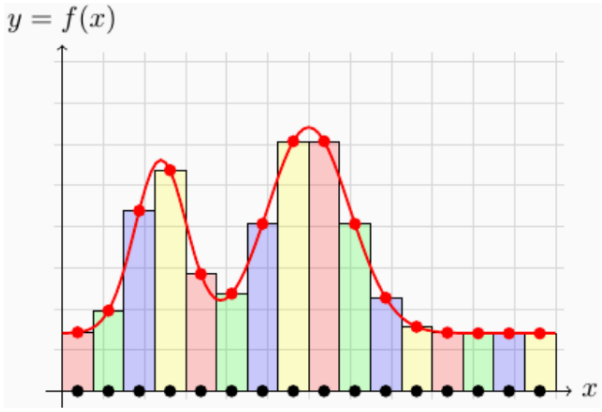
$$\pi \approx \frac{1}{m} \sum_{i=0}^{m-1} \frac{4}{1+x^2}$$



Implementaciones

Realizaremos tres implementaciones:

1. Implementación secuencial
2. Implementación con programación concurrente: reparto secuencial
3. Implementación con programación concurrente: reparto cíclico

Reparto secuencial	Reparto cíclico
	

1. Implementación secuencial

Teniendo en cuenta que `m` es el número de subintervalos o muestras y que `n` es el número de hebras, podemos implementar la aproximación secuencial como:

```
/**
 * @brief Evalúa la función a integrar:
 *            f(x) = 4/(1+x²)
 * @param x: valor en el que evaluar la función
 * @return Imagen de la función en x
 */
double f(double x) {
    return 4.0/(1.0+x*x);
}

/**
 * @brief Cálculo secuencial de la integral (implementación 1)
 * @return La suma total de la integral
 */
double integralSecuencial() {
    double suma = 0.0;
    for ( long i = 0; i < m; i++ )
        suma += f( (i+double(0.5))/m );
    return suma/m;
}
```

2. Implementación concurrente: reparto secuencial

En este caso, asignamos a cada hebra bloques contiguos.

```
/**
 * @brief Función que ejecuta cada hebra (implementación 2)
 * @param h: índice de la hebra, debe ser 0 <= h < n
 * @return Suma parcial que calcula la hebra h
 */
```

```cpp
double funcionHebra_1(long h) {
    double suma = 0.0;
    for ( long i = m/n*h; i < m/n*(h+1); i++ )
        suma += f( (i+double(0.5))/m );
    return suma/m;
}

/**
 * @brief Cálculo concurrente de la integral
 * @param funcionHebra: función a usar, funcionHebra_i (i=1,2)
 * @return La suma total de la integral
 */
double integralConcurrente(double (*funcionHebra)(long h)) {
    future<double> futuros[n];
    double suma = 0.0;

    for ( long int i = 0; i < n; i++ )
        futuros[i] = async(launch::async, funcionHebra, i);

    for ( long int i = 0; i < n; i++ )
        suma += futuros[i].get();

    return suma;
}
```

Donde en la función `integralConcurrente` he usado punteros a funciones para evitar redundancia en el código.

3. Implementación concurrente: reparto cíclico

En este caso, haremos un reparto uniforme a lo largo de los intervalos para cada hebra.

```cpp
/**
 * @brief Función que ejecuta cada hebra (implementación 3)
 * @param h: índice de la hebra, debe ser 0 <= h < n
 * @return Suma parcial que calcula la hebra h
 */
double funcionHebra_2(long h) {
    double suma = 0.0;
    for ( long i = h; i < m; i+=n )
        suma += f( (i+double(0.5))/m );
    return suma/m;
}

// mismo que anterior
double integralConcurrente(double (*funcionHebra)(long h));
```

Medición de tiempos

Para medir los tiempos, podemos usar el siguiente `main`.

```
int main() {
    time_point<steady_clock> inicio_1 = steady_clock::now();
    const double resultado_1 = integralSecuencial();
    time_point<steady_clock> fin_1 = steady_clock::now();

    time_point<steady_clock> inicio_2 = steady_clock::now();
    const double resultado_2 = integralConcurrente(funcionHebra_1);
    time_point<steady_clock> fin_2 = steady_clock::now();

    time_point<steady_clock> inicio_3 = steady_clock::now();
    const double resultado_3 = integralConcurrente(funcionHebra_2);
    time_point<steady_clock> fin_3 = steady_clock::now();

    duration<float, milli> tiempo_1 = fin_1 - inicio_1;
    duration<float, milli> tiempo_2 = fin_2 - inicio_2;
    duration<float, milli> tiempo_3 = fin_3 - inicio_3;

    const float porc_2 = 100.0*tiempo_2.count()/tiempo_1.count();
    const float porc_3 = 100.0*tiempo_3.count()/tiempo_1.count();

    constexpr double pi = 3.14159265358979238461;

    cout << "Número de muestras (m) ... : " << m << endl
         << "Número de hebras (n) ..... : " << n << endl
         << setprecision(18)
         << "Valor de pi ............... : " << pi << endl
         << "Aproximación 1 ........... : " << resultado_1 << endl
         << "Aproximación 2 ........... : " << resultado_2 << endl
         << "Aproximación 3 ........... : " << resultado_3 << endl
         << setprecision(5)
         << "Tiempo 1 ................. : " << tiempo_1.count() << " ms" << endl
         << "Tiempo 2 ................. : " << tiempo_2.count() << " ms" << endl
         << "Tiempo 3 ................. : " << tiempo_3.count() << " ms" << endl
         << setprecision(4)
         << "Porcentaje impl.2/impl.1 . : " << porc_2 << "%" << endl
         << "Porcentaje impl.3/impl.1 . : " << porc_3 << "%" << endl
         << endl
         << "Alumno: Miguel Ángel Fernández Gutiérrez (3º DGIIM)" << endl
         << endl
         << "LEYENDA" << endl
         << "-----------------------------------------------" << endl
         << "Implementación 1: secuencial" << endl
         << "Implementación 2: concurrente, reparto secuencial" << endl
         << "Implementación 3: concurrente, reparto cíclico" << endl;
}
```

Quedando, al ejecutar el código, el siguiente resultado:

```
Número de muestras (m) ... : 1073741824
```

```
Número de hebras (n) ..... : 4
Valor de pi .............. : 3.14159265358979312
Aproximación 1 ........... : 3.14159265358998185
Aproximación 2 ........... : 3.14159265358982731
Aproximación 3 ........... : 3.14159265358978601
Tiempo 1 ................. : 15914 ms
Tiempo 2 ................. : 4139.3 ms
Tiempo 3 ................. : 4147.1 ms
Porcentaje impl.2/impl.1 . : 26.01%
Porcentaje impl.3/impl.1 . : 26.06%

Alumno: Miguel Ángel Fernández Gutiérrez (3º DGIIM)

LEYENDA
------------------------------------------------
Implementación 1: secuencial
Implementación 2: concurrente, reparto secuencial
Implementación 3: concurrente, reparto cíclico
```

Vemos claramente que los programas concurrentes son mucho más rápidos que el secuencial. Sin embargo, no hay diferencia significativa entre sendas implementaciones concurrentes.

Además, vemos que el uso de los procesadores tiene sentido: