Prácticas de Cálculo Numérico en C++

Juan José Gómez Cadenas

Juan Zúñiga Román

Departament de Física Atòmica Molecular i Nuclear

Facultat de Física
Universitat de València
Curso 2003-2004

Índice general

Capítulo 1

Introducción al C++ (I)

1.1. Introducción

En esta práctica vamos a aprender el manejo elemental del compilador de GNU, `g++` y el programa gráfico, `gnuplot`. Compilaremos, enlazaremos y ejecutaremos una serie de programas muy sencillos que ilustran los rudimentos de C++.

Los ejercicios que se proponen son variantes muy sencillas de los ejemplos que hemos comentado en el primer capítulo del curso de Cálculo Numérico.

1.2. El entorno de trabajo

Las prácticas del curso de cálculo numérico pueden realizarse indistintamente utilizando Windows o Linux. Ambos sistemas están disponibles en el laboratorio y los alumnos pueden elegir cuál de ellos usar.

Esto es posible gracias a que hemos instalado `cygwin`[1] en el sistema Windows98 de los ordenadores del laboratorio; `cygwin` proporciona en Windows un entorno de trabajo similar al de Linux. En particular, permite el uso de la familia de compiladores gcc[2], parte del proyecto de software libre GNU[3]. g++ es el compilador de C++ de la familia gcc.

Necesitamos, además, un editor apropiado para manejar programas de C++. Utilizaremos emacs[4], un potentísimo editor de textos, capaz entre otras muchas cosas de reconocer la sintaxis de C++. Es software libre y puede utilizarse tanto en Linux como en Windows.

A lo largo del curso vamos a necesitar también, a menudo, representar funciones gráficamente. Para ello utilizaremos el programa gnuplot[5] el cual, a pesar de ofrecer una potente funcionalidad es muy sencillo de utilizar para aplicaciones simples.

Finalmente, todas las notas y manuales del curso se han preparado utilizando latex[6] y el programa WinEdt[1].

1.3. Organización de un ejecutable

Un programa en C++ se escribe a partir de uno o varios ficheros de código fuente. Por convención, los ficheros con extensión `.cpp` contienen código en C++, mientras que los ficheros de cabecera, con extensión `.h` contienen típicamente declaraciones de tipos y funciones (las cabeceras estándar de C++, como se comentó en el primer tema del curso de Cálculo Numérico no llevan la extensión `.h`).

Para construir un ejecutable, primero debemos compilar el fichero o ficheros con el código fuente. El compilador genera un fichero objeto, escrito en código máquina, generalmente, con extensión `.o`. Este

[1] WinEdt (http://www.winedt.com) es un editor de textos particularmente apropiado para trabajar en latex. Se distribuye como "shareware".

fichero todavía no es ejecutable, porque aún no se han resuelto las eventuales llamadas a funciones externas (por ejemplo a las funciones definidas en la *Std*). El siguiente paso es generar un ejecutable, extensión `.exe`, a partir de uno o más ficheros objeto. Un ejecutable es un fichero escrito en código máquina, al igual que un fichero objeto, pero en el cual, todas las referencias a funciones externas han sido resueltas. Como su nombre indica, este fichero puede ejecutarse, produciendo un resultado, que en ciertas ocasiones, se parece al que pretendía lograr el esforzado programador.

Cuando sólo necesitamos compilar un fichero y no hay referencias a bibliotecas externas en nuestro código (a excepción de aquellas que el compilador sabe encontrar automáticamente, como la *Std*), podemos compilar y enlazar en un solo paso, sin necesidad de crear un fichero objeto intermedio, como veremos en los ejemplos que siguen.

1.4. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso /home/calnum.
2. A continuación cread una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[2]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.
3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[3] el fichero comprimido correspondiente a la práctica uno `P1.tgz`.
4. Extraed el contenido de `P1.tgz` al directorio `P1`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P1.tgz
   ```
5. Examinad el contenido de la carpeta `P1`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

   ```
   cd P1
   ```

   para entrar en el directorio `P1`, seguida de:

   ```
   ls
   ```

   que nos produce un listado de los programas que vamos a examinar en la práctica.

1.5. Hola Mundo!

Comencemos compilando el primer programa del curso, `holaMundo.cpp`. La instrucción:

```
$ g++ holaMundo.cpp
```

[2]Kavafis es un alumno del curso, amigo del profesor, al que nos referiremos como ejemplo. Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido.

[3]http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

resulta en compilar el programa y enlazarlo con la *Std* , produciendo un ejecutable cuyo nombre por defecto (en cygwin) es `a.exe` (en Linux suele ser `a.out`). Para ejecutarlo tecleamos en la pantalla (el símbolo $ representa el *prompt* típico de Linux, aunque en vuestra pantalla quizás sea otro símbolo parecido, por ejemplo >, − >, etc.):

```
$ ./a.exe
```

o bien

```
$ ./a.out
```

en Linux. El resultado es:

```
$ ./a.exe
Hola mundo!
```

Por otra parte, sería deseable que nuestro ejecutable se llamara con el mismo nombre que el código fuente, tanto en cygwin como en Linux. Para ello, basta con escribir:

```
$ g++ -o holaMundo.exe holaMundo.cpp
```

y, finalmente:

```
$ ./holaMundo.exe
Hola mundo!
```

1.6. Estudiad los ejemplos

Además de `holaMundo.cpp`, el capítulo uno del curso de Cálculo Numérico comenta siete ejemplos, `euros.cpp`, `euro2.cpp`, `euro3.cpp`, `euro4.cpp`, `euroCalculadora.cpp`, `euroCalculadora2.cpp` y `newton.cpp`.

Compilad cada uno de los programas anteriores, creando en cada caso un ejecutable que se llame como el programa pero con extensión `.exe`. Ejecutad los programas y estudiad el resultado. Es importante que entendáis bien cada uno de los ejemplos para poder realizar los ejercicios de esta práctica.

1.7. Uso de gnuplot

A lo largo del curso vamos a necesitar, a menudo, representar funciones gráficamente. Para ello utilizaremos el programa gnuplot. Para invocar el programa haced doble click en el icono correspondiente (normalmente encontraréis un acceso directo si estáis en Windows) o bien, desde el terminal:

```
$ gnuplot.exe
```

el programa gnuplot ofrece una pantalla de trabajo interactiva con esta forma (para windows):

```
G N U P L O T
MS-Windows 32 bit version 3.7
patchlevel 0
last modified Thu Jan 14 19:34:53 BST 1999

Copyright(C) 1986 - 1993, 1998, 1999
Thomas Williams, Colin Kelley and many others

Type ‘help‘ to access the on-line reference manual
```

```
        The gnuplot FAQ is available from
                <http://www.uni-karlsruhe.de/~ig25/gnuplot-faq/>

        Send comments and requests for help to <info-gnuplot@dartmouth.edu>
        Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>


Terminal type set to 'windows' gnuplot>
```

si ahora tecleamos, por ejemplo:

```
 gnuplot> plot sin(x)
```

el programa produce una pantalla gráfica con la función seno.

Podéis bajar diversos manuales de gnuplot de la página web del curso. Conviene que los estudiéis a fin de familiarizaros con el programa.

En general, gnuplot nos permite representar funciones bien directamente (por ejemplo `plot sin(x)+x**3` representa la suma de la función seno y la función x^3) o bien a partir de un fichero en disco. Por ejemplo, el fichero `recta.dat`, en vuestro directorio de trabajo contiene los números:

```
0 0
1 1
2 4
3 6
4 8
5 10
...
```

La primera columna del fichero contiene las abscisas (los valores de x) y la segunda las ordenadas (los valores de $y = f(x)$ para la función $y = 2 \cdot x$. Las dos columnas están separadas por un espacio en blanco (también pueden separarse por un tabulador o por más de un espacio en blanco).

Para representar el contenido del fichero hacemos:

```
gnuplot> plot "recta.dat"
```

el resultado es una pantalla gráfica en la que se dibujan los puntos que hemos introducido en el fichero. Si tecleamos:

```
gnuplot> plot "recta.dat" with lines
```

gnuplot une los puntos con una recta. Naturalmente, si queremos representar de esta forma una función diferente de una recta (tal como un seno, por ejemplo) debemos utilizar muchos puntos en el intervalo que queramos representar para aproximar la función correctamente.

1.8. Ejercicios

Para familiarizarnos con gnuplot y practicar las nociones que hemos comentado en este capítulo, vamos a escribir una serie de programas sencillos que producen como resultado ficheros en disco adecuados para representar diversas funciones.

1. Escribid un programa que os permita representar cualquier línea recta $y = a \cdot x + b$ utilizando gnuplot. El programa debe de ser capaz de solicitar la pendiente a y ordenada en el origen b al usuario, así como el intervalo en el que se va a representar la recta (x_{min} y x_{max}) y el número de puntos que se desea calcular entre x_{min} y x_{max}. El programa debe escribir una tabla de valores $x, f(x)$ por la pantalla y también en un fichero en disco, que luego utilizaréis para representar diversas rectas con gnuplot.

   Compilad y ejecutad el programa. Utilizad gnuplot para representar las rectas.

2. Escribid un programa que resuelva ecuaciones de segundo grado $ax^2 + bx + c = 0$. El programa ha de contemplar la posibilidad de que no existan raíces reales.

   Compilad y ejecutad el programa. Resolved diferentes ecuaciones buscando casos en los que haya dos, una o ninguna raíz real.

3. Escribid un programa que permita al usuario dibujar una parábola $f(x) = ax^2 + bx + c$. El usuario debe poder elegir los parámetros a, b y c, el intervalo y el número de puntos al igual que en el ejercicio 1.

   Compilad y ejecutad el programa, representad las parábolas correspondientes a las ecuaciones resueltas en el ejercicio anterior y comprobad graficamente los resultados.

4. Deducid una variante de la fórmula de Newton que pueda aplicarse al cálculo de una raíz cúbica. Escribid un programa que implemente esa fórmula. El programa debe poderse ejecutar continuamente, ofreciendo la posibilidad de calcular otra raíz o finalizar. Si el usuario elige calcular otra raíz el programa debe solicitar el número y devolver tres números. La raíz cúbica exacta (para lo que usaréis la función `pow()`), la aproximación a la raíz calculada por el algoritmo y el número de iteraciones que han sido necesarias para alcanzar la tolerancia especificada.

   Compilad y ejecutar el programa. Calcular varias raíces cúbicas como ejemplo.

5. Escribid una segunda versión del programa anterior que permita elegir al usuario la precisión del cálculo y el número máximo de iteraciones. El bucle debe interrumpirse cuando se alcanza la precisión solicitada o se excede el número máximo de iteraciones.

Bibliografía

[1] http://cygwin.com/

[2] http://www.gnu.org/software/gcc/gcc.html

[3] http://www.gnu.org/

[4] emacs viene instalado por defecto en cualquiera de las distribuciones de Linux. Para instalarlo en Windows, consultar http://www.gnu.org/software/emacs/windows/ntemacs.html

[5] http://www.gnuplot.vt.edu/

[6] ftp://ctan.tug.org, http://www.miktex.org

Capítulo 2

Introducción al C++ (II)

2.1. Introducción

En esta práctica completamos la introducción al C++ necesaria para comenzar a programar métodos numéricos. Los ejemplos que vamos a comentar corresponden a los capítulos 2, 3 y 4 del *Curso de Cálculo Numérico en C++*. Es imprescindible que los estudiéis para aprovechar la práctica.

Los ejercicios que se proponen son variantes muy sencillas de estos ejemplos.

2.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).
2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.
3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica dos `P2.tgz`.
4. Extraed el contenido de `P2.tgz` al directorio `P2`. Podéis hacerlo con el ratón o bien, desde una ventana con la instrucción:

   ```
   tar -xzf P2.tgz
   ```
5. Examinad el contenido de la carpeta `P2`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

   ```
   cd P2
   ```

   para entrar en el directorio `P2`, seguida de:

   ```
   ls
   ```

   que nos produce un listado de los programas que vamos a examinar en la práctica.

[1] ¿Todavía no sabéis quién era Kavafis? Una pista: Su ciudad preferida es Alejandría. Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido

[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

2.3. Estudiad los ejemplos

Los ejemplos de esta práctica son:

1. `charToInt.cpp` (capítulo 2). Ilustra la relación entre el tipo `char` y el tipo `int`.
2. `sizeOf.cpp` (capítulo 2). Tamaño en memoria de los distintos tipos.
3. `alcance.cpp` (capítulo 2). Alcance de una variable.
4. `puntero.cpp` (capítulo 3). Punteros.
5. `punteroYArreglo.cpp` (capítulo 3). Relación entre punteros y arreglos.
6. `arregloDinamico.cpp` (capítulo 3). Uso de la instrucción `new` y `delete` para asignar memoria a un arreglo dinámicamente.
7. `referencia.cpp` (capítulo 3). Uso de referencias y punteros.
8. `iarreglo.cpp` (capítulo 3). Ejemplo de estructura de datos.
9. `coulomb.cpp` (capítulo 4). Declaración de una función.
10. `argumentos.cpp` (capítulo 4). Paso de argumentos a una función.
11. `punteroAFuncion.cpp` (capítulo 4). Punteros a funciones.
12. `funcToFunc.cpp` (capítulo 4). Punteros de una función a otra.

Compilad cada uno de los programas anteriores, creando en cada caso un ejecutable que se llame como el programa pero con extensión `.exe`. Ejecutad los programas y estudiad el resultado. Es importante que entendáis bien cada uno de los ejemplos para poder realizar los ejercicios de esta práctica.

2.4. Ejercicios

1. Escribid el valor entero de los siguientes caracteres: 'a','b','c','x','y','z'. ¿Hay algo que os llame la atención?
2. Escribid un programa que utilize un bucle `for` para imprimir el valor entero de las letras minúsculas del alfabeto.
3. ¿Cuál es el tamaño, en vuestro ordenador, de los siguientes tipos: `long double`, `long int`, `short int`?
4. Escribid un programa que defina un arreglo de dobles (`double darrary[]`) de dimensión 5 y lo inicialice de la siguiente manera:

   ```
   double darray[] ={1.,2.,3.,4.,5.};
   ```

   - Imprimid el contenido del arreglo usando índices.
   - Lo mismo pero usando notación de puntero.
5. Escribid una variante del programa anterior que:

   - Declare `darray` como un arreglo de `double` de dimensión 10.
   - Utilize un bucle `for` para inicializar `darray` a los siguientes valores: `darray[0]=1`, `darray[1]=2`, ... , `darray[9]=10`:

```
for (int i = 0; i < 10); i++){ // incrementa i
// inicializar arreglo
}
```

- Invierta el contenido del arreglo, de manera que: `darray[0]=10`, `darray[1]=9`, ... ,`darray[9]=1`. Para ello podéis usar un bucle `while`:

```
int i = 0;
int j = 9;

while (i < j) {
// copiar el contenido de x[i] en una variable temporal
// copiar el contenido de x[j] en x[i]
// copiar el contenido de la variable temporal en x[j]
// incrementar i
// disminuir j
}
```

6. Escribid una variante del programa anterior que haga lo mismo pero usando punteros.
7. Escribid una función que intercambie el valor de dos variables enteras. Es decir:

```
int i = 2;  // por ejemplo
int j = 7;
intercambia(i,j); // cambia i por j
cout << i << endl; // i debe valer 7
cout << j<< endl; // j debe valer 2
```

   Escribid un programa que imprima el valor de las dos variables, llame a la función `intercambia` e imprima las dos variables de nuevo. (Pista: Usar referencias a `int` en los argumentos formales de la función `intercambia`).
8. Repetid el ejercicio anterior, pero usando punteros como argumentos formales de la función `intercambia`. ¿Qué version os parece más fácil?
9. Escribid un programa que calcule el producto escalar de dos vectores de dimensión arbitraria. Para ello:
   - Escribid una función con el siguiente prototipo:

     `double productoEscalar (double v1[], double v2[], int n);`

     es decir, una función que tome dos arreglos de `double` y un `int` que no es sino la dimensión de los dos vectores. La función debe calcular y devolver el producto escalar.
   - En `main`, antes de invocar a `productoEscalar`, debéis solicitar al usuario la dimensión de los vectores cuyo producto escalar quiere calcular. Una vez obtenida esta dimensión, definid dinámicamente los arreglos `v1` y `v2` (deben ser arreglos de tipo `double` y dimensión `n`, usad la instrucción `new`).
   - Pedid al usuario que introduzca uno a uno los valores de `v1` y `v2` y almacenadlos en los vectores.
   - Calculad el producto escalar llamando a la función `productoEscalar`.
   - Finalmente, usad la instrucción `delete [] v1` (idem para `v2`) para liberar la memoria, antes de terminar el programa.

Capítulo 3

Solución de ecuaciones no lineales

3.1. Introducción

En esta práctica abordamos el estudio de los métodos numéricos que nos permiten resolver ecuaciones no lineales. El ejemplo que vamos a comentar corresponde al capítulo 5 del *Curso de Cálculo Numérico en C++*. Es imprescindible que lo estudiéis detenidamente para aprovechar la práctica.

3.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).
2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.
3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica tres `P3.tgz`. También tenéis que bajar el fichero comprimido `Util.tgz`
4. Extraer el contenido de `P3.tgz` al directorio `P3`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P3.tgz
   ```
5. Extraer el contenido de `Util.zip` al directorio `Util`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf Util.tgz
   ```
6. Examinar el contenido de la carpeta `P3`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

   ```
   cd P3
   ```

[1] Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido
[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

para entrar en el directorio `P3`, seguida de:

```
ls
```

que produce un listado de los programas que vamos a examinar en la práctica. Debéis encontrar al menos los siguientes ficheros:

```
Makefile      func.h        raices.h      xraices.cpp
func.cpp      raices.cpp    xp3.cpp
```

7. Subid un nivel y bajad a la carpeta Util (`cd ../Util`). Listad el contenido de la carpeta, que debe contener al menos los siguientes ficheros:

```
algoritmo.h  cdatos.h     error.h      specfunc.h
ccontrol.h   dvector.cpp  matutil.h    tvector.h
cdatos.cpp   dvector.h    medida.h
```

8. Volved de nuevo al directorio `P3` (`cd ../P3`).

3.3. Estudiad el ejemplo

En esta práctica, ilustramos los métodos numéricos de bisección, regula-falsi y Newton-Raphson encontrando el cero de la función $x - \cos(x)$. Hemos preparado un programa principal, `xraices.cpp` que ofrece al usuario la posibilidad de utilizar cualquiera de los tres métodos mencionados. La función a estudiar se declara en `func.h` y se define en `func.cpp`. Los algoritmos propiamente dichos se declaran en `raices.h` y se definen en `raices.cpp`. Además, utilizamos las utilidades matemáticas que se encuentran definidas en el fichero `matutil.h`, en el directorio `Util`.

En esta práctica, compilamos varios programas juntos, produciendo objetos que luego enlazamos en un sólo ejecutable. Concretamente, para crear el ejecutable `xraices.exe` se requieren los siguientes ficheros:

1. `xraices.cpp`.- Programa principal.
2. `raices.cpp`.- Algoritmos de búsqueda (bisección, regula falsi y Newton-Raphson).
3. `func.cpp`.- Contiene la función a estudiar.

Para compilar y enlazar `xraices` debemos ejecutar:

```
g++ -g -I. -I../Util xraices.cpp -c
g++ -g -I. -I../Util raices.cpp -c
g++ -g -I. -I../Util func.cpp -c
g++ -g -o xraices.exe xraices.o raices.o func.o
```

Las instrucciones del tipo

```
g++ -g -I. -I../Util fichero.cpp -c
```

compilan `fichero.cpp`, produciendo `fichero.o`. El calificador `-g` quiere decir compilar con la opción debugger (útil para trazar posibles errores de ejecución), `-I.` y `-I../Util` añade el directorio donde estamos trabajando y el directorio `Util` (que debe estar al mismo nivel de nuestro directorio de trabajo) a la ruta del compilador, de tal manera que encuentre todas las cabeceras que necesita (`error.h` y `matutil.h` están en Util, las otras en el directorio de trabajo).

Recordad que la estructura de directorios que debéis tener es la siguiente:

```
-----------------/home/calnum-------------
-----------------------|
                -----Kavafis-----
                |   |   |   |   |
                P1  P2  P3 Util ..
                -----   |    |
           --------------    ---------------
           raices.h                  error.h
           raices.cpp                matutil.h
           ...
```

La instrucción de la forma:

```
g++ -g -o fichero.exe fichero1.o fichero2.o fichero3.o
```

crea el fichero final que ejecutamos `fichero.exe`. Las operaciones de compilar enlazar y crear el ejecutable se pueden realizar simultaneamente con un solo comando de la siguiente forma:

```
g++ -g  -I. -I../Util -o fichero.exe fichero1.cpp fichero2.cpp fichero3.cpp
```

Una última opción es teclear `make` que actualiza los ejecutables de todos aquellos ficheros fuente que hemos modificado.

3.4. Ejercicios

1. Estudiad la función:
$$e^{-x} - 2x^2 \tag{3.1}$$
Representadla gráficamente, acotad todos sus ceros y determinadlos con una precisión relativa de 10^{-6} por los tres métodos que hemos estudiado. Para ello, debéis editar los programas `func.h` y `func.cpp` para declarar y programar la nueva función 3.1 asícomo su derivada, que necesitaréis para el método de Newton-Raphson. Comentad vuestros resultados.
Para compilar los programas, una vez introducidas vuestras modificaciones seguid las instrucciones del apartado anterior.

2. Lo mismo para la función:
$$x^3 + 3x^2 - 1$$

3. Estudiad el cero de la función $f(x) = \arctan(x)$ con una precisión de 10^{-6} mediante los tres métodos en el intervalo $[-5, 5]$ y comparad el número de iteraciones. Comentad el resultado.

4. Para evitar divergencias en el método de Newton-Raphson se suele combinar con el método de bisección. Si al realizar una nueva iteración con el método de Newton-Raphson, la nueva aproximación x_n está fuera del intervalo inicial $[a, b]$, es decir, $x_n < a$ ó $x_n > b$ entonces, realizamos una bisección.
Escribid un programa que combine el método de Newton con el de bisección para encontrar el cero de la función del ejercicio anterior

5. El método de Steffensen es un método iterativo similar al método de Newton-Raphson pero no necesita el cáculo de la derivada de la función. Cada nueva aproximación de la raíz, viene dada en función de la anterior según la siguiente expresión:
$$x_{n+1} = x_n - \frac{[f(x_n)]^2}{f(x_n + f(x_n)) - f(x_n)}$$
Programar dicho método y aplicarlo a los problemas 1 y 2. Comparad con el método de Newton-Raphson.

Capítulo 4

Clases de vectores y matrices

4.1. Introducción

Esta práctica está dedicada a familiarizarnos con el concepto de clase en C++, en particular con las clases de vectores y matrices que vamos a utilizar para representar estos objetos en temas sucesivos. Es imprescindible que estudiéis detenidamente el capítulo 6 del *Curso de Cálculo Numérico en C++* para aprovechar la práctica.

4.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).

2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.

3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica cuatro `P4.tgz`. También tenéis que bajar el fichero comprimido `Util.tgz`.

4. Extraed el contenido de `P4.tgz` al directorio `P4`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P4.tgz
   ```

5. Extraed el contenido de `Util.tgz` al directorio `Util`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf Util.tgz
   ```

6. Examinad el contenido de la carpeta `P4`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

[1] Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido.
[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

```
cd P4
```

para entrar en el directorio `P4`, seguida de:

```
ls
```

que produce un listado de los programas que vamos a examinar en la práctica. Debéis encontrar al menos los siguientes ficheros:

```
Makefile
agenda.cpp        matriz.cpp        xagenda.cpp       xvector.cpp
agenda.h          matriz.h          xcomplejo.cpp
complejo.cpp      vector.cpp        xmatriz.cpp
complejo.h        vector.h          xp4.cpp
```

4.3. Estudiad el ejemplo

En esta práctica, estudiamos el concepto de clase en C++ y en particular las clases de vectores y matrices. Comentamos con detalle cuatro ejemplos:

1. Clase `Agenda`.- La definición de la clase está en el fichero `agenda.h` y su implementación en `agenda.cpp`. El programa `xagenda.cpp` muestra un sencillo ejemplo de uso.

2. Clase `Complejo`.- La definición de la clase está en el fichero `complejo.h` y su implementación en `complejo.cpp`. El programa `xcomplejo.cpp` muestra un sencillo ejemplo de uso.

3. Clase `RVector`.- La definición de la clase está en el fichero `vector.h` y su implementación en `vector.cpp`. El programa `xvector.cpp` muestra un sencillo ejemplo de uso.

4. Clase `RMatriz`.- La definición de la clase está en el fichero `matriz.h` y su implementación en `matriz.cpp`. El programa `xmatriz.cpp` muestra un sencillo ejemplo de uso.

En esta práctica, compilamos varios programas juntos, produciendo objetos que luego enlazamos en un sólo ejecutable. Concretamente:

1. `xagenda.exe` requiere `xagenda.cpp` (programa principal), `agenda.cpp`.

2. `xcomplejo.exe` requiere `xcomplejo.cpp`, `complejo.cpp`.

3. `xvector.exe` requiere `xvector.cpp`, `vector.cpp`.

4. `xmatriz.exe` requiere `xmatriz.cpp`, `matriz.cpp`.

Para compilar y enlazar `xagenda.exe` debemos ejecutar:

```
g++ -g -I. -I../Util -c xagenda.cpp
g++ -g -I. -I../Util -c agenda.cpp
g++ -g -o xagenda.exe agenda.o xagenda.o
```

O bien, teclear `make xagenda`. Lo mismo para los demás programas, sin más que cambiar `xagenda` por `xcomplejo`, `xvector` y `xmatriz`, respectivamente.

Recordad que la estructura de directorios que debéis tener es la siguiente:

```
-----------------/home/calnum-------------
-----------------------|
               -----Kavafis-----
               |   |   |   |   |
               P1  P2  P3  P4 Util ..
               -----       |   |
           ------------------   ---------------
           agenda.h                   error.h
           complejo.h                matutil.h
           vector.h
           ...
```

4.4. Ejercicios

1. Hemos preparado un programa, `xp4.cpp` que podéis compilar sin más que escribir `make xp4`. En su versión actual el programa no hace nada, no es más que una cáscara prefabricada para que incluyáis vuestro código.

   Modificad el programa `xp4.cpp` para realizar las siguientes operaciones con matrices y vectores:

$$A = B \cdot B^T + C \cdot D$$
$$A+ = B$$
$$C* = \lambda$$
$$\mu \cdot (A + B) - (B + A) \cdot \mu$$
$$V_1+ = V2$$
$$\lambda \cdot (V_1 + V_2) - (V_2 + V_1) \cdot \lambda$$

   Donde A, B, C y D son las siguientes matrices:

$$B = \begin{pmatrix} -2 & -2 & -9 \\ 1 & 5 & 1 \\ -3 & 0 & -1 \end{pmatrix}$$

$$C = \begin{pmatrix} 2 & 3 \\ -1 & -6 \\ 5 & 4 \end{pmatrix}$$

$$D = \begin{pmatrix} 1 & 2 & -7 \\ -3 & 5 & -4 \end{pmatrix}$$

   Los escalares $\lambda = 1{,}5$, $\mu = 3{,}7$ son números reales y los vectores valen:

$$V_1 = (1, -5, 2, 4) \quad V_2 = (4, 0, -3, 3)$$

2. Escribid las siguientes funciones para ampliar la clase `RVector`:

   a) La función `norma` para calcular la norma euclídea del vector:

```
double norma(const RVector& v );
```

   b) Las funciones `maximo` y `minimo`, que devuelven el índice del elemento máximo (mínimo):

```
int maximo(const RVector&);
int minimo(const RVector&);
```

c) La función `intercambia`, que intercambia el elemento `i` y el elemento `j` del vector:

```
RVector& intercambia(int i, int j);
```

Algunas de estas funciones deben escribirse como internas a la clase, mientras que otras son externas. Podéis ver cuál es cuál sin más que estudiar sus declaraciones. Razonad en cada caso por qué cada una de las funciones anteriores es externa o interna. Modificad apropiadamente el fichero `vector.h` para incluir la declaración de todas estas funciones y el fichero `vector.cpp` para incluir su definición. Modificad el programa `xvector` para comprobar que todas funcionan correctamente.

3. Escribid las siguientes funciones para ampliar la clase `RMatriz`:

a) La función `Traza`, que devuelve la traza de la matriz:

```
double Traza(const RMatriz& m);
```

b) La función `proyectaFila`, que crea un vector y copia en éste una fila escogida (en el argumento a la función) de la matriz:

```
RVector proyectaFila(const RMatriz& m, int fila);
```

c) La función `proyectaColumna`, que crea un vector y copia en éste una columna escogida (en el argumento a la función) de la matriz:

```
RVector proyectaColumna(const RMatriz& m, int columna);
```

d) La función `intercambiaFila`, que intercambia la fila `i` y la fila `j` de la matriz:

```
RMatriz& intercambiaFila(int i,int j);
```

e) La función `intercambiaColumna`, que intercambia la columna `i` y la columna `j` de la matriz:

```
RMatriz& intercambiaColumna(int i,int j);
```

Como en el caso anterior, razonar qué funciones son internas y cuáles externas a la clase. Modificad apropiadamente el fichero `matriz.h` para incluir la declaración de todas estas funciones y el fichero `matriz.cpp` para incluir su definición. Notad que las funciones `proyectaFila` y `proyectaColumna` requieren la definición de la clase `RVector` y por lo tanto tenéis que incluirla como cabecera en `matriz.h`. Modificad el programa `xmatriz` para comprobar que todas vuestras nuevas funciones funcionan (valga la cacofonía) correctamente.

Capítulo 5

Problemas de álgebra lineal y diagonalización de matrices

5.1. Introducción

Dedicamos esta práctica a familiarizarnos con los algoritmos estudiados en los capítulos 7 (problemas de álgebra lineal) y 8 del *Curso de Cálculo Numérico en C++*.

5.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).

2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.

3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica cinco `P5.tgz`. También tenéis que bajar del fichero comprimido `Util.tgz`. *Es muy importante* que os bajéis en cada práctica `Util.tgz`, ya que los ficheros de utilidades que guardamos en dicho directorio se actualizan periódicamente.

4. Extraed el contenido de `P5.tgz` al directorio `P5`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P5.tgz
   ```

5. Extraed el contenido de `Util.tgz` al directorio `Util`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf Util.tgz
   ```

[1] Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido.
[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

6. Examinad el contenido de la carpeta `P5`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

   ```
   cd P5
   ```

   para entrar en el directorio `P5`, seguida de:

   ```
   ls
   ```

   que produce un listado de los programas que vamos a examinar en la práctica. Debéis encontrar al menos los siguientes ficheros:

   ```
   Makefile
   lineal.cpp   permutacion.cpp  vector.cpp    xdiag.cpp         xsmatriz.cpp
   linalg.cpp   rdiagonal.cpp    xalin.cpp     xp5.cpp
   matriz.cpp   smatriz.cpp      xalineal.cpp  xpermutacion.cpp

   alineal.h    matriz.h         rdiagonal.h   vector.h
   linalg.h     permutacion.h    smatriz.h

   m1d5.dat     b1d5.dat         m6d3.dat
   ```

7. Para esta práctica hemos ampliado las clases de matrices (`matriz.h`, `matriz.cpp`) y vectores (`vector.h`, `vector.cpp`), añadiendo además nuevas funciones que involucran operaciones entre ambos tipos (en `linalg.h`, `linalg.cpp`) tal como se explica en el capítulo 7 del curso de Cálculo Numérico. Añadimos además dos nuevas clases de datos (matrices simétricas (`smatriz.h`, `smatriz.cpp`) y permutaciones (`permutacion.h`,`permutacion.cpp`). Por último encontramos un nuevo tipo de clases, llamadas clases de algoritmos (descritas en el capítulo 7 del curso) que se ocupan, respectivamente, de resolver problemas de álgebra lineal (`alineal.h`,`alineal.cpp`) y de diagonalizar matrices reales y simétricas (`rdiagonal.h`, `rdiagonal.cpp`).

   Los ficheros `m1d5.dat`, `b1d5.dat` y `m6d3.dat` son ficheros de datos que contienen matrices y vectores útiles para la práctica.

5.3. Estudiad el ejemplo

En esta práctica, estudiamos la resolución de problemas de álgebra lineal (diagonalización LU, solución de un sistema lineal de ecuaciones a partir de la matriz factorizada, inversión de una matriz, cálculo del determinante) así como la diagonalización de matrices reales y simétricas.

Hay dos programas de ejemplo.

1. El programa `xalin.cpp` que resuelve un problema completo de álgebra lineal para una matriz $n \times n$ y un vector de n términos independientes.
2. El programa `xdiag.cpp` que diagonaliza una matriz real y simétrica.

Para crear los correspondientes ejecutables tenéis que hacer:

1. `xalin.exe` requiere `xalin.cpp` (programa principal), `permutacion.cpp`, `vector.cpp`, `matriz.cpp`, `linalg.cpp`, `alineal.cpp`.
2. `xdiag.exe` requiere `xdiag.cpp` (programa principal), `rdiagonal.cpp`, `smatriz.cpp`, `matriz.cpp`, `permutacion.cpp`, `vector.cpp`, `linalg.cpp`, `alineal.cpp`.

Para compilar y enlazar cada uno de estos ejecutables hay que compilar cada uno de los ficheros anteriores por separado

```
g++ -g -I. -I../Util -c fichero.cpp (fichero = permutacion,vector,matriz,linalg....)
```

Y luego enlazarlos, escribiendo la serie de objetos que necesitamos para cada ejecutable, por ejemplo, para `xpermutacion.exe`:

```
g++ -g -o xpermutacion.exe xpermutacion.o permutacion.o vector.o matriz.o linalg.o
```

y análogamente para el resto. También podemos utilizar la `Makefile`. Por ejemplo, basta con teclear `make xalin` para compilar y enlazar el programa `xalin.exe` y análogamente para el resto.

5.4. Ejercicios

1. Sea el sistema de ecuaciones:

$$\begin{array}{rcrcrcr} 2x_1 & - & x_2 & + & x_3 & = & -1 \\ 3x_1 & + & 3x_2 & + & 9x_3 & = & 0 \\ 3x_1 & + & 3x_2 & + & 5x_3 & = & 4 \end{array}$$

   a) Obtener la descomposición LU de la matriz a mano.

   b) Resolver el sistema mediante dicha descomposición a mano.

   c) Escribid la matriz del sistema en un fichero en disco, así como el vector de coeficientes. Modificad en lo necesario el programa `xalin.cpp` para que os permita resolver el sistema. Comprobad que el resultado es correcto.

2. Sean los sistemas de ecuaciones:

$$\begin{array}{rcrcrcrcr} -x_1 & + & x_2 & & & - & 3x_4 & = & 4 \\ x_1 & & & + & 3x_3 & + & x_4 & = & 0 \\ & & x_2 & - & x_3 & - & x_4 & = & 3 \\ 3x_1 & & & + & x_3 & + & 2x_4 & = & 1 \end{array} \qquad \begin{array}{rcrcrcrcr} 6x_1 & - & 2x_2 & + & 2x_3 & + & 4x_4 & = & 0 \\ 12x_1 & - & 8x_2 & + & 4x_3 & + & 10x_4 & = & -10 \\ 3x_1 & - & 13x_2 & + & 3x_3 & + & 3x_4 & = & -39 \\ -6x_1 & + & 4x_2 & + & 2x_3 & - & 18x_4 & = & -16 \end{array}$$

   a) Resolvedlos mediante el programa `xalin.cpp`.

   b) Calculad las matrices inversas y los determinantes de cada una de las matrices de los sistemas anteriores. Comprobad que los resultados son correctos.

   c) Modificad el programa `xalin.cpp` para poder resolver el sistema a partir del conocimiento de la inversa, es decir, vuestro programa debe ser capaz de calcular:

      1) La inversa de la matriz del sistema A^{-1}.

      2) La solución del sistema de ecuaciones a partir de la ecuación: $\mathbf{x} = A^{-1}\ \mathbf{w}$, donde $\mathbf{w}$ es el vector de términos independientes y $\mathbf{x}$ el vector de incógnitas

3. Considerad la matriz real y simétrica:

$$A = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 3 & -2 \\ 1 & -2 & -1 \end{pmatrix}$$

   a) Realizad una primera iteración del algoritmo de Jacobi a mano.

   b) Modificad el programa `xdiag.cpp` para que calcule los valores propios de la matriz A y sus autovectores.

c) Asociad cada valor propio a su autovector correspondiente comprobando que:

$$A\mathbf{x_i} = \lambda_\mathbf{i}\mathbf{x_i}$$

d) Demostrad numéricamente que P es ortogonal, $P^T = P^{-1}$.

e) Demostrad numéricamente que: $D = P^T AP$, donde D es la matriz diagonalizada.

4. Repetid el ejercicio anterior, salvo el primer apartado, para la matriz:

$$A = \begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix}$$

5. Para el caso de matrices reales, simétricas y definidas positivas, existe una factorización más sencilla que la LU, llamada factorización de Choleski, en la que $A = L \cdot L^T$. Uno de los atractivos del algoritmo es que no es necesario efectuar permutaciones.

   a) Deducid la forma de la matriz L, de manera análoga a como hemos deducido la factorización LU.

   b) Deducid la solución de un sistema lineal (mediante sustitución progresiva y regresiva) de manera análoga a como hemos deducido la solución de un sistema lineal para la factorización LU.

   c) Escribid una función, `int choleski(RMatriz& lu)` que toma una matriz en el argumento y devuelva la matriz factorizada por choleski al final de la ejecución. Usad la función `int descomponeLU(RMatriz& lu,Permutacion& p)` como ejemplo (recordad que no es necesario permutar).

   d) Escribid una función, `int resuelveCholeski(const RMatriz& lu, RVector& x)` que tome la matriz descompuesta por Choleski en el argumento así como el vector de términos independientes y devuelva el vector de soluciones al final de la ejecución. Usad la función `int resuelveLU(const RMatriz& lu, const Permutacion& p, RVector& x)` como ejemplo (recordad que no es necesario permutar).

   e) Utilizad estas funciones para resolver el sistema lineal asociado a las matrices:

$$A = \begin{pmatrix} 4 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & -1 & 4 \end{pmatrix}$$

$$B = \begin{pmatrix} 60 & 30 & 20 \\ 30 & 20 & 15 \\ 20 & 15 & 12 \end{pmatrix}$$

   y los vectores de términos independientes:

$$w_1 = (3, 2, 2, 2, 3) \qquad w_2 = (40, 5, -2)$$

Capítulo 6

Interpolación

6.1. Introducción

Dedicamos esta práctica a familiarizarnos con problemas de interpolación, capítulo 9 del *Curso de Cálculo Numérico en C++*.

6.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).

2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.

3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica seis `P6.tgz`. También tenéis que bajar el fichero comprimido `Util.tgz`. *Es muy importante* que os bajéis en cada práctica `Util.tgz`, ya que los ficheros de utilidades que guardamos en dicho directorio se actualizan periódicamente.

4. Extraed el contenido de `P6.tgz` al directorio `P6`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P6.tgz
   ```

5. Extraed el contenido de `Util.tgz` al directorio `Util`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf Util.tgz
   ```

6. Examinad el contenido de la carpeta `P6`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

[1] Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido
[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

```
cd P6
```

para entrar en el directorio `P6`, seguida de:

```
ls
```

que produce un listado de los programas que vamos a examinar en la práctica. Debéis encontrar al menos los siguientes ficheros:

```
              interpol.h    xfundat.cpp     xinterpol3.cpp
Makefile      xcdatos.cpp   xinterpol.cpp   xp6.cpp
interpol.cpp  xdvector.cpp  xinterpol2.cpp
```

7. El algoritmo de interpolación está en los ficheros `interpol.cpp` y `interpol.h`. Los programas `xdvector.cpp`, `xfundat.cpp`, `xinterpol.cpp`, `xinterpol2.cpp` y `xinterpol3.cpp` son ejemplos que utilizaremos a lo largo de la práctica.

6.3. Estudiad el ejemplo

En esta práctica, estudiamos problemas de interpolación utilizando polinomios.
Para ello vamos a utilizar tres programas:

1. El programa `xfundat.cpp` que genera un conjunto de datos de acuerdo a una o varias funciones definidas por el usuario. En el ejemplo que tenéis, `xfundat.cpp` os permite elegir entre dos funciones de ejemplo, $\sin^3(x)$ y $(1+x^2)^{-1}$. También os permite escoger el conjunto de nodos con espaciado uniforme o utilizando nodos de Chebyshev.

2. El programa `xinterpol.cpp` que lee el fichero de datos creado por `xfundat.cpp` e interpola un polinomio cuyo grado, nodos inicial y final puede ser elegido por el usuario. Además de la salida por pantalla, el programa genera un fichero de datos que puede usarse para representación gráfica con gnuplot.

3. El programa `xinterpol2.cpp` que lee el fichero de datos creado por `xfundat.cpp` y calcula la aproximación para un punto determinado, proporcionado por el usuario, comenzando por un polinomio de grado uno con con los nodos que encierran el punto, y calculando la corrección que la adición de un nuevo nodo a la derecha, supone en cada paso.

4. El programa `xinterpol3.cpp` análogo a `xinterpol2.cpp` pero permite al usuario elegir los nodos inferior y superior entre los que se calcula el polinomio interpolador.

Para crear los correspondientes ejecutables basta con que hagáis `make xfundat`, `make xinterpol` y `make xinterpol2`. La instrucción `make xfundat` es equivalente a:

```
g++ -g -I. -I../Util xfundat.cpp -c -o xfundat.o
g++ -g -o xfundat.exe xfundat.o
```

La primera instrucción compila `xfundat.cpp`, incluyendo las cabeceras del directorio en el que estamos y del directorio `Util` y crea `xfundat.o`. La segunda instrucción crea `xfundat.exe` a partir del fichero objeto `xfundat.o`.

La instrucción `make xinterpol` es equivalente a:

```
g++ -g -I. -I../Util xinterpol.cpp -c -o xinterpol.o
g++ -g -I. -I../Util interpol.cpp -c -o interpol.o
g++ -g -o xinterpol.exe xinterpol.o interpol.o
```

Es decir, compilamos `xinterpol.cpp` y `interpol.cpp` y los enlazamos para formar `xinterpol.exe`.

Idénticos comentarios para `xinterpol2.cpp` y `xinterpol3.cpp`

6.4. Ejemplo

Queremos estudiar la función $\sin^3(x)$ en el intervalo $[-\pi, \pi]$. Para ello, vamos a representarla en primer lugar, utilizando gnuplot:

```
gnuplot> plot[-3:3] sin(x)**3
```

A continuación vamos a generar un conjunto de 15 nodos igualmente espaciados utilizando el programa `xfundat.cpp`. Para ello:

```
$ ./xfundat.exe
 Cuántos puntos quiere generar? (max 25 ) 15
 Qué función desea? (1 = sin**3(x), 2 = runge, 3 = F)1
 Qué espaciado desea? (1 = uniforme, 2 = Chebyshev)1
 Introduzca intervalo [a,b]
a? -3
b? 3
```

El programa escribe los datos por la pantalla así como un fichero cuyo nombre depende de la función y espaciado que hayamos escogido. El nombre del fichero se decide en el fragmento de código:

```
    if (ifun == 1){
    if (iespa ==1){
        d = ispace(sin3, npoints, a, b);
        fdatos = "sin3.uni";
    }
    else{
        d = cheb(sin3, npoints, a, b);
        fdatos = "sin3.che";
    }
    }
    else if (ifun == 2){
    if (iespa ==1){
        d = ispace(runge, npoints, a, b);
        fdatos = "runge.uni";
    }
    else{
        d = cheb(runge, npoints, a, b);
        fdatos = "runge.che";
    }
    }
    else if (ifun == 3){
    if (iespa ==1){
        d = ispace(ff, npoints, a, b);
        fdatos = "ff.uni";
    }
    else{
        d = cheb(ff, npoints, a, b);
        fdatos = "ff.che";
    }
    }
```

donde `ifun` e `iespa` son dos variables enteras que utilizamos para decidir el tipo de función y el tipo de espaciado. Dependiendo del tipo de espaciado llamamos a la función `ispace` (nodos con espaciado

uniforme) o bien `cheb` (nodos con espaciado de Chebyshev). Estas funciones se llaman pasando un puntero a la función `sin3` ($\sin^3(x)$) o `runge` ($(1+x^2)^{-1}$) dependiendo de `ifun`.

Los ficheros de datos posibles son:

1. `sin3.uni`.- Función $\sin^3(x)$ con espaciado uniforme.
2. `sin3.che`.- Función $\sin^3(x)$ con espaciado de Chebyshev.
3. `runge.uni`.- Función $(1+x^2)^{-1}$ con espaciado uniforme.
4. `runge.che`.- Función $(1+x^2)^{-1}$ con espaciado de Chebyshev.
5. `ff.uni`.- Función de usuario con espaciado uniforme.
6. `ff.che`.- Función de usuario con espaciado de Chebyshev.

Así pues, en el ejemplo que estamos desarrollando, tendremos un fichero en disco que se llama `sin3.uni` con los datos generados por la función. Podemos representar dichos datos con gnuplot, superponiéndolos a la función que hemos representado anteriormente.

```
gnuplot> replot "sin3.uni"
```

A continuación vamos a utilizar el programa `xinterpol` para interpolar varios polinomios de diferentes grados a esos puntos. Empezamos por el polinomio de mayor grado posible:

```
$ ./xinterpol.exe
fichero de datos a interpolar? sin3.uni
 Qué función desea? (1 = sin**3(x), 2 = runge, 3 = F)1
 Este programa interpola un polinomio a un conjunto de 15 nodos
 obtenidos a partir de la función representada en el fichero sin3.uni
 El polinomio se interpola entre los nodos [imin, imax], donde imin >= 0
 y imax <= 14
imin? 0
imax? 14
```

El programa interpola un polinomio y escribe una tabla por la pantalla (y en un fichero en disco). La primera columna de la tabla son puntos escogidos para que no coincidan con los nodos. La segunda es el valor real de la función en esos puntos, la tercera el valor del polinomio y la cuarta la diferencia en valor absoluto entre ambos. Notad que la aproximación es excelente en la parte central de la tabla y se deteriora hacia los bordes. Este es un efecto típico cuando interpolamos polinomios de grado alto, ya que los términos $(x - x_i)$ se hacen muy grandes en los extremos del polinomio. Para visualizar gráficamente la aproximación vamos a dibujar el conjunto de puntos $(x, p(x))$ de la tabla, superponiéndolo a las funciones anteriores con gnuplot. El programa `xinterpol` genera un fichero de salida que se llama como el fichero de entrada con una "p" delante, por lo tanto "psin3.uni" en este ejemplo.

```
gnuplot>  replot "psin3.uni" with lines
```

A continuación ejecutad el programa `xinterpol2` para estudiar las aproximaciones con polinomios de grado creciente.

6.5. Ejercicios

1. Usando el programa `xfundat.cpp` generad un conjunto de 10 nodos igualmente espaciados, correspondientes a la función de Runge $(1+x^2)^{-1}$ en el intervalo $[-5, 5]$. A continuación interpolad dichos nodos mediante un polinomio de grado 9. Representad el resultado con gnuplot,

superponiendo el valor real de la función y el resultado de la interpolación. Repetid el ejercicio generando 15 nodos y una vez más con 20 nodos. ¿Qué obtenéis? ¿Qué explicación podéis darle a ese resultado?

2. Repetid el ejercicio utilizando 10,15 y 20 nodos de Chebyshev. ¿Qué obtenéis? ¿Cuál es vuestra interpretación?

3. Generad un conjunto de 10 nodos igualmente espaciados de acuerdo a la función $f(x) = \arctan(x)$. Encontrad un polinomio de grado 9 que interpole $f(x)$ en el intervalo $[-1, 1]$. Representad gráficamente y comentar.

4. Para el ejemplo anterior, suponer que deseamos conocer el valor aproximado de la función en el punto $-0{,}4$ con una precisión mejor que 10^{-4}. ¿Cuál es el polinomio de grado mínimo que lo consigue?. Si no conociérais la forma explícita de la función y tuviérais que guiaros por la estimación del error a partir de las correcciones que aportan polinomios de grado consecutivo. ¿Cuando detendrías el proceso de añadir polinomios? Realizad una estimación del valor de la función en $x = 1{,}5$. Comentad el resultado. Utilizad los programas `xinterpol2` y `xinterpol3`.

5. Generad 10 nodos igualmente espaciados para la función $f(x) = xe^x$. Encontrad un polinomio de grado 9 que interpole $f(x)$ en el intervalo $[0, 2]$. Representad gráficamente y comentad el resultado.

6. Realizad una estimación del valor aproximado de la función anterior en el punto 0,3 ¿Qué precisión puede obtenerse? ¿Cuál es el polinomio de menor grado que la obtiene? Si no conociérais la forma explícita de la función y tuviérais que guiaros por la estimación del error a partir de las correcciones que aportan polinomios de grado consecutivo. ¿Cuando detendrías el proceso de añadir polinomios? Realizad una estimación del valor de la función en $x = 3$. Comentad el resultado. Utilizad los programas `xinterpol2` y `xinterpol3`.

7. Programad una función que calcule un polinomio interpolador utilizando la fórmula de Lagrange. El prototipo de la función puede ser:

   ```
   double lagrange(CDatos& nodos, double xp);
   ```

   donde `nodos` es una variable de tipo `CDatos` que contiene el conjunto de nodos y `x` es el punto donde queremos conocer el valor aproximado de la función. La función debe interpolar un polinomio de grado $n - 1$ a los n nodos de `CDatos` utilizando la fórmula de Lagrange para un polinomio de orden n que interpola $n + 1$ datos:

   $$p(x) = y_0 l_0(x) + y_1 l_1(x) + y_2 l_2(x) + ... + y_n l_n(x) = \sum_{k=0}^{n} y_k l_k(x) \tag{6.1}$$

   donde, las funciones $l_i(x)$ (llamadas funciones cardinales) se escriben como:

   $$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{6.2}$$

Utilizar vuestra función para interpolar 10 nodos a las funciones $\sin^3(x)$, $\arctan(x)$ y xe^x en los intervalos definidos en los apartados anteriores. Escribid un programa que genere una tabla de cincuenta puntos con la forma $x, p(x)$, donde x son puntos que no coinciden con los nodos y $p(x)$ es la aproximación a la función obtenida mediante el polinomio interpolador que os proporciona vuestra función. Escribid la tabla en un fichero y utilizadla para representar el resultado mediante gnuplot.

Capítulo 7

Diferenciación e integración numérica

7.1. Introducción

Dedicamos esta práctica a familiarizarnos con problemas de diferenciación e integración numérica, capítulo 10 del *Curso de Cálculo Numérico en C++*.

7.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).

2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.

3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica siete `P7.tgz`. También tenéis que bajar el fichero comprimido `Util.tgz`. *Es muy importante* que os bajéis en cada práctica `Util.tgz`, ya que los ficheros de utilidades que guardamos en dicho directorio se actualizan periódicamente.

4. Extraed el contenido de `P7.tgz` al directorio `P7`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P7.tgz
   ```

5. Extraed el contenido de `Util.tgz` al directorio `Util`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf Util.tgz
   ```

[1] Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido.
[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato

6. Examinad el contenido de la carpeta `P7`. Podéis hacerlo desde el entorno gráfico, o desde el terminal de cygwin o Linux, con la instrucción:

   ```
   cd P7
   ```

   para entrar en el directorio `P7`, seguida de:

   ```
   ls
   ```

   que produce un listado de los programas que vamos a examinar en la práctica. Debéis encontrar al menos los siguientes ficheros:

   ```
   Makefile    derint.cpp  derint.h  xderint.cpp  xp7.cpp
   ```

7. El algoritmo de derivación e integración está en los ficheros `derint.cpp` y `derint.h`. El programa `xderint.cpp`, es un ejemplo que utilizaremos a lo largo de la práctica.

7.3. Estudiad el ejemplo

En esta práctica, estudiamos problemas de derivación e interpolación numérica utilizando el método de la extrapolación de Richardson.

Para ello vamos a utilizar el programa `xderint.cpp`, que ofrece al usuario la oportunidad de escoger entre varias funciones en él definidas $sin^2(x) - cos(x)$ y xe^{x-10}, o de definir una nueva. Una vez dado este paso, el programa integra numéricamente, entre los límites que el usuario determine, y/o deriva numéricamente, en el punto escogido, dicha función, utilizando las extrapolaciones de Richardson hasta el orden determinado por el usuario. El programa también devuelve el vector correspondiente a la diagonal del tablero de Richardson calculada, con las sucesivas aproximaciones a la integral (derivada) realizada.

Los resultados se devuelven en una variable de tipo `Medida`, que contiene el resultado de la operación y el error correspondiente.

Para crear los correspondientes ejecutables basta con que hagáis `make xderint`. La instrucción `make xderint` es equivalente a:

```
g++ -g -I. -I../Util derint.cpp -c -o derint.o
g++ -g -I. -I../Util xderint.cpp -c -o xderint.o
g++ -g -o xderint.exe xderint.o derint.o
```

La primera instrucción compila `xderint.cpp`, incluyendo las cabeceras del directorio en el que estamos y del directorio `Util` y crea `xderint.o`. La segunda instrucción crea `xderint.exe` a partir del fichero objeto `xderint.o`.

7.4. Ejemplo

Queremos estudiar la función $sin^2(x) - cos(x)$ en el intervalo $[-3, 3]$. Para ello, vamos a representarla en primer lugar, utilizando gnuplot:

```
gnuplot> plot[-3:3] sin(x)**2 - cos(x)
```

A continuación vamos a integrar la función utilizando el programa `xderint.cpp`. Para ello:

```
$ ./xderint.exe
 Qué función desea? (1 = sin(x)*sin(x) - cos(x), 2 = exp(x - 10), 3 = F) 1
 Orden de la extrapolación de Richardson: 6
 Quieres integrar o derivar la función escogida? i
 Límite inferior: -3
 Límite superior: 3
```

El programa escribe el resultado de la integral por la pantalla, así como el error cometido para el orden de Richardson escogido. También escribe el tablero de Richardson.

Ahora vamos derivar la función, calculándola por ejemplo en el punto x = 4, utilizando este mismo programa, para lo cual:

```
$ ./xderint.exe
 Qué función desea? (1 = sin(x)*sin(x) - cos(x), 2 = exp(x - 10), 3 = F) 1
 Orden de la extrapolación de Richardson: 4
 Quieres integrar o derivar la función escogida? d
 Punto en el que quieres el cálculo de la derivada 4
```

Al igual que antes, el programa escribe el resultado por pantalla, así como la tabla de Richardson asociada.

7.5. Ejercicios

1. Estudiad la integral y la derivada de la función $x * exp(x - 10)$ del siguiente modo:

   a) Integrad numéricamente la función, entre los límites [0,4], realizando a mano la extrapolación de Richardson. ¿Hasta qué orden debéis llegar en dicha extrapolación para obtener un error de la milésima en el valor de la integral?

   b) Comparad el resultado con el obtenido utilizando el programa `xderint.cpp`.

   c) Comparad con el resultado exacto.

   d) Derivad numéricamente la misma función, desarrollando también a mano la extrapolación de Richardson y calculad el valor de la derivada en el punto x = 0.5, haciendo un seguimiento de la evolución del error y deteniéndoos cuando éste sea menor que la milésima parte del resultado.

   e) Verificad el resultado obtenido con el programa y analíticamente.

2. Dibujad la función $\sqrt(1+x^4)$ con gnuplot en el intervalo [-10,10] y observad su comportamiento. Calculad el área encerrada bajo la curva utilizando `xderint.cpp` en el intervalo citado, utilizando la extrapolación de Richardson a orden 5. A continuación calculad la misma área, con Richardson al mismo orden, pero utilizando solo la mitad del intervalo anterior y multiplicándola por 2. Comparad los dos resultados obtenidos. ¿Con qué procedimiento se obtiene un error menor? ¿A qué creéis que es debido? ¿Cuánto debemos aumentar el orden en la extrapolación de Richardson del método menos eficiente para lograr alcanzar el error del más eficiente? Razonad vuestras respuestas. Calculad el valor de la derivada de la función $\sqrt(1+x^4)$ en el punto x = 1, a orden 4.

3. Programad una función que calcule una integral por el método de la regla trapezoidal dividida con n intervalos. El prototipo de la función puede ser:

   ```
   double trap(func F, double a, double b, int n);
   ```

   donde `F` es la función a integrar, `a y b` son los límites de integración y `n` es el número de subintervalos en el que queréis dividir el intervalo [a,b]. La función devuelve el valor de la integral. Comprobad vuestro programa con cualquier ejemplo que podáis resolver analíticamente.

4. Programad una función que calcule una integral utilizando la regla de Simpson repetida. Recordad que el número de subintervalos debe ser par. El prototipo de la función puede ser:

   ```
   double simp(func F, double a, double b, int n);
   ```

donde `F` es la función a integrar, `a y b` son los límites de integración y `n` es el número de subintervalos en el que queréis dividir el intervalo [a,b]. La función devuelve el valor de la integral. Cread un programa que verifique el correcto funcionamiento de las nuevas funciones comparando los resultados obtenidos con ambas, y establecer conclusiones sobre la bondad de los métodos. En particular, comprobad que el método de Simpson arroja la misma precisión que la regla trapezoidal con una extrapolación de Richardson (utilizad `Derint` para comprobar este punto). Razonad por qué.

5. Mejorad la función anterior para calcular una integral por el método de la regla trapezoidal dividida hasta alcanzar un cierto error especificado. El prototipo de la función puede ser:

   ```
   Medida Trapezoidal(func F, double a, double b, double tol);
   ```

   donde `F` es la función a integrar, `a y b` son los límites de integración y `tol` es la tolerancia requerida. La función debe realizar una serie de iteraciones internamente, doblando cada vez el número de subintervalos y calculando la precisión como la diferencia en valor absoluto entre el resultado actual (digamos con n subintervalos) y el resultado anterior (con $n/2$ subintervalos). Una vez que se alcance o mejore la precisión especificada en `tol` la función termina, devolviendo el valor de la integral y su error estimado en una variable de tipo `Medida`. Ayuda: podéis utilizar llamadas a la función pragramada en el ejercicio 3. Comprobad el correcto funcionamiento del programa.

6. Idéntica mejora para la integral por el método de Simpson. El prototipo de la función puede ser:

   ```
   Medida Simpson(func F, double a, double b, double tol);
   ```

Capítulo 8

Ajustes de datos

8.1. Introducción

Dedicamos esta práctica a familiarizarnos con problemas de ajustes por mínimos cuadrados, capítulo 11 del *Curso de Cálculo Numérico en C++*.

8.2. Directorio de trabajo

1. En primer lugar debéis situaros en el directorio raíz del curso (/home/calnum).

2. A continuación cread (si no existe ya) una nueva carpeta colgando del directorio raíz. En Linux (o desde el terminal de cygwin) la instrucción:

   ```
   mkdir Kavafis
   ```

   crea el directorio `Kavafis`[1]. También podéis crear el directorio (la carpeta) desde el entorno gráfico de Windows o Linux.

3. Entrad en vuestra carpeta de trabajo (el directorio `Kavafis` colgando del directorio raíz) y bajaos de la página web del curso[2] el fichero comprimido correspondiente a la práctica ocho `P8.tgz`. También tenéis que bajar el fichero comprimido `Util.tgz`.

4. Además, para esta práctica tenéis que bajar el fichero `DistAleatorias.tgz`.

5. Extraed el contenido de `P8.tgz` al directorio `P8`. Podéis hacerlo con el ratón o bien, desde un terminal con la instrucción:

   ```
   tar -xzf P8.tgz
   ```

6. Extraed el contenido de `Util.tgz` y `AlgebraLineal.tgz`.

7. Bajad al directorio `AlgebraLineal` y teclead `make` para crear la biblioteca de álgebra lineal.

8. Bajad al directorio `P8` y teclead `make`. Si habéis ejecutado los pasos anteriores correctamente obtendréis tres ejecutables, `xchi2.exe`,`xgendat.exe` y `xajuste.exe`

[1] Cambiad el nombre Kavafis por vuestro apellido, nombre de pila o apodo preferido.
[2] http://evalu29.ific.uv.es/docencia/calnum/index.html o bien http://www.uv.es/fisato/

8.3. Estudiad el ejemplo

Hemos preparado tres programas de ejemplo para esta práctica:

1. `xchi2`.- Genera la p.d.f. $\chi^2(\nu)$.
2. `xgendat`.- Genera un conjunto de datos de acuerdo a una función de usuario. Los datos se generan con errores distribuidos normalmente.
3. `xajuste`.- Ajusta un conjunto de datos a un modelo especificado por el usuario utilizando mínimos cuadrados (caso lineal).

Para crear los correspondientes ejecutables basta con teclear `make` en el directorio.

8.4. Ejemplo

En primer lugar generamos un conjunto de datos:

```
$ ./xgendat.exe
 Cuántos puntos quiere generar? 20
 Introduzca rango [xmin,xmax]
xmin? 0 xmax? 4
 Introduzca error medio de los puntos
sigma? 1.
 Introduzca dispersión de los errores
dispersión? 0.1
introduzca un entero para inicializar el
generador de números aleatorios 112345
Qué función desea?
Opciones: (1 = a*x+b, 2 = a*x**2+b*x+c, 3 = F)2
 Introduzca a,b,c
a? 4. b? 2. c? 1.
```

El programa genera 20 puntos (que almacena en el fichero `parabola.d`) con error promedio de una unidad (el error en sí mismo varía de unos puntos a otros de acuerdo con la dispersión del error, de 0.1) a partir de una parábola $y = ax^2 + bx + c$ con parámetros $a = 4$, $b = 2$ y $c = 1$.

A continuación ejecutamos el programa `xajuste.exe`:

```
$ ./xajuste.exe fichero de datos a ajustar? parabola.d Número
de parámetros del ajuste? 3
 Este programa ajusta una función modelo a un conjunto de 20 puntos
modelo a ajustar (1) recta (2) parabola (3) F?2
Parámetros
obtenidos por el ajuste
(        3.9           2             1.5 )
Matriz de covarianza
|    0.029         -0.11           0.071         |
|    -0.11          0.49           -0.35         |
|    0.071         -0.35            0.36         |

Chi2 = 0.75 Chi2 prob = 0.75
```

El programa lee un fichero con los datos (el que hemos generado previamente `parabola.d`). Debemos especificar además el modelo que queremos ajustar (en este caso una parábola) y el número de

parámetros del modelo. El resultado del ajuste es el vector de parámetros (en el ejemplo obtenemos $a = 3{,}9$, $b = 2$ y $c = 1{,}5$ y la matriz de covarianza de éstos, (los términos diagonales son los cuadrados de los errores de los parámetros). También obtenemos el chi2 por grados de libertad y su probabilidad, que indican que el ajuste es verosímil.

Los datos y el ajuste se muestran en la figura 8.1.

Figura 8.1: Ejemplo de ajuste a una parábola

8.5. Ejercicios

1. Considerad la función

$$f(x) = a \cdot \sin(x) + b \cdot e^{-x}$$

Se trata de una función trigonométrica modulada por una exponencial.

a) Representadla con gnuplot para los valores de $a = 2$ y $b = 20$.

b) Generad 30 puntos con `xgendat`, en el intervalo $[0, 10]$, con error de los puntos 1 y dispersión de los errores 0. Representad los conjuntos de datos con gnuplot. Usad la instrucción `plot "nombre-fichero"with yerrorbars`, para dibujar también los errores de los puntos. Ajustar los datos con `xajuste`. Comentad vuestros resultados.

c) A continuación generad de nuevo los datos tres o cuatro veces, con una semilla diferente para el generador aleatorio en cada caso. El chi2 y la probabilidad del chi2 no salen iguales. ¿Por qué?

d) Ejecutad el programa `xchi2.exe` y representar con gnuplot el resultado. El programa os produce un fichero (`chi2-nu28`) con la distribución del chi2 por grados de libertad. ¿Cómo podéis relacionar esa distribución con los resultados que acabáis de obtener?

2. Generar con `xgendat` una parábola $y = ax^2 + bx + c$ (20 puntos entre 0 y 40 con error medio 20 y dispersión del error 5) con los parámetros `a=0.1, b=5, c=0.5`. A continuación ajustar el fichero de datos que os sale (`parabola.d`) a dos modelos diferentes, una parábola y una recta. ¿Podéis decidir entre ambos modelos? ¿Por qué? Repetid el experimento generando 20 puntos entre los mismos límites pero con error promedio 4 y dispersión del error 1. ¿Se puede ahora decidir entre los dos modelos? Razonad la respuesta.

3. La abundancia de una muestra de partículas elementales que se desintegra de acuerdo a una vida media τ puede describirse mediante una función exponencial:

$$N(t) = N_0 \cdot e^{-t/\tau}$$

   a) El conjunto de datos `tau.d` se corresponde a un experimento ideal en el que se han realizado veinte medidas de la abundancia de cierta muestra de partículas elementales en función del tiempo. Representar el conjunto con gnuplot. Las abcisas son el tiempo, en unidades de microsegundos, las ordenadas el número de partículas en función del tiempo.

   b) La función problema no es lineal, claro está, pero puede hacerse lineal fácilmente. Hacedlo. Escribid una variante del programa `xajuste` (llamad a vuestra versión `xp8.cpp`, os hemos preparado una cáscara para el programa y la Makefile está así mismo preparada para compilar un programa que se llame `xp8`) que os permita obtener los parámetros N_0 y τ. Realizad el ajuste, comentad vuestros resultados. A partir de la vida media que habéis obtenido deducid que partículas estamos observando.

4. El caso de un ajuste a una recta se puede tratar de manera muy simplificada, sin recurrir a las ecuaciones normales. Repasad las ecuaciones desarrolladas en la sección 11.5.1 (ajuste de una recta) y modificadlas de tal manera que os sirvan para ajustar cualquier función lineal de dos parámetros $y = f_1(x) * p_1 + f_2(x) * p_2$. A continuación escribid una función que implemente el ajuste. La función puede tener como prototipo:

```
double lajuste(const RVector& x, const RVector& y, const DMatriz& V,
    double& p1, double& p2, RMatriz& C);
```

   Tomando como argumentos el vector de abcisas, el de ordenadas y la matriz (diagonal) de covarianza de los datos. La función devuelve en los argumentos, vía referencia, los dos parámetros del ajuste y la matriz de covarianza y en el tipo devuelto (un doble) el valor del chi2. No necesitáis un puntero a la función modelo puesto que la conocéis (una función lineal de dos parámetros) así que podéis evaluarla directamente en el interior de la función.

5. Repetid el ejercicio 1 con vuestra función. Comprobad que os salen resultados correctos.