Capítulo 7

Derivación e integración numérica

7.1. Derivación Numérica

El programa `deriv.cpp` calcula las derivadas numéricas con una precisión prefijada `eps` en un punto dado, utilizando el método de extrapolación al límite de Richardson. Puede calcular derivadas de orden 1, 2, 3 y 4, para las que utiliza las fórmulas centrales con un número impar de puntos (3 puntos para las derivadas primera y segunda y 5 puntos para las derivadas tercera y cuarta). La derivación numérica es una de las bestias negras del cálculo numérico y no se deben de exigir errores excesivamente pequeños al aplicar estas fórmulas.

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <vector>
typedef double Real;
using namespace std;
typedef Real(*func) (Real);

template < class T > inline T derval(func F, T x, T h, int ord)
{
//Devuelve el valor de la derivada de F de orden ord
    switch (ord) {
        case 1:
            return (F(x + h) - F(x - h)) / (2 * h);
            break;
        case 2:
            return (F(x + h) - 2. * F(x) + F(x - h)) / (h * h);
            break;
        case 3:
            return (F(x + 2. * h) - 2. * F(x + h) + 2. * F(x - h)
                    - F(x - 2. * h)) / (2. * h * h * h);
```

```
            break;
        case 4:
            return (F(x + 2. * h) - 4 * F(x + h) + 6 * F(x)
                    - 4 * F(x - h) + F(x - 2. * h)) / (h * h * h * h);
            break;
        default:
            cout << "orden de la derivada = " << ord << "  invalido" << endl;
    }
}


template < class T > vector < T > deriv(func F, T x, T h, T eps,
                                        int nprec, int ord)
{
    if (nprec == 0)
        nprec = 6;
    vector < T > der;
    int k = 1;
    int n = 1;

//Derivada.Primera fila;
    T   deri = derval(F, x, h, ord);
    int nwidth = nprec + int (log10(deri)) + 4;

    der.push_back(deri);
    T   delta = 1;

    cout <<  " h = " <<  setprecision(nprec) << h << "  Derivada  orden " <<
         ord << setw(nwidth) << setprecision(nprec)<< " =  "  << deri << endl << endl;
    while (delta > eps) {
        h = h / 2;
//Calculo de la derivada con h / 2
        deri = derval(F, x, h, ord);
        cout << " h = "  << h << "  Derivada orden  " << ord << setw(nwidth)
            << setprecision(nprec) << " =  " << deri << endl;
//Relleno de la fila n + 1
        der.push_back(deri);
        int index = n * (n + 1) / 2;

        for (int i = 0; i < n; i++) {
            deri = (pow(4., i + 1) * der[index + i] - der[index + i - n])
                   / (pow(4., i + 1) - 1);
            der.push_back(deri);
```

```
        }
        cout << endl;
        delta = abs(deri - der[index - 1]);
//cout << "delta= " << delta << endl;
        if (n > 10) {
            cout << "maximo numero de iteraciones" << endl;
            break;
        }
        n++;
    }

//impresion del triangulo de Richardson
    int nn = der.size();
    int nk = 1;
    int index = 0;

    while (nn > 0) {
        for (int j = index; j < index + nk; j++)
            cout << setw(nwidth) << setprecision(nprec)
                << der[j] << " ";
        cout << endl;
        index += nk;
        nk++;
        nn -= nk;
    }
    return der;
}


template < class T > inline T f(T x)
{
    return exp(x);
}


int
main()
{
//precision de resultados
    int mm = 18;

//LLamada a deriv
    Real eps = 1.e-7;
```

```
    Real x = 1.;
    Real h = 0.2;

    vector < Real > der = deriv < Real > (f < Real >, x, h, eps, mm, 1);
    int n = der.size();

//Impresion de resultados
    cout << " Derivada = " << der[n - 1] << endl;
    cout << "Valor exacto = " << exp(1.) << endl;

    cout << endl << endl << endl;

    der = deriv < Real > (f < Real >, x, h, eps, mm, 2);
    n = der.size();
//Impresion de resultados
    cout << " Derivada = " << der[n - 1] << endl;
    cout << "Valor exacto = " << exp(1.) << endl;

    cout << endl << endl << endl;

    der = deriv < Real > (f < Real >, x, h, eps, mm, 3);
    n = der.size();
//Impresion de resultados
    cout << " Derivada = " << der[n - 1] << endl;
    cout << "Valor exacto = " << exp(1.) << endl;

    cout << endl << endl << endl;

    der = deriv < Real > (f < Real >, x, h, eps, mm, 4);
    n = der.size();
//Impresion de resultados
    cout << " Derivada = " << der[n - 1] << endl;
    cout << "Valor exacto = " << exp(1.) << endl;

    return 0;
}
```

7.2. Regla trapezoidal

El programa `trapN.cpp` calcula la integral de una función en un intervalo $[a,b]$ mediante la regla trapezoidal con un número *nintervalos* de subintervalos de integración, elegidos por el usuario:

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <vector>
typedef double Real;
using namespace std;
typedef Real(*func) (Real);

template < class T > inline T f(T x)
{
//Definir aqui la funcion a integrar
    return exp(x);
}


double
trap(func F, Real a, Real b, int n)
{
    Real h = (b - a) / n;
    Real S = (F(a) + F(b));

    for (int j = 1; j < n; j++) {
        S = S + 2 * F(a + j * h);
    }
    Real Integral = S * h / 2;

    cout << "Trapezoidal  " << n << " subintervalos=" << setw(20)
        << setprecision(18) << Integral << endl;

    return Integral;
}


int
main()
{
    int nintervalos = 200;

    cout << "Entrar numero subintervalos" << endl;
    cin >> nintervalos;

//Limites inferior y superior
    Real a = 0;
```

```
    Real b = 1.;

//LLamada a trap
    Real trapezoidal = trap(f, a, b, nintervalos);
//Impresion de resultados
    Real integral = exp(1) - 1;
    cout << " Integral regla trapezoidal = " << trapezoidal << endl;
    cout << "Valor exacto = " << integral << endl;
    cout << " Error  = " << abs(integral - trapezoidal) << endl;
    return 0;
}
```

El programa `trapezoidal.cpp` calcula la integral de una función `f,` definida en el programa, en el intervalo $[a,b]$, mediante la regla trapezoidal, doblando el número de intervalos hasta alcanzar una precisión prefijada `eps,` fijada por el usuario. Los parámetros a fijar por el usuario son los límites de integración `a` y `b` y la precisión `eps` en `main(),` y la función en `f`.

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <vector>
typedef double Real;
using namespace std;
typedef Real(*func) (Real);

template < class T > vector < T > trapezoidal(func F, T a, T b, T eps, int nprec)
{
    if (nprec == 0)
        nprec = 6;
    vector < T > trapz;
    int k = 1;
    int n = 1;
    T   h = (b - a);
    T   S = (F(a) + F(b));
    T   trap = S * h / 2;
    int nwidth = nprec + int (log10(trap)) + 4;

//regla trapezoidal de dos puntos.Primera fila;
    trapz.push_back(trap);
    T   delta = 1.;
    T   oldtrap = trap;

    cout << "Trapezoidal  " << k << " subintervalos=" << setw(nwidth)
```

```
        << setprecision(nprec) << trap << endl;
    while (delta > eps) {
        h = h / 2;
//Calculo de la regla trapezoidal con el doble de intervalos
// utilizando T_2N = (T_N + M_N) / 2
        T sum = T(0);
//Calculo de M_N
        for (int j = 0; j < k; j++) {
            sum = sum + 2 * F(a + (2 * j + 1) * h);
        }
        S = (S + sum);
        trap = S * h / 2;
        k *= 2;
        cout << "Trapezoidal  " << k << " subintervalos=" << setw(nwidth)
            << setprecision(nprec) << trap << endl;
        delta = abs(trap - oldtrap);
        oldtrap = trap;
        trapz.push_back(trap);
        n++;
    }

    return trapz;
}


template < class T > inline T f(T x)
{
    return exp(x);
}


int
main()
{
//precision de resultados
    int mm = 18;

//LLamada a Trapezoidal
    Real a = 0., b = 1.;;
    Real eps = 1.e-13;

    vector < Real > trap = trapezoidal < Real > (f < Real >, a, b, eps, mm);
    int n = trap.size();
```

```
//Impresion de resultados
    Real integral = exp(1.) - 1.;
    cout << " Integral regla Trapezoidal = " << trap[n - 1] << endl;
    cout << pow(2., n) << " subintervalos; " << n << " iteraciones" << endl;
    cout << " Error  estimado= " << abs(trap[n - 1] - trap[n - 2]) << endl;
    cout << "Valor exacto = " << integral << endl;
    cout << " Error real = " << abs(integral - trap[n - 1]) << endl;
    return 0;
}
```

7.3. Regla de Simpson

El programa `simpsonN.cpp` integra una función mediante la regla de Simpson, de forma similar a `trapN.cpp`, en un número par `nintervalos` subintervalos de integración, sin estimar la precisión obtenida:

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <vector>
typedef double Real;
using namespace std;
typedef Real(*func) (Real);

template < class T > inline T f(T x)
{
//Definir aqui la funcion a integrar
    return exp(x);
}


Real
simpson(func F, Real a, Real b, int n)
{
//Para obteber n subintervalos h = (b - a) / n
    Real h = (b - a) / n;
    Real Suma = (F(a) + F(b));
    int N = n / 2;

    for (int j = 0; j < N - 1; j++) {
        Suma = Suma + 4 * F(a + (2 * j + 1) * h) + 2 * F(a + (2 * j + 2) * h);
```

```
    }
    Suma = Suma + 4 * F(a + (2 * N - 1) * h);
    Real Integral = Suma * h / 3;

    cout << "Simpson  " << n << " subintervalos= " << setw(20) <<
        setprecision(18) << Integral << endl;
    return Integral;
}


int
main()
{
//precision de resultados(eps) y numero de cifras impresas(mm)
    int ncifras = 18;
    int nintervalos = 200;

    cout << "Entrar numero subintrervalos" << endl;
    cin >> nintervalos;
    if (2 * (nintervalos / 2) != nintervalos) {
        cout << "Error. El numero se subintervalos debe ser par" << endl;
        cout << "Entrar numero subintrervalos" << endl;
        cin >> nintervalos;
    }
//Limites inferior y superior
    Real a = 0;
    Real b = 1.;

//LLamada a Simpson
    Real Simpson = simpson(f, a, b, nintervalos);
//Impresion de resultados
    Real integral = exp(1) - 1;
    cout << " Integral regla Simpson = " << Simpson << endl;
    cout << "Valor exacto = " << integral << endl;
    cout << " Error  = " << abs(integral - Simpson) << endl;
    return 0;
}
```

Un programa adicional, que utiliza una función en forma de template, en la que se calcula la regla de Simpson como el primer paso de extrapolación al límite de la regla trapezoidal, es `simpsontrap.cpp`

```
#include <cmath>
```

```
#include <iostream>
#include <iomanip>
#include <vector>
typedef double Real;
using namespace std;
typedef Real(*func) (Real);

template < class T > vector < T > simpson(func F, T a, T b, T eps, int nprec)
{
    if (nprec == 0)
        nprec = 6;
    vector < T > simp;
    int k = 1;
    int n = 1;
    T   h = (b - a);
    T   S = (F(a) + F(b));
    T   trap = S * h / 2;
    int nwidth = nprec + int (log10(trap)) + 4;

//regla trapezoidal de dos puntos.Primera fila;
    T   delta = 1.;
    T   oldtrap = trap;
    T   simpson, oldsimpson = T(0);

    while (delta > eps) {
        h = h / 2;
//Calculo de la regla trapezoidal con el doble de intervalos
// utilizando T_2N = (T_N + M_N) / 2
        T sum = T(0);
//Calculo de M_N
        for (int j = 0; j < k; j++) {
            sum = sum + 2 * F(a + (2 * j + 1) * h);
        }
//trap regla trapezoidal con 2 N subintervalos
        S = S + sum;
        trap = S * h / 2;
        k *= 2;
//cout << "Trapezoidal  " << k << " subintervalos=" << setw(nwidth)
// <<setprecision(nprec) << trap << endl;
//Calculo de Simpson como S_2N = (4 T_2N - T_N) / 3
        simpson = (4 * trap - oldtrap) / 3;
        cout << "Simpson  " << k << " subintervalos=" << setw(nwidth)
            << setprecision(nprec) << simpson << endl;
```

```
        delta = abs(simpson - oldsimpson);
        oldsimpson = simpson;
        oldtrap = trap;
        simp.push_back(simpson);
        n++;
    }

    return simp;
}


template < class T > inline T f(T x)
{
//Definir aqui la funcion a integrar
    return exp(x);
}


int
main()
{
//precision de resultados(eps) y numero de cifras impresas(mm)
    int ncifras = 18;
    Real eps = 1.e-12;

//Limites inferior y superior
    Real a = 0;
    Real b = 1.;

//LLamada a Simpson
// vsimpson vector con las sucesivas iteraciones de Simpson
    vector < Real > vsimpson = simpson < Real > (f < Real >, a, b, eps, ncifras);
    int n = vsimpson.size();

//Impresion de resultados
    Real integral = exp(1) - 1;
    cout << " Integral regla Simpson = " << vsimpson[n - 1] << endl;
    cout << pow(2., n) << " subintervalos; " << n << " iteraciones" << endl;
    cout << " Error  estimado= " << abs(vsimpson[n - 1] - vsimpson[n - 2]) << endl;
    cout << "Valor exacto = " << integral << endl;
    cout << " Error real = " << abs(integral - vsimpson[n - 1]) << endl;
    return 0;
}
```

En este programa se hace uso del hecho de que las evaluaciones de la función que hacen falta para pasar de T_N a T_{2N} es la regla del punto medio M_N

$$T_{2N} = \frac{T_N + M_N}{2}$$

y de aquí calculamos la regla de Simpson como

$$S_{2N} = \frac{4T_{2N} - T_N}{3}$$

Por lo tanto para pasar de T_N a S_{2N}sólo hacen falta las evaluaciones de la función necesarias para calcular M_N. El programa evalúa la diferencia entre la integral en dos números consecutivos de subintervalos N y $2N$ como estima del error numérico:

$$Error(S_{2N}) = |S_{2N} - S_N|$$

y vuelve a doblar el número de pasos de integración si no se alcanza la precisión requerida.

7.4. Integración Gaussiana

El programa `gausslegendre.cpp` es una funcion (sin main) que realiza la integración gaussiana en el intervalo [-1,1] utilizando los ceros del polinomio de Legendre de grado 10 como abcisas

```
gauslegendre(Real func(const Real), const Real a, const Real b)
{

    static const Real x[] = {
        0.1488743389816312, 0.4333953941292472,
        0.6794095682990244, 0.8650633666889845, 0.9739065285171717
    };

    static const Real w[] = {
        0.2955242247147529, 0.2692667193099963,
        0.2190863625159821, 0.1494513491505806, 0.0666713443086881
    };
    Real xr, xm, dx, s;
    xm = 0.5 * (b + a);
    xr = 0.5 * (b - a);
    s = 0;
    for (int j = 0; j < 5; j++) {
        dx = xr * x[j];
```

```
      s += w[j] * (func(xm + dx) + func(xm - dx));
    }
    return s *= xr;
}
```

El siguiente programa utiliza los ceros de los polinomios de Chebychev para realizar la integral de una función con peso

$$\omega(x) = \frac{1}{\sqrt{1-x^2}}$$

en el intervalo [-1,1].

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

//Programa de integracion por Gauss - Chebychev;

inline double
fun(double x)
{
//funcion a integrar en[-1, 1] dividida por el peso sqrt(1 - x * x) ^ -1
    return asin(x);
}

int main(void)
{
    double s = 0., x;
    double pi = 4 * atan(1.);
    int n = 48;
//Numero de puntos de integracion
    cout.precision(16);
    for (int i = 0; i < n; i++) {
        x = cos((2 * i + 1.) * pi / (2 * n));
        s = s + fun(x);
    }
    s = s * pi / n;
    cout << setw(20) << "integral=" << setw(20) << s << endl;
    //    cout << setw(20) << "exacto=" << setw(20) << 4 / pi << endl;
    return 0;
}
```

7.5. Integración por Romberg

El programa `romberg.cpp` implementa el método de extrapolación al límite de la regla trapezoidal para calcular la integral por el método Romberg:

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <vector>
typedef double Real;
using namespace std;
typedef Real(*func) (Real);

template < class T > vector < T > romberg(func F, T a, T b, T eps, int nprec)
{
    if (nprec == 0)
        nprec = 6;
    vector < T > romb;
    int k = 1;
    int n = 1;
    T   h = (b - a);
    T   S = (F(a) + F(b));
    T   trap = S * h / 2;
    int nwidth = nprec + int (log10(trap)) + 4;

//regla trapezoidal de dos puntos.Primera fila;
    romb.push_back(trap);
    T   delta = 10.;

    cout << "Trapezoidal  " << k << " subintervalos=" << setw(nwidth)
        << setprecision(nprec) << trap << endl;
    while (delta > eps) {
        h = h / 2;
//Calculo de la regla trapezoidal con el doble de intervalos
// utilizando T_2N = (T_N + M_N) / 2
        T sum = T(0);
//Calculo de M_N
        for (int j = 0; j < k; j++) {
            sum = sum + 2 * F(a + (2 * j + 1) * h);
        }
        S = (S + sum);
        trap = S * h / 2;
        k *= 2;
```

```
        cout << "Trapezoidal  " << k << " subintervalos=" << setw(nwidth)
            << setprecision(nprec) << trap << endl;
//Relleno de la fila n + 1
        romb.push_back(trap);
        int index = n * (n + 1) / 2;

        for (int i = 0; i < n; i++) {
            trap = (pow(4., i + 1) * romb[index + i] - romb[index + i - n])
                  / (pow(4., i + 1) - 1);
            romb.push_back(trap);
        }
        cout << endl;
        delta = abs(trap - romb[index - 1]);
//cout << "delta= " << delta << endl;
        n++;
    }

//impresion del triangulo de Romberg
    int nn = romb.size();
    int nk = 1;
    int index = 0;

    while (nn > 0) {
        for (int j = index; j < index + nk; j++)
            cout << setw(nwidth) << setprecision(nprec)
                << romb[j] << " ";
        cout << endl;
        index += nk;
        nk++;
        nn -= nk;
    }
    return romb;
}


template < class T > inline T f(T x)
{
    return exp(x);
}


int
main()
```

```
{
//digitos con que se imprimen los resultados
    int digits = 18;

//Precision de calculo de la integral
    Real prec = 1.e-13;

//Limite inferior de integracion
    Real a = 0.;

//Limite superior de integracion
    Real b = 1.;

//LLamada a Romberg
    vector < Real > romb = romberg < Real > (f < Real >, a, b, prec, digits);
    int n = romb.size();

//Impresion de resultados
    cout << setprecision(digits) << " Integral por Romberg = " << romb[n - 1] << endl;
    cout << " Estima del error = "<< fixed << abs(romb[n-1]-romb[n-2]) << endl;
    cout << "Valor exacto = " << exp(1.) - 1. << endl;
    return 0;
}
```

Se evalúan los pasos de extrapolación de Richardson necesarios para que la estima del error sea menor que una cantidad prefijada `eps`. La regla trapezoidal T_{2n} se calcula a partir de T_n mediante la relación

$$T_{2n} = \frac{T_n + M_n}{2}$$

donde M_n es la regla del punto medio. De esta forma se evita repetir evaluaciones de la función.

7.6. Integrales multidimensionales

El programa `integ3d.cpp` realiza la integración en tres dimensiones de una función dada. En la forma en que se está escrito. se puede utilizar Simpson, Romberg o Gauss-Legendre para realizar las integrales unidimensionales, aunque es inmediato añadir métodos adicionales. Se recurre a variables globales, definidas fuera de las funciones, para definir el punto en que se integra y los parámetros de la función que se integra. El programa llama de forma recursiva al método de integración unidimensional utilizado para calcular la integral, que debe de elegir el usuario.

```
#include <iostream>
#include <iomanip>
```

```
#include <cmath>
#include <vector>
using namespace std;

//Programa de integracion en 3 d recursivo;
// Siguiendo el metodo de Numerical Recipes


typedef double Real;
typedef Real(*func) (Real);

//permite definir la precision del programa
static Real xmax;
//variable global xmin = -xmax

// Real y1(const Real), y2(const Real) :curvas superiores e inferiores
// Real z1(const Real, const Real): superficie inferior
// Real z2(const Real, const Real): superficie superior
Real xglob, yglob;

//variables globales
Real z1(const Real x, const Real y)
{
    return -sqrt(xmax * xmax - x * x - y * y);
}


Real
z2(const Real x, const Real y)
{
    return sqrt(xmax * xmax - x * x - y * y);
}


Real
y1(const Real x)
{
    return -sqrt(xmax * xmax - x * x);
}


Real
y2(const Real x)
```

```
{
    return sqrt(xmax * xmax - x * x);
}


Real
fun(const Real x, const Real y, const Real z)
{
    return x * x + y * y + z * z;
//función a integrar
}


//Regla de integracion 1 d; tiene como argumento una funcion de 1 variable
Real gausslegendre(Real func1(const Real), const Real a, const Real b)
{
//Metodo gaussiano.Es mas eficiente
    static const Real x[] = {
        0.1488743389816312, 0.4333953941292472, 0.6794095682990244,
        0.8650633666889845, 0.9739065285171717
    };
    static const Real w[] = {
        0.2955242247147529, 0.2692667193099963, 0.2190863625159821,
        0.1494513491505806, 0.0666713443086881
    };
    int j;
    Real xr, xm, dx, s;

    xm = 0.5 * (b + a);
    xr = 0.5 * (b - a);
    s = 0;
    for (j = 0; j < 5; j++) {
        dx = xr * x[j];
        s += w[j] * (func1(xm + dx) + func1(xm - dx));
    }
    return s *= xr;
}


Real simpson(Real func1(const Real), const Real a, const Real b)
{

//Simpson 2 N intervalos
```

```
    int N = 10;
    Real h = (b - a) / (2 * N);
    Real xm, s;

//Las raices cuadrados no definidas si x < 0. Resrar e - 15 en bordes
    s = func1(a + 1.e-15) + 4. * func1(b - h) + func1(b - 1.e-15);
    xm = a + h;
    for (int j = 1; j < N; j++) {
        s += (4 * func1(xm) + 2 * func1(xm + h));
        xm += 2 * h;
    }
//cout << a << " " << b << "  " << s << "  " << h << endl;
    return s *= h / 3;
}


template < class T > T romberg(Real F(const Real), T a, T b, T eps, int nprec)
{
    if (nprec == 0)
        nprec = 6;
    vector < T > romb;
    int k = 1;
    int n = 1;
    T   h = (b - a);
    T   S = (F(a + 1.e-15) + F(b - 1.e-15));
    T   trap = S * h / 2;
    int nwidth = nprec + int (log10(trap)) + 4;

//regla trapezoidal de dos puntos.Primera fila;
    romb.push_back(trap);
    T   delta = 10.;

    while (delta > eps) {
        h = h / 2;
//Calculo de la regla trapezoidal con el doble de intervalos
// utilizando T_2N = (T_N + M_N) / 2
        T sum = T(0);

//Calculo de M_N
        for (int j = 0; j < k; j++) {
            sum = sum + 2 * F(a + (2 * j + 1) * h);
        }
        S = (S + sum);
```

```
        trap = S * h / 2;
        k *= 2;

//Relleno de la fila n + 1
        romb.push_back(trap);
        int index = n * (n + 1) / 2;

        for (int i = 0; i < n; i++) {
            trap = (pow(4., i + 1) * romb[index + i] - romb[index + i - n])
                   / (pow(4., i + 1) - 1);
            romb.push_back(trap);
        }
        delta = abs(trap - romb[index - 1]);
        n++;
    }

    return trap;
}


Real integ1d(Real func1(const Real), const Real a, const Real b)
{
    Real eps = 1.e-7;

//return gausslegendre(func1, a, b);
//permite seleccionar el metodo deseado en 1 d
    return romberg < Real > (func1, a, b, eps, 10);
//return simpson(func1, a, b);
}


//Llamada recursiva integ3d->f1->f2->f3->func
Real f3(const Real z)
{
    return fun(xglob, yglob, z);
}


Real
f2(const Real y)
{
    yglob = y;
    return integ1d(f3, z1(xglob, y), z2(xglob, y));
```

```
}


Real
f1(const Real x)
{
    xglob = x;
    return integ1d(f2, y1(x), y2(x));
}


Real integ3d(Real fun(const Real, const Real, const Real), const Real x1,
const Real x2)
{
    return integ1d(f1, x1, x2);
}


//fin ciclo recursivo

int
main(void)
{
    const int nradio = 10;

//numero de radios para calcular V
    const Real PI = 3.141592653589793238;
    Real xmin, s;

    cout << "Integral de r**2 en volumen esfera" << endl << endl;
    cout << setw(11) << "radio" << setw(13) << "INTEG3D";
    cout << setw(12) << "exacto" << endl << endl;
    cout << fixed << setprecision(6);
    for (int i = 0; i < nradio; i++) {
        xmax = 0.1 * (i + 1);
        xmin = -xmax;
        s = integ3d(fun, xmin, xmax);
        cout << setw(12) << xmax << setw(13) << s;
        cout << setw(12) << 4.0 * PI * pow(xmax, Real(5.0)) / 5.0 << endl;
    }
    return 0;
}
```

7.7. La clase vector de la librería estándar

En los programas anteriores se ha utilizado la clase `vector` de la librería estándar. No confundir esta clase con la clase Vector de la librería TNT. La clase vector es esencialmente un contenedor de magnitudes vectoriales, y no tiene ningín operador sobrecargado. Se puede invocar si se incluye la cabecera `<vector>`:

```
#include <vector>
```

Un elemento se declara de la clase `vector` mediante, por ejemplo,

```
vector<double>mivector;
```

Podemos rellenar el vector mediante la sentencia

```
double x=5.;
mivector.push_back(x); //introduce x en mi vector
```

Podemos saber cuantos elementos hemos introducido mediante la sentencia

```
int dim=mivector.size();
```

Si con estas características de los vectores nos basta, el utilizar la clase `vector` es una opción adecuada.

7.8. Ejercicios a presentar como memoria

1. Estudiar las derivadas numéricas primera, segunda, tercera y cuarta de la función $\exp(x)$ en $x = 1$, con $h = 0{,}2,\ 0{,}1,\ 0{,}05$, y las correspondientes extrapolaciones al límite, mediante el programa `deriv.cpp`. Repetir lo mismo para la función $\sqrt{x}$ en $x = 1$. Comentad los resultados (precisión, número de evaluaciones de la función para cada orden de derivación) .

2. Consideremos la función $\cos(x)$. Dibujadla con `Gnuplot`. ¿Cúal es su principal característica? Calcular su integral en el intervalo $[-\pi/2, \pi/2]$ directamente y como el doble de la integral en el intervalo $[0, \pi/2]$, con la regla trapezoidal con 10 subintervalos de integración (utilizad el programa `trapN.cpp`) ¿Cual de los dos métodos da menos error? Dar una explicación ¿Con cuántos puntos hay que calcular la integral directamente (entre $[-\pi/2, \pi/2]$) para que se obtenga el mismo error que por el segundo método?

3. Calculad la integral de la función $\exp(-x^2)$ entre -1 y 1 por las reglas trapezoidal, Simpson y Romberg con un error de 10^{-8}(Utilizad los programas `trapezoidal.cpp`, `simpsontrap.cpp` y `romberg.cpp`) ¿Que valor de h hace falta en cada método? ¿Cúantas evaluaciones de la función son necesarias en cada método?

4. Calcular el volumen de un elipsoide de revolución de semiejes $a = 1$, $b = 2$ y $c = 3$, integrando recursivamente mediante la regla de Simpson con 100 puntos, mediante la regla de Romberg con con un error de 10^{-7}, y mediante la regla de Gauss-Legendre con 10 puntos (Utilizad el programa `integ3d.cpp`, seleccionando la regla de integración correspondiente en cada caso, y poniendo los parámetros de la función como variables globales.

Deberéis cambiar las funciones de las curvas y superficies superior e inferior, a las adecuadas en vuestro caso. El programa está preparado para la integral de r^2 sobre una esfera). Comparar el resultado numérico con el valor exacto. La ecuación del elipsoide es

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1$$

y su volumen $\frac{4}{3}\pi abc$. Comentar el resultado.

5. (Alternativo al 4, grupo C). Comprobad Utilizando un programa basado en trapN.cpp, que el error depende de h^2, para el caso del ejerccio 3. (Calculad el error exacto como el valor exacto menos la integral numérica y multiplicadlo por N^2. Debe de salir aproximadamente la misma cantidad para diversos valores de N). Comprobad igualmente que el error del método de Simpson varía como h^4.