Capítulo 8

Ajuste de datos experimentales

8.1. Generación de datos aleatorios

El programa `gendat.cpp` genera conjuntos de datos aleatorios para una serie de funciones. Nos servirá para tener conjuntos de datos experimentales simulados con los que verificar el funcionamiento de los programas de ajuste realizados en esta práctica. Se compila simplemente con

```
g++ -o gendat gendat.cpp
```

Estudiad en detalle el contenido del programa. La función `rand()` es un generador de números aleatorios que genera números con una distribución uniforme entre 0 y 1. La función `agauss()` utiliza `rand()` para generar números aleatorios que satisfacen una distribución gaussiana. La función rand() necesita una semilla, que en general es un entero grande. Para cada semilla se genera una serie de números aleatorios diferentes. Hemos tomado como semilla el tiempo del procesador dado por la función `time()`. El programa `gendat` genera ficheros de puntos adecuados para ser leídos como vectores utilizando la librería TNT, que son los ficheros con extensión `.d`, y otros ficheros, adecuados para ser dibujados por `gnuplot`, que tienen la extensión `.gpl`. El programa utiliza la función `popen()` para dibujar este último fichero con `gnuplot`. Entre los ficheros de la práctica, están `genpol.d`, `genparabola1.d`, `genclf.d` y `gentau.d`. Estos son ficheros de datos para `gendat`, para generar los datos de los diferentes ejercicios. Ejecutad

```
gendat<genpol.d
gendat<genclf.d
gendat<gentau.d
```

para generar los ficheros de datos necesarios.

Ejercicio 1: Generad los datos y dibujadlos con `gnuplot`.

8.2. La distribución χ^2

El programa `chi2.pp` es una función que calcula χ^2 y su probabilidad. Tiene programado en su interior una serie de funciones especiales. Para compilar esta función, hay que introducir

en los otros ficheros que se compilen con ella la cabecera `chi2.h`. El programa `chi2main.cpp` calcula la distribución chi2 para un número dado de grados de libertad. Se compila con

```
g++ -I. -o chi2 chi2main.cpp chi2.cpp
```

Con este programa se genera la función de distribución de χ^2 para un número de grados de libertad dado. Es útil para conocer la probabilidad de valores de χ^2 obtenidos en los ajustes de datos, o comparar con un histograma de muchos ajustes consecutivos. Si se realizan muchos ajustes de conjuntos de datos comparables con un número de grados de libertad dado, el histograma de los valores de χ^2 obtenido debe de comportarse como la distribución χ^2 con estos grados de libertad, una vez que se normalice al número total de sucesos.

El único dato del programa es el número de grados de libertad. Este programa genera dos ficheros, `chi2-nuN` con la distribución de probabilidad, y `pchi2-nuN` con la probabilidad integrada de que χ^2 sea mayor que un valor dado, donde `N` denota el número de grados de libertad. Estos dos ficheros están preparados para ser dibujados con `gnuplot`.

Ejercicio 2: Generad los datos para 5, 10, y 30 grados de libertad y representadlos con `gnuplot`.

8.3. Ajuste de datos mediante una línea recta

El programa `linfit.cpp` realiza el ajuste mediante una combinación lineal de dos funciones, definidas por el usuario, a una serie de datos experimentales con errores en la ordenada. Está planteado como un único programa, sin funciones y sin llamadas a clases. Este programa consiste de una única función `main()` que lee los datos, calcula los coeficientes que ajustan los datos e imprime los resultados. El único requerimiento para su funcionamiento es un compilador C++. Se puede modificar fácilmente para el ajuste de una recta $y = mx + n$. Los datos se leen desde la entrada estándar, normalmente desde un fichero, e.g. `linfit.dat`, que contiene el número de datos en la primera línea, y cada dato, en el orden abcisa, ordenada y error de la ordenada, en cada una de las siguientes líneas. Se ejecuta como

```
linfit<linfit.dat
```

El programa imprime los datos leídos y los resultados del ajuste: parámetros con su error, matriz de covarianzas, y χ^2 total y por punto. Imprime también un fichero, `linfit.gpl`, que contiene en las tres primeras columnas los datos experimentales con su error, y en la cuarta los valores ajustados. Este fichero sirve para visualizar los resultados del ajuste con `gnuplot`. Dentro de `gnuplot` hacemos

```
plot "linfit.gpl" using 1:2:3 with yerrorbars
replot "linfit.gpl" using 1:4 with lines
```

A continuación se presenta el contenido del fichero `linfit.cpp`:

```
// Programa ajuste lineal mediante dos funciones
// Compilar con g++ -o linfit linfit.cpp

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

```
#include <iomanip>
#include <fstream>
using namespace std;

//Introducir aqui las dos funciones con las que se ajusta
// f1-- > 1 f2-- > x linea recta y = a +  b*x
inline double f1(double x)
{
    return cos(x);
}


inline double f2(double x)
{
    return sin(x);
}


int
main()
{

//Fichero de datos - cambiar cin --> fin
// ifstream fin("linfit.dat");
    int n;

//Leer numero de puntos n
    cin >> n;

//numero de funciones de ajuste.
    int m = 2;

//Definir matriz de coeficientes y termino independiente
    double **mat;
    mat = new double *[m];

    for (int i = 0; i < m; i++)
        mat[i] = new double[m + 1];

//Definir vectores de puntos y de errores
    double *y = new double[n];
    double *x = new double[n];
    double *sigma = new double[n];
```

```
//Definir matriz de covarianzas
    double **covp;
    covp = new double *[m];

    for (int i = 0; i < m; i++)
        covp[i] = new double[m];

//Leer puntos
    for (int i = 0; i < n; i++)
        cin >> x[i] >> y[i] >> sigma[i];

//Verificar que la lectura es correcta
    cout << "Puntos experimentales" << endl;
    for (int j = 0; j < n; j++)
        cout << x[j] << "   " << y[j] << "   " << sigma[j] << "   " << endl;
    cout << endl;

//Calcular matriz coeficientes
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m + 1; j++)
            mat[i][j] = 0;
    for (int i = 0; i < n; i++)
        mat[0][0] = mat[0][0] + f1(x[i]) * f1(x[i]) / (sigma[i] * sigma[i]);
    for (int i = 0; i < n; i++)
        mat[0][1] = mat[0][1] + f1(x[i]) * f2(x[i]) / (sigma[i] * sigma[i]);
    for (int i = 0; i < n; i++)
        mat[1][1] = mat[1][1] + f2(x[i]) * f2(x[i]) / (sigma[i] * sigma[i]);
    for (int i = 0; i < n; i++)
        mat[0][2] = mat[0][2] + f1(x[i]) * y[i] / (sigma[i] * sigma[i]);
    for (int i = 0; i < n; i++)
        mat[1][2] = mat[1][2] + f2(x[i]) * y[i] / (sigma[i] * sigma[i]);
    mat[1][0] = mat[0][1];

//Imprimir matriz de coeficientes
    cout << "Matriz coeficientes" << endl;
    for (int i = 0; i < m; i++) {
        cout << endl;
        for (int j = 0; j < m + 1; j++)
            cout << mat[i][j] << "   ";
    }
    cout << endl << endl;
```

```
//Calculo de las soluciones
    double a, b, det;

    det = mat[0][0] * mat[1][1] - mat[0][1] * mat[0][1];
    a = mat[0][2] * mat[1][1] - mat[1][2] * mat[0][1];
    b = mat[0][0] * mat[1][2] - mat[0][1] * mat[0][2];
    a = a / det;
    b = b / det;
    cout << "a= " << a << "  b= " << b << endl << endl;

//Calculo de los errores de los parametros
    covp[0][0] = mat[1][1] / det;
    covp[1][1] = mat[0][0] / det;
    covp[1][0] = covp[0][1] = -mat[0][1] / det;
    double ea = sqrt(covp[0][0]);
    double eb = sqrt(covp[1][1]);

    cout << "ea= " << ea << "  eb= " << eb << "  cov(a,b)="
         << covp[0][1] << endl << endl;

//Calculo de chi2
    double chi2 = 0;
    for (int i = 0; i < n; i++)
        chi2 = chi2 + pow((a * f1(x[i]) + b * f2(x[i]) - y[i]) / sigma[i], 2);
    int nu = n - 2;
    double chi2nu = chi2 / nu;
    cout << "chi2= " << chi2 << "  chi2/nu=  " << chi2nu << endl;

//Creacion de fichero gnuplot
    ofstream fout("linfit.gpl");
    for (int i = 0 ; i < n; i++ )
      fout << x[i] << "  " << y[i] << "  " << sigma[i] << "  " <<
a * f1(x[i])  + b * f2(x[i]) << endl;
    fout.close();

//Destruccion de la memoria asignada
    for (int i = 0; i < m; i++)
        delete[] mat[i];
    for (int i = 0; i < m; i++)
        delete[] covp[i];
    delete[] mat;
    delete[] covp;
    mat = 0;
```

```
    covp = 0;
    return 0;
}
```

Ejercicio 3: Compilad y ejecutad el programa. con el fichero linfit.dat. Comparar el valor de χ^2 obtenido con la distribución de χ^2 correspondiente.

8.4. Ajuste lineal de orden superior

8.4.1. Ajuste polinomial

El ajuste polinomial ajusta los datos mediante una combinación lineal de un número de potencias consecutivas de la variable independiente, hasta un valor máximo determinado por el orden del ajuste. El programa `polfit.cpp`, dado a continuación, realiza el ajuste mediante un polinomio de grado `m`. Los datos se leen de la entrada estándar. En la práctica, se leerán del fichero `pol.d`, generado por `gendat.cpp`, ejecutando el programa de la forma `polfit<pol.d`. El programa utiliza las clases `Vector` y `Matrix` de la librería TNT junto con algunos los templates adicionales codificados en `templates.h`. Como se observará, la programación es mucho más compacta que en el caso del ajuste lineal, en parte debido que se a utilizado el operador de producto escalar de vectores. Además es mucho más legible (a condición de haber estudiado la teoría). El programa dibuja los datos junto con el ajuste utilizando la librería `gnuplot_i++`.

```
// Programa de ajuste polinomial
//Compilar con g++ -I. -I./templates -o polfit polfit.cpp
//chi2.cpp gnuplot_i_calnum.cpp

#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>
#include "algebralineal.h"
#include "templates.h"
#include "chi2.h"
#include "gnuplot_i_calnum.h"
using namespace std;
using namespace CALNUM;
typedef double Real;

int main()
{
//Fichero de datos
// ifstream fin("F.d");
    int n;
```

```
    int m;

//Leer numero de parametros = grado del polinomio + 1)
    cin >> m;

//Declaracion de matriz de coeficientes, de covarianzas y vector
// de terminos independientes
    Matrix < Real > mat(m, m), covp(m, m);
    Vector < Real > vec(m);

//Definir vectores de puntos y de errores
    Vector < Real > x;
    Vector < Real > y;
    Vector < Real > sigma;

//Leer puntos
    cin >> x >> y >> sigma;
    n = x.size();

//Verificar que la lectura es correcta
    cout << "Puntos experimentales" << endl << "     x    " <<
          "       y     " << "      sigma  " << endl;
    for (int j = 0; j < n; j++)
        cout << setw(10) << x[j] << "   " << setw(10) << y[j] <<
              "   " << setw(10) << sigma[j] << "   " << endl;
    cout << endl;

//Calcular matriz coeficientes y vector de terminos independientes
    for (int i = 1; i <= m; i++) {
        vec(i) = (pow(x, i - 1) / sigma) * (y / sigma);

        for (int j = 1; j <= i; j++) {
            mat(i, j) = (pow(x, i - 1) / sigma) * (pow(x, j - 1) / sigma);
            if (i > j)
                mat(j, i) = mat(i, j);
        }
    }

//Imprimir matriz de coeficientes
    cout << "Matriz de coeficientes" << endl << mat << endl;
    cout << "vector de terminos independientes" << endl << vec << endl;

//Calculo de las soluciones
```

```
    Vector < Real > a(m);
    LU < Real > lu(mat);
    a = lu.solve(vec);
    cout << "Vector de parametros a " << endl << a << endl;

//Calculo de los errores de los parametros
    covp = lu.inv();
    Vector < Real > ea = diag(covp);
    cout << "Vector de errores de los parametros ea " << endl <<
         sqrt(ea) << endl;

//Calculo de chi2
    Real chi2 = normsq((pol(a, x) - y) / sigma);
    Real nu = n - m;
    Real chi2nu = chi2 / nu;
    Real chi2p = Chi2P(chi2, nu);

    cout << "chi2= " << chi2 << "  chi2/nu=  " << chi2nu <<
          " Prob(chi2/nu)= " << chi2p << endl;

//evaluacion del polinomio y funcion en los puntos de dibujo(200)
// se dibuja un 2 * deltax100 % fuera del intervalo
    int ndib = 200;
    double delta = 0.;

//cout << " Entrar delta" <<
//"(Dibujo en [x0-(x1-x0)*delta, x1+(x1-x0)*delta] " << endl << endl;
//cin >> delta;
    double x1 = x[n - 1] + (x[n - 1] - x[0]) * delta;
    double x0 = x[0] - (x1 - x0) * delta;
    double hh = (x1 - x0) / ndib;

    ndib++;
    Vector < double >xx(ndib), yy(ndib), ff(ndib);

    for (int i = 0; i < ndib; i++) {
        xx[i] = x0 + i * hh;
        yy[i] = pol(a, xx[i]);
    }

//Dibujo del polinomio y la funcion
    Gnuplot g1 = Gnuplot();
    g1.set_style("lines");
```

```
    g1.plot_xy(xx, yy, "Polinomio ajustado ");
    g1.set_style("points");
    g1.cmd("replot \"parabola.gpl\" with errorbars");

//Descomentar y seleccionar terminal de gnuplot  y extension del fichero
// para imprimir en fichero grafico
// g1.cmd("set terminal postscript");
//g1.cmd("set output 'polfit.ps' ");
//g1.plot_xy(xx, yy, "Polinomio ajustado ");
//g1.cmd("replot \"pol.gpl\" with errorbars");

//Eleccion del tiempo de permanencia de la grafica en pantalla
    int SLEEP_LGTH = 25;

    sleep(SLEEP_LGTH);
    return 0;
}
```

El programa se compila con

```
g++ -I. -I./templates -o polfit polfit.cpp chi2.cpp gnuplot_i_calnum.cpp
```

Ejercicio 4: Compilad el programa. Generad con `gendat` datos correspondientes a la función $(1+x)^m$ con error de los datos$\sigma = 1$ y el error de σ nulo, para diferentes valores de m, y ajustadlos con `polfit`. Podéis poner los datos de `gendat` en un fichero llamado `genpol.dat` por ejemplo, en el que modificáis el valor de `m`. El comando

```
gendat<genpol.dat
```

genera cada vez un fichero llamado `pol.d`. Para ajustar los datos contenidos en este fichero, simplemente hacéis

```
polfit<pol.d
```

si se desea escribir los los resultados del programa en pantalla, o con

```
polfit<pol.d>polres.dat
```

si se desea escribirlos en el fichero `polres.dat`. Junto con los programas de la práctica, se proporcionan los ficheros `genpol.d`, con los datos de `gendat` para generar conjuntos de datos aleatorios de $(1+x)^5$.

8.4.2. Ajuste lineal por una combinación lineal de funciones

El programa `funfit.cpp` ajusta puntos experimentales mediante una combinación lineal de funciones. Hay definidas 10 funciones, que se seleccionan mediante un úndice entero. La llamadas a las funciones son de la forma `func(x,j)`, donde `x` puede ser un escalar o un vector. En el segundo caso el resultado es un vector cuyos elementos son los valores de `fun(x[i],j)`. El orden de definición de las funciones es el mismo que en `gendat`, de forma que se pueda comparar los coeficientes de los datos generados y los obtenidos mediante ajuste. Al igual que en el programa `polfit.cpp` se ha utilizado la librería TNT y las funciones suplementarias definidas,

con las que se logra una programación elegante y compacta. Se ha definido un template de una función cuyos argumentos son un escalar y un vector, y que devuelve un escalar, dado por el valor de la combinación lineal de funciones definida por el vector, en el argumento escalar. Se ha definido una función cuyos argumentos son dos vectores, y que devuelve un vector, cuyos elementos son los valores de la combinación lineal de funciones definida por el segundo vector, en el conjunto de puntos definido por el primer vector:

$$\overrightarrow{f}(\overrightarrow{x},\overrightarrow{a}) = \left(\sum_{j=0}^{m} a_j f(x_0,j), \sum_{j=0}^{m} a_j f(x_1,j), \ldots, \sum_{j=0}^{m} a_j f(x_n,j)\right)$$

Se calcula la probabilidad asociada al valor de χ^2 compilando `funfit.cpp` con `chi2.cpp`. Los datos se dibujan compilando con `gnuplot_i_calnum.cpp`.

El programa se compila con

```
g++ -I. -I./templates -o funfit funfit.cpp chi2.cpp
```

Si generáis datos con `gendat`, se creará un fichero llamado `clfun.d`. Para ajustar esos datos con `funfit`, simplemente ejecutáis funfit leyendo el fichero `clfun.d` por la entrada estándar:

```
funfit<clfun.d
```

con lo que se obtienen los resultados en pantalla. Si queréis obtener los datos en un fichero de resultados llamado `funfit.res`, por ejemplo, simplemente hacéis

```
funfit<clfun.d>funfit.res
```

Estudiad en detalle el contenido de este programa. Observad que en un momento determinado se ha definido un vector de vectores, mediante los templates definidos en el programa.

A continuación se da un listado del programa `funfit.cpp`:

```
// Programa de ajuste por combinacion lineal de funciones
// Compilar con g++ -I. -I./ templates - o funfit funfit.cpp
// chi2.cpp gnuplot_i_calnum.cpp

#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>
#include "algebralineal.h"
#include "templates.h"
#include "chi2.h"
#include "gnuplot_i_calnum.h"
using namespace std;
typedef double Real;
using namespace CALNUM;

//Declaracion de de funciones
template < class T > T func(T & x, Vector < T > &a)
{
```

```
//Combinacion lineal de funciones definidas en un punto.
//Coeficientes en vector a.
    int n = a.size();
    T   tmp = T(0);

    for (int i = 0; i < n; i++)
        tmp += a[i] * func(x, i);
    return tmp;
}


template < class T > Vector < T > func(Vector < T > &v, Vector < T > &a)
{
//Combinacion lineal de funciones definidas en un vector de puntos.
//Coeficientes definidos en vector a
    int n = v.size();

    Vector < T > tmp(n);
    for (int i = 0; i < n; i++)
        tmp[i] = func(v[i], a);
    return tmp;
}


template < class T > Vector < T > func(Vector < T > &v, int &p)
{
//Funcion definida en  un vector de puntos.
    int n = v.size();

    Vector < T > tmp(n);
    for (int i = 0; i < n; i++)
        tmp[i] = func(v[i], p);
    return tmp;
}


template < class T > T func(T & x, int &i)
{
//Definicion de las funciones que aparecen en la combinacion lineal
//Deben ser las mismas que en gendat.
//Definir aqui con un switch cada
// una de las funciones para cada valor de i
    switch (i) {
```

```
        case 0:
            return sin(x);
            break;
        case 1:
            return exp(-x);
            break;
        case 2:
            return sqrt(x);
            break;
        case 3:
            return log(x);
            break;
        case 4:
            return tan(x);
            break;
        case 5:
            return cosh(x);
            break;
        case 6:
            return cos(x);
            break;
        case 7:
            return pow(x, 3);
            break;
        default:
            return 0.;
    }
}


int main()
{

//Fichero de datos alternativo a la entrada estadar
// ifstream fin("F.d");
    int n;
    int m;

//Leer numero de funciones m(numero de parametros)
    cin >> m;

//Declaracion de matriz de coeficientes, de covarianzas y vector
// de terminos independientes
```

```
    Matrix < Real > mat(m, m), covp(m, m);
    Vector < Real > vec(m);

//Definir vectores de puntos y de errores
    Vector < Real > x;
    Vector < Real > y;
    Vector < Real > sigma;

//Leer puntos
    cin >> x >> y >> sigma;
    n = x.size();

//Verificar que la lectura es correcta
    cout << "Puntos experimentales" << endl << "     x    "
         << "       y     " << "      sigma  " << endl;
    for (int j = 0; j < n; j++)
        cout << setw(10) << x[j] << "   " << setw(10) << y[j]
             << "   " << setw(10) <<   sigma[j] << "   " << endl;
    cout << endl;

//Creamos un vector de vectores cuyos elementos
// son los vectores formados por los valores de cada una de las funciones en el
// vector conjunto de puntos
    Vector < Vector < Real > > vfunc(m, Vector < Real > (n));
    for (int i = 0; i < m; i++)
        vfunc[i] = func < Real > (x, i) / sigma;
    Vector < Real > vy = y / sigma;
//Calcular matriz coeficientes y vector de terminos independientes
    for (int i = 0; i < m; i++) {
//vec[i] = (func < Real > (x, i) / sigma) * (y / sigma);
        vec[i] = vfunc[i] * vy;
//Calculamos solo la parte triangular superior de mat. Simetrica.
        for (int j = 0; j <= i; j++) {
//mat[i][j] = (func < Real > (x, i) / sigma) * (func < Real > (x, j) / sigma);
            mat[i][j] = vfunc[i] * vfunc[j];
            if (i > j)
                mat[j][i] = mat[i][j];
        }
    }

//Imprimir matriz de coeficientes
    cout << "Matriz de coeficientes" << endl << mat << endl;
    cout << "vector de terminos independientes" << endl << vec << endl;
```

```
//Calculo de los parametros ajustados a-- > Solucion mat * a = vec;
    Vector < Real > a(m);
    LU < Real > lu(mat);
    a = lu.solve(vec);
    cout << "Vector de parametros a " << endl << a << endl;

//Calculo de los errores de los parametros
    covp = lu.inv();
    Vector < Real > ea = diag(covp);
    cout << "Vector de errores de los parametros ea " << endl
         << sqrt(ea) << endl;

//Calculo de chi2
    Real chi2 = normsq((func(x, a) - y) / sigma);
    Real nu = n - m;
    Real chi2nu = chi2 / nu;
    Real chi2p = Chi2P(chi2, nu);

    cout << "chi2= " << chi2 << "  chi2/nu=  " <<
         chi2nu << " Prob(chi2/nu)= " << chi2p << endl;

//evaluacion del polinomio y funcion en los puntos de dibujo(200)
// se dibuja un 2 * deltax100 % fuera del intervalo
    int ndib = 200;
    double delta = 0.;

 // cout << " Entrar delta" <<endl;
 //cout << "(Dibujo en [x0-(x1-x0)*delta, x1+(x1-x0)*delta] " << endl << endl;
 //    cin >> delta;
    double x1 = x[n - 1] + (x[n - 1] - x[0]) * delta;
    double x0 = x[0] - (x1 - x0) * delta;
    double hh = (x1 - x0) / ndib;

    ndib++;
    Vector < double >xx(ndib), yy(ndib);

    for (int i = 0; i < ndib; i++) {
        xx[i] = x0 + i * hh;
        yy[i] = func(xx[i], a);
    }

//Dibujo del polinomio y la funcion
```

```
    Gnuplot g1 = Gnuplot();
    g1.set_style("lines");
    g1.plot_xy(xx, yy, "Polinomio ajustado ");
    g1.set_style("points");
    g1.cmd("replot \"clfun.gpl\" with errorbars");

//Descomentar y seleccionar terminal y extension para
//imprimir en fichero grafico
// g1.cmd("set terminal postscript");
//g1.cmd("set output 'clfunfit.ps' ");
//g1.plot_xy(xx, yy, "Polinomio ajustado ");
//g1.cmd("replot \"clfun.gpl\" with errorbars");

//Eleccion del tiempo de permanencia de la grafica en pantalla
    int SLEEP_LGTH = 25;

    sleep(SLEEP_LGTH);
    return 0;
}
```

8.5. Ajuste con polinomios ortogonales sobre un conjunto de puntos

El uso de funciones ortogonales evita el mal condicionamiento de las ecuaciones y tiene la propiedad de permanencia, es decir, si queremos aumentar el orden del ajuste podemos aprovechar lo que ya hemos hecho. Aunque el cálculo de coeficientes elevados se ve también afectado por los límites del mayor y menor real representables por el compilador, debido a que los polinomios ortogonales de orden elevado contienen potencias elevadas, el uso de aritmética de precisión arbitraria se puede limitar a los coeficientes más elevados que sean problemáticos, calculando los coeficientes más bajos con aritmética ordinaria. El siguiente programa utiliza el ajuste mediante polinomios ortogonales sobre un conjunto discreto de puntos, con pesos iguales al cuadrado de los errores, es decir, de forma que los productos escalares den las mismas ecuaciones normales que del método de minimización de χ^2.

```
// Programa de ajuste por polinomios ortogonales
// Compilar con
// g++ -I. -I./templates -o ortpolfit ortpolfit.cpp chi2.cpp
// gnuplot_i_calnum.cpp
// Utilizacion:./ ortpol < pol.d
#include <unistd.h>
#include "gnuplot_i_calnum.h"
#include <iostream>
```

```
#include <cmath>
#include <iomanip>
#include <fstream>
#include "algebralineal.h"
#include "templates.h"
#include "chi2.h"
using namespace std;
using namespace CALNUM;
typedef double Real;

//Calculo de polinomios ortogonales

template < class T >
Vector < Vector < T > >ortpol(const Vector < T > &x, const Vector < T > &y,
const Vector < T > &sigma, Vector < T > &beta, Vector < T > &gamma, const int &m)
{
//Devuelve m vectores de dimension n que contienen los valores
// de los m polinomios ortogonales sobre n puntos
// en los n puntos
// Los vectores beta y gamma dan los coeficientes de recurrencia
// de los polinomios ortogonales
    int n = x.size();

    Vector < Vector < T > >ortpol(m, Vector < T > (n));
    Vector < T > opold, opnew;
    Vector < T > xsigma = x / sigma;
    T   tmpold, tmpnew;

    ortpol[0] = Vector < T > (n, T(1.));
    opnew = ortpol[0] / sigma;
    tmpnew = opnew * opnew;
    gamma[0] = T(0.);
    beta[0] = x * pow(opnew, 2) / tmpnew;
//Simbolo %:producto de arrays elemento a elemento(libreria TNT)
    ortpol[1] = (x - beta[0]) % ortpol[0];
    for (int i = 2; i < m; i++) {
        tmpold = tmpnew;
        opnew = ortpol[i - 1] / sigma;
        tmpnew = opnew * opnew;
        beta[i - 1] = x * pow(opnew, 2) / tmpnew;
        gamma[i - 1] = tmpnew / tmpold;
        ortpol[i] = (x - beta[i - 1]) % ortpol[i - 1] - gamma[i - 1] * ortpol[i - 2];
```

```
    }
    return ortpol;
}


template < class T >
Vector < T > eval_ortpol(const Vector < T > &beta, const Vector < T > &gamma,
const Vector < T > &a, Vector < T > &xx)
{
//Evalua una combinacion de polinomios ortogonales con vector de coeficientes
// a en las abcisas dadas por el vector xx
// beta y gamma son los coeficientes de la relacion de recurrencia
// de los polinomios ortogonales

// numero de coeficientes
    int m = a.size();

//numero de puntos
    int n = xx.size();

//vector de valores de la combinacion lineal
    Vector < T > fun(n, T(0));
//ortpol: vector de vectores(elemento i:polinomio i en los n puntos)
    Vector < Vector < T > >ortpol(m, Vector < T > (n));
    ortpol[0] = Vector < T > (n, T(1.));
    ortpol[1] = (xx - beta[0]) % ortpol[0];
    for (int i = 2; i < m; i++) {
        ortpol[i] = (xx - beta[i - 1]) % ortpol[i - 1] - gamma[i - 1] * ortpol[i - 2];
    }
    for (int i = 0; i < m; i++)
        fun += a[i] * ortpol[i];
    return fun;
}


int
main()
{

//Fichero de datos "pol.d" leido de la entrada standard ortopolfit < pol.d
    int n;
    int m;
```

```
//Leer numero de funciones m(numero de parametros)
    cin >> m;

//Declaracion de vector de coeficientes(vcoef), de covarianzas(covp) y vector
//  de terminos independientes(a).beta y gamma definen los polinomios
    Vector < Real > vcoef(m), covp(m), a(m), beta(m), gamma(m);

    Vector < Real > vec(m);

//Definir vectores de puntos y de errores
    Vector < Real > x;
    Vector < Real > y;
    Vector < Real > sigma;

//Leer puntos
    cin >> x >> y >> sigma;
    n = x.size();

//Si los puntos no tienen error el mismo error para todos
// Minimos cuadrados en vez de Chi2
    for (int i = 0; i < n; i++)
        if (sigma[i] == 0)
            sigma[i] = 1.;

//Verificar que la lectura es correcta
    cout << "Puntos experimentales" << endl << "     x    " << "       y     " <<
        "      sigma  " << endl;
    for (int j = 0; j < n; j++)
        cout << setw(10) << x[j] << "   " << setw(10) << y[j] << "   " <<
              setw(10) << sigma[j] << "   " << endl;
    cout << endl;

//si sigma = 0. ponemos sigma = 1. Puntos con el mismo error.Minimos cuadrados.
    for (int i = 0; i < n; i++)
        if (sigma[i] == 0)
            sigma[i] = 1.;

//Creamos un vector de vectores para almacenar cada una de las funciones en el
// conjunto de puntos
    Vector < Vector < Real > >vortpol(m, Vector < Real > (n));
    vortpol = ortpol < Real > (x, y, sigma, beta, gamma, m);

//Comprobacion de ortogonalidad
```

```
// for (int i = 0; i < m; i++) {
//for (int j = 0; j <= i; j++) cout << "  " << i << "  " << j << "  "
// <<vortpol[i] * vortpol[j];
//cout << endl;
//}


Vector < Real > vy = y / sigma;
Vector < Vector < Real > >vortpolsigma(m);
for (int i = 0; i < m; i++)
    vortpolsigma[i] = vortpol[i] / sigma;

//Calcular vector de coeficientes vecoef
// y vector de terminos independientes vec
for (int i = 0; i < m; i++)
{
    vec[i] = (vortpolsigma[i]) * vy;
    vcoef[i] = vortpolsigma[i] * vortpolsigma[i];
}


//Calculo del vector de parametros
a = vec / vcoef;

//Impresion de los coeficientes de recurrencia
cout << "beta=" << beta << endl;
cout << "gamma=" << gamma << endl;

//Imprimir matriz de coeficientes, de terminos independientes y parametros
cout << "Vector de coeficientes" << endl << vcoef << endl;
cout << "vector de terminos independientes" << endl << vec << endl;
cout << "Vector de parametros a " << endl << a << endl;

//Calculo de los errores de los parametros
Vector < Real > ea = Real(1.) / vcoef;
cout << "Vector de errores de los parametros ea " << endl << sqrt(ea) << endl;

//Calculo de la funcion ajustada en el vector fun
// vortpol[i] es un vector de vectores cada uno de los cuales contiene
//  los valores de l polinomio ortogonal[i] en los n puntos de ajuste
//  a[i] * vortpol[i] producto de escalar por vector
Vector < Real > fun(n, Real(0));
```

```
for (int i = 0; i < m; i++)
    fun += a[i] * vortpol[i];
cout << "Puntos ajustados=" << fun << endl;

//Calculo de chi2
Real chi2 = normsq((fun - y) / sigma);
Real nu = n - m;
Real chi2nu = chi2 / nu;
Real chi2p = Chi2P(chi2, nu);

cout << "chi2= " << chi2 << "  chi2/nu=  " << chi2nu <<
" Prob(chi2/nu)= " << chi2p << endl;

//Dibulo puntos con su error y funcion ajustada
// Dibujamos la funcion ajustada en 10 veces elnumero
// de puntos experimentales
int ndib = 10 * n;

Vector < Real > xx(ndib), ff(ndib);
double h = abs(x[n - 1] - x[0]) / (ndib - 1);

for (int i = 0; i < ndib; i++)
    xx[i] = x[0] + h * i;
ff = eval_ortpol < Real > (beta, gamma, a, xx);
//Impresion de la funcion ajustada en ortpol.gpl
ofstream fout("ortpol.gpl");
for (int i = 0; i < ndib; i++)
    fout << xx[i] << "  " << ff[i] << endl;
fout.close();

//Creacion de elemento de la clase Gnuplot
Gnuplot g1 = Gnuplot();
g1.set_style("lines");
g1.plot_xy(xx, ff, " Ajuste polinomios ortogonales ");
g1.cmd("replot \"pol.gpl\" with errorbars");

//Tiempo de permanencia de la grafica en pantalla
int SLEEP_LGTH = 30;

sleep(SLEEP_LGTH);
return 0;
}
```

El programa `ortopolfit` calcula el valor de χ^2 y su probabilidad y dibuja los datos utilizando `gnuplot`. Se compila con

```
g++ -I. -I./templates -o ortopolfit ortopolfit.cpp chi2.cpp gnuplot_i_calnum.cpp
```

Ejercicio 5: Compilad el programa. Ejecutadlo con el fichero `pol.d`, generado anteriormente.

8.6. Ejercicios a presentar como memoria

1. Considerad la función
$$f(x) = a \cdot \sin(x) + b \cdot e^{-x}$$

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

   b) Generad 40 puntos con `gendat` en el intervalo [0,10], con error de los puntos 1 y dispersión de los errores 0. Utilizad el modelo de función combinación lineal de funciones con 2 funciones, y tomad 2 y 20 como los valores de los parámetros. Representad el conjunto de datos, creado en el fichero `clfun.gpl`, con `gnuplot`
   ```
   plot "clfun.gpl" with errorbars
   ```

   c) Ajustad el conjunto de datos con `funfit.cpp`. Anotad el valor de chi2 y de la probabilidad de chi2. Realizad el proceso 10 veces. Para automatizarlo, utilizad el fichero `genclfun.d` en el que deberéis verificar el valor correcto de los parámetros. Cada ejecución de
   ```
   ./gendat<genclfun.d
   ```
   produce un fichero de datos distinto, llamado `clfun.d`, que ajustáis con
   ```
   ./funfit<clfun.d
   ```
   Anotáis chi2 y chi2/nu para cada ejecución. Con el programa `chi2` generad la distribución χ^2 con 38 grados de libertad (compilad con `g++ -o chi2 chi2main.cpp chi2.cpp`). Representad un histograma con los valores de χ^2 obtenidos junto con la distribución obtenida. Ponéis los puntos de este histograma, normalizados a la densidad de probabilidad (es decir frecuencias divididas por el número total de sucesos -10 en vuestro caso- y divididas por la anchura del intervalo), en un fichero, por ejemplo `chi2.gpl` y lo dibujáis después de `chi2-nu38` con
   ```
   replot "chi2.gpl" with boxes
   ```
   ¿Qué conclusión extraéis? Tomad cuatro o cinco intervalos de clase, por ejemplo [20,25], [25,30], [30,35], 35,40], [40,60].
   Entregad el histograma, uno de los resultados de ajustes y el fichero `clfun.d` con el que se ha obtenido.

2. Generad con `gendat` una parábola $y = ax^2 + bx + c$. Tomad 20 puntos comprendidos en el intervalo de x [0, 40] con error medio de los puntos 20 y dispersión del error 5. Tomad como parámetros $a = 0{,}1$, $b = 5$, y $c = 0{,}5$. A continuación ajustad el fichero de datos que os sale `parabola.d` con el programa `polfit.cpp` a una parábola y a una recta. Para esto último, tendréis que cambiar el número de parametros de `parabola.d` de 3 a 2 (editando con

`emacs` por ejemplo). ¿Podéis decidir cual es el mejor de ambos modelos mediante el valor de χ^2? Volved a generar los puntos, pero esta vez con un error promedio de 4 y dispersión del error 1. ¿Se puede ahora decidir entre los dos modelos? Razonad la respuesta. Entregad los dos ficheros `parabola.d` y los resultados de los 4 ajustes.

3. Las partículas elementales y en general todos los elementos radiactivos se desintegran con una vida media τ y su abundancia en función del tiempo sigue la ley exponecial

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

donde N_0 es el número de partículas en $t = 0$. Esta distribución se genera mediante el modelo `tau` de `gendat`.

   a) Generad un conjunto de 20 puntos mediante `gendat`, con la opción tau, eligiendo 10000 como valor de N_0 y tomando $\tau = 2$ y en intervalo entre 0 y 10. El error sale como $\sqrt{N}$ independientemente del valor que le déis a sigma. Si la unidad de tiempo son microsegundos, esta ley correspondería a la desintegración de muones ($\tau = 2\mu s$). Intentad aproximarlo con `polfit` con un orden elevado, por ejemplo 5. Comentad el resultado obtenido. La ley, tal como la acabamos de escribir es no lineal. Deberéis de copiar `tau.gpl` en `parabola.gpl` (`cp tau.gpl parabola.gpl`) para que dibuje los resultados del ajuste con los datos en vez de con los datos del ejercicio 2.

   b) Escribid la ley anterior tomando logaritmos. Resulta una ley lineal. Ajustad el logaritmo de las ordenadas mediante una recta en función de la abcisa, utilizando por ejemplo linfit.cpp, convenientemente modificado. Podéis tambien sustituir las ordenadas por sus logaritmos y los errores por los errores de los logaritmos en el fichero de datos, y utilizar polfit.cpp, aunque es más simple tomar los logaritmos de las ordenadas en `linfit2.cpp`, y sustituir `sigma[i]` por `sigma[i]/y[i]` ¿Que valor del tiempo de vida y del número inicial de partículas encontráis? (`linfit2.cpp` es el programa `linfit.cpp` con las funciones de ajuste $f1(x) = 1.$ y $f2(x) = x.$)

4. Generad mediante `gendat`, un conjunto de 40 puntos en el intervalo $[0,3]$ utilizando la función tipo 5, $y = (a + 0{,}1 * x)^n$, tomando $a = 1$ y $n = 20$. Tomad error de los puntos 0.01 y dispersión del error 0. Se genera el fichero `pol.d` en el que figuran 21 parámetros de ajuste en la primera línea. Ajustadlo con `polfit.cpp` y `ortopolfit.cpp`. Recompilad `polfit.cpp` para que dibuje `pol.d` en vez de `parabola.d`. En general saldrán resultados distintos, debido a problemas numéricos en la resolución de las ecuaciones normales en `polfit`, a pesar de que los valores del intervalo de generación se han elegido de forma que no haya pérdida de precisión ni por overflows ni underflows. Dismuid el orden de la aproximación, cambiando el primer número de `pol.d`, hasta que `polfit` y `ortopolfit` den el mismo valor de χ^2 con dos cifras decimales, En este orden de ajuste, los problemas de mal condicionamiento empiezan a desaparecer. ¿Que valor M os sale para el orden de ajuste en que `ortopolfit` y `polfit` dan el mismo resultado? Entregad los resultados de `polfit` y `ortopolfit` para n=21, y n=M.