Capítulo 5

Algebra lineal y diagonalización de matrices

Dedicamos este capítulo a la resolución de problemas lineales tales como sistemas de ecuaciones, determinantes, inversión de matrices y diagonalización.

5.1. Eliminación de Gauss y descomposición LU

Como vimos en el tema anterior, en C++ se pueden hacer cálculos de matrices mediante el empleo de arreglos. La forma más conveniente de trabajar es definir los arreglos como punteros simples y dobles para vectores y matrices, respectivamente. Si bien esta no es la forma más cómoda, es la que produce programas con mayor portabilidad. Como ejemplo práctico de la definición de matrices como punteros dobles, damos el programa `gauslu.cpp`, que resuelve un sistema de ecuaciones lineales realizando la eliminación de Gauss con pivotado (y obtiene la descomposición LU de la matriz de los coeficientes como un subproducto) :

```
//Programa eliminacion Gauss   con pivotado: matriz cuadrada y descomposicion LU
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>
using namespace std;

void gaussp(int n, double** m, double* v, double** l){

//Definir vector  permutacion
int* per;
per=new int[n];
for (int i=1;i<n;i++) per[i]=i;
```

```
for (int i=0;i<n; i++){
  l[i][i]=1;
  if (i<n-1) for(int j=i+1; j<n; j++) l[i][j]=0;
  }


//ciclo eliminacion
for (int i=0;i<n;i++){
//search for the pivot
   double piv=abs(m[i][i]);;
   int ipiv=i;
   for (int j=i;j<n;j++){
     if( abs(m[j][i])>piv){
       ipiv=j;
       piv= abs(m[j][i]);
       }
      }
//Permutar filas i y  ipiv
     for (int j=i;j<n;j++) {
     double temp=m[i][j];
     m[i][j]=m[ipiv][j];
     m[ipiv][j]=temp;
     }
     double temp = v[i];
     v[i]=v[ipiv];
     v[ipiv]=temp;

//Permutar vector permutacion
     int itemp=per[i];
     per[i]=per[ipiv];
     per[ipiv]=itemp;

//Eliminacion
  for (int j=i+1; j<n; j++){
//m[j][i] se anula despues de la primera elim.
    double   piv=m[j][i]/m[i][i];
    l[j][i]=piv;
    for (int k=i; k<n; k++){
      m[j][k]=m[j][k]- piv*m[i][k];
      }
      v[j]=v[j]- piv*v[i];
    }
```

```
  }
}

void backsus(int n,double**m, double* v, double* x){
//solucion de un sistema triangular superior por sustitucion hacia atras
for (int i=n-1;i>=0;i--){
   x[i]=v[i];
  for (int j= n-1;j>i;j--)
    x[i]=x[i]-m[i][j]*x[j];
    x[i]=x[i]/m[i][i];
  }
}

int  main(){
//definir data file
ifstream fin("gausslu.dat");
int n;
//leer dimension del sistema
fin>>n;
//Definir matriz  coefficientes
double** mat;
mat = new double* [n];
for (int i=0;i<n;i++) mat[i]=new double [n];
//definir vector terminos independientes
double* b;
b = new double [n];
//Leer matriz coeficient;
for (int i=0; i<n; i++)
  for (int j=0;j<n; j++)
    fin>>mat[i][j];

//Leer vector terminos  independientes
for (int i=0;i<n;i++)
  fin>>b[i];

// Verificar lectura de datos

cout<<"Datos leidos"<<endl;
for (int i=0; i<n; i++){
cout<<endl;
  for (int j=0;j<n; j++)
    cout<<mat[i][j]<<"   ";
   }
```

```
cout <<endl<<endl;

for (int i=0; i<n;i++)
  cout<<b[i]<<"  ";
cout<< endl<<endl;

//Definir matriz L
double** l;
l = new double* [n];
for (int i=0;i<n;i++) l[i]=new double [n];


gaussp(n,mat,b,l);

// Imprimir  sistema reducido
cout<<"Sistema  reducido"<<endl;
 for (int i=0; i<n; i++){
cout<<endl;
  for (int j=0;j<n; j++)
    cout<<mat[i][j]<<"   ";
   }
cout <<endl<<endl;;

for (int i=0; i<n;i++)
  cout<<b[i]<<"  ";
cout<< endl<<endl;;


// Print l
cout<<" Matriz  L"<<endl;
 for (int i=0; i<n; i++){
cout<<endl;
  for (int j=0;j<n; j++)
    cout<<l[i][j]<<"   ";
   }
cout <<endl<<endl;

//Solucion del sistema por  substitucion hacia atras
double* x;
x= new double [n];
backsus(n,mat,b,x);

//Imprimir  solucion
```

```
cout<<"Vector de  soluciones"<<endl;
for (int i=0; i<n;i++)
  cout<<"x["<<i<<"]="<<x[i]<<"  ";
cout<< endl;

return 0;
}
```

Como podemos observar, el volumen de líneas de programación utilizado cuando se definen las matrices mediante punteros es mucho mayor que el que se necesita cuando se utilizan clases de matrices.

Ejercicio 1: Examinad, compilad y ejecutad el programa, resolviendo diferentes sistemas de ecuacciones (modificando el fichero `gausslu.dat`).

5.2. Diagonalización

En el mismo espíritu que el apartado anterior, el programa `jacobi.cpp` realiza la diagonalización de una matriz por el método de Jacobi, utilizando punteros:

```
//Programa de diagonalizacion por Jacobi
//con busqueda del mayor elemento fuera de la diagonal
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>
using namespace std;

//Numero de digitos y precision
const int nwide=10;
const int nprec=6;

void
jacobi(int n, double **m, double **v)
{

//Numero maximo de iteraciones
    int maxiter = 100;

//Tolerancia
    double tol = 1e-12;
```

```
//definir v(matriz vectores propios) como matriz identidad
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            v[i][j] = 0;
        v[i][i] = 1;
    }

//Calcular Delta
    double delta = 0.;

    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            delta = delta + m[i][j] * m[i][j];

    cout << "Delta inicial = " << delta << endl;

//ciclo de iteraciones
    for (int i = 0; i < maxiter; i++)
    {
//Busqueda del mayor elemento fuera de la diagonal
//Esto no es eficiente para grandes matrices. Hacer un barrido en este caso.
        double apq = 0;
        int ip = 0;
        int iq = 1;

        for (int j = 0; j < n; j++)
        {
            for (int k = j + 1; k < n; k++)
            {
                if (abs(m[j][k]) > apq)
                {
                    ip = j;
                    iq = k;
                    apq = abs(m[ip][iq]);

                }
            }
        }
        cout << "p= " << ip << " q= " << iq << " apq= " << apq << endl;

//Determinar c, s, c, tau
        double t = 1;
```

```
        if (m[ip][ip] != m[iq][iq])
        {
            double alpha = (m[iq][iq] - m[ip][ip]) / (2*m[ip][iq]);

            t = -alpha + alpha / abs(alpha) * sqrt(alpha * alpha + 1);
        }
        double c = 1 / sqrt(t * t + 1);
        double s = t * c;
        double tau = s / (1 + c);
        double d = m[ip][iq];

        cout << "  s= " << s << "  c= " << c << " t=" << t << " tau="
              << tau << endl << endl;

//Actualizar matriz m en y por encima de la diagonal
        m[ip][ip] = m[ip][ip] - t * m[ip][iq];
        m[iq][iq] = m[iq][iq] + t * m[ip][iq];

        for (int j = 0; j < ip; j++)
        {
            double temp1 = m[j][ip];
            double temp2 = m[j][iq];

            m[j][ip] = temp1 - s * (temp2 + tau * temp1);
            m[j][iq] = temp2 + s * (temp1 - tau * temp2);
        }
        for (int j = ip + 1; j < iq; j++)
        {
            double temp1 = m[ip][j];
            double temp2 = m[j][iq];

            m[j][iq] = temp2 + s * (temp1 - tau * temp2);
            m[ip][j] = temp1 - s * (temp2 + tau * temp1);
        }
        for (int j = iq + 1; j < n; j++)
        {
            double temp1 = m[ip][j];
            double temp2 = m[iq][j];

            m[iq][j] = temp2 + s * (temp1 - tau * temp2);
            m[ip][j] = temp1 - s * (temp2 + tau * temp1);
        }
```

```
        m[ip][iq] = 0;

//Imprimir m en cada iteracion
        cout << "Matriz iteracion " << i + 1 << endl;
        for (int j = 0; j < n; j++)
        {
            cout << endl;
            for (int k = 0; k <= j; k++)
                cout << setw(nwide) <<  m[k][j] << "   ";
        }
        cout << endl << endl << endl;

//Actualizar v
        for (int j = 0; j < n; j++)
        {
            double temp1 = v[j][ip];
            double temp2 = v[j][iq];

            v[j][ip] = c * temp1 - s * temp2;
            v[j][iq] = s * temp1 + c * temp2;
        }

//Imprimir v

        cout << "V iteracion " << i + 1 << endl;
        for (int j = 0; j < n; j++)
        {
            cout << endl;
            for (int k = 0; k < n; k++)
                cout << setw(nwide) << v[j][k] << "   ";
        }
        cout << endl << endl;

//Actualizar delta
        delta = delta - d * d;

        cout << " Delta= " << delta << endl << endl;

//Si ha convergido actualizar diagonal inferior de m y volver a main
        if (abs(delta) <= tol)
        {
            for (int j = 0; j < n; j++)
                for (int k = j + 1; k < n; k++)
```

```
                    m[k][j] = m[j][k];
            cout << " Jacobi ha convergido  con " << i + 1 <<
                     " iteraciones" << endl << endl;
            cout << "Delta= " << delta << endl << endl;
            return;
        }
    }
//Ha hecho maxiter iteraciones
    cout << "Numero de  maximo de iteraciones en jacobi sin convergencia"
         << endl << endl;
    cout << "  Delta= " << delta << endl << endl;
}


int
main()
{
//Definir data file
    ifstream fin("jacobi.dat");
    int n;

//Leer dimension del sistema(jacobi.dat)
    fin >> n;

//Definir matriz coeficientes

    double **mat;
    mat = new double *[n];

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

//Definir matriz vectores propios
    double **vecp;
    vecp = new double *[n];

    for (int i = 0; i < n; i++)
        vecp[i] = new double[n];

//Elegir matriz en jacobi.dat o matriz de Hilbert

//Leer matriz en jacobi.dat;
    for (int i = 0; i < n; i++)
```

```
        for (int j = 0; j < n; j++)
            fin >> mat[i][j];
/*
//Matriz de Hilbert;
for (int i=0; i<n; i++)
  for (int j=0;j<n; j++)
  mat[i][j]=1/double(i+j+2);
*/
//Verificar lectura datos
    for (int i = 0; i < n; i++)
    {
        cout << endl;
        for (int j = 0; j < n; j++)
            cout << setw(nwide) << mat[i][j] << "   ";
    }
    cout << endl;

    jacobi(n, mat, vecp);

//Imprimir matriz diagonalizada

    cout << "Matriz diagonalizada" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << endl;
        for (int j = 0; j < n; j++)
            cout << setw(nwide) << mat[i][j] << "   ";
    }
    cout << endl << endl;

//Imprimir matriz vectores propios

    cout << "Vectores propios (columnas) " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << endl;
        for (int j = 0; j < n; j++)
            cout << setw(nwide) << vecp[i][j] << "   ";
    }
    cout << endl << endl;

//Comprobacion ortogonalidad vectores propios
```

```
    cout << "Productos escalares de vectores propios" << endl << endl;

    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            double temp = 0;

            for (int k = 0; k < n; k++)
                temp = temp + vecp[k][i] * vecp[k][j];
            cout << setw(nwide) <<temp << "  ";
        }
        cout << endl;
    }
//Liberar memoria
    for (int j = 0; j < n; j++) delete[] mat[j];
    delete[] mat;
    delete[] vecp;
    mat = 0;
    vecp = 0;
    return 0;
}
```

Ejercicio 2: Examinad compilad y ejecutad el programa `jacobi.cpp` para diversas matrices simétricas.

5.3. Clases algoritmos

Junto con la librería TNT, se proporciona la librería JAMA. La librería JAMA es una clase de patrones, y los ficheros que la componen se incluyen con la cabecera `algebralineal.h` utilizada en la práctica anterior. Las funciones de la librería JAMA son accesibles con el namespace CALNUM, al igual que las funciones de la librería TNT. Esta librería proporciona algoritmos para la descomposición por Cholesky, la descomposición LU, y el cálculo de valores propios (aparte de otros algoritmos que no vamos a utilizar). Cada algoritmo es una clase de C++. Por ejemplo, si queremos descomponer una matriz por Cholesky, creamos un algoritmo de la clase Cholesky, que es una clase de patrónes que depende de un parámetro, que es el tipo de los elementos de la matriz. Por ejemplo:

```
Cholesky<double>mi_chol1(A);
```

donde A es la matriz, y `mi_chol1` es el nombre que le doy al objeto particular de la clase del algoritmo de Cholesky, que he creado para trabajar con la matriz A. Si tengo una matriz B del tipo `float` que quiero descomponer LU, creo un elemento de la clase LU:

```
LU<float>mi_lu(B);
```

Si deseo descomponer una matriz C, de elementos del tipo `long double` en valores propios, creo un elemento de la clase `Eigenvalue`:

```
Eigenvalue<long double>mis_valores_propios(C);
```

Lo algoritmos están programados de forma que se verifique si las dimensiones de las matrices asignadas son correctas. Por ejemplo, si intentamos descomponer por Cholesky una matriz no simétrica, se produce un mensaje de error. El dato de cada uno de estos tres algoritmos es la matriz asociada. Con los datos, las clases de algoritmos producen una serie de resultados, utilizando las funciones correspondientes de la clase. Una diferencia esencial entre las clases del C++ y las estructuras del C, es que las clases contienen,además de datos, funciones. Para invocar las funciones de la clase, se escribe el elemento de la clase unido al nombre de la función por un punto. Por ejemplo, si `milu` es un elemento de la clase LU, el determinante se calcula invocando la función `det()` sobre este elemento, lo que se hace escribiendo `milu.det()`. Si queremos guardar el determinante, debemos de hacerlo en una variable del tipo que devuelve la función. Por ejemplo, `det()` devuelve un `double`, por lo que para guardar el determinante tendremos que escribir, por ejemplo,

```
double deter=milu.det();
```

donde `deter` es una variable de tipo `double` que hemos creado para guardar el determinante.

Las funciones de cada una de las clases son:

Cholesky

Si `chol` es un algoritmo de la clase Cholesky y v es un vector y B una matriz, las siguientes funciones están definidas:

1. `chol.getL()`: Devuelve la matriz triangular de Cholesky.
2. `chol.is_spd();` Devuelve 1 si la matriz es definida positiva y 0 en caso contrario. Su uso es, por ejemplo
   ```
   if (chol.is_spd) L= chol.getL();
   else cout<<"error: matriz no definida positiva"<<endl;
   ```
3. `chol.solve(v)`, con v un vector: Devuelve el vector x, solucion de $A*x=v$.
4. `chol.solve(B)`; Da la matriz X, solución de $A*X=B$.

El programa `jama_cholesky.cpp` ilustra el empleo de estas funciones:

```
//Ejemplos con TNT y JAMA
//compilar con g++ -I./templates  jama_cholesky.cpp
#include <iostream>
#include <iomanip>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;
```

```
typedef double  Real; //tipo generico Real

int main(){
cout<< "Entrar dimension"<<endl;
int n;
cin>>n;
Matrix<Real> D(n,n);
Matrix<Real> E(n,n);
Vector<Real> b(n);
 b=1.;


//Choleski. Atencion la matriz debe ser definida positiva
for(int  i=0;i<n;i++)
  for(int j=0;j<n;j++)
   D[i][j]=Real(1.)/Real(i+j+1.);
cout<<D<<endl;
Cholesky<Real> chol(D);
Matrix<Real>M=chol.getL();
Matrix<Real> MT= transpose(M);
cout<<"matriz de Cholesky M"<<endl<<M<<endl;
cout<<"transpuesta M"<<endl<<MT<<endl;
 cout<<"producto cholesky"<<endl<<M*transpose(M)<<endl<<
"matriz inicial"<<endl<<D<<endl;

//Resolucion de ecuaciones
Vector<Real> x= chol.solve(b); //Solucion de A*x=b
cout<<" x= "<< x<<endl;
cout<<"D*x= "<<D*x<<endl; //Comprobacion A*x=b

return 0;
}
```

LU

Si `lu` es un algoritmo de la clase LU, que tiene asignado la matriz A y si v es un vector y B una matriz, las siguientes funciones están definidas:

1. `lu.getL()`: Devuelve la matriz inferior L.
2. `lu.getU()`: Devuelve la matriz superior U.
3. `lu.det()`: Devuelve el determinante.

4. `lu.getPivot()`: Devuelve el vector de pivotes.

5. `lu.solve(v)`: Devuelve el vector x, solución de $A*x=v$.

6. `lu.solve(B)`: Devuelve la matriz X, solución de $A*X=B$.

El programa `jama_lu.cpp` ilustra el empleo de las funciones de la clase LU:

```
//Ejemplos con TNT y JAMA
//compilar con g++ - I./templates jama_lu.cpp
#include <iostream>
#include <iomanip>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;
typedef  double Real; //tipo generico Real

int main()
{
  //Descomposicion LU de una matriz de Hilbert
cout << "Entrar dimension" << endl;
int             n;
cin >> n;
Matrix < Real > H(n, n);
        Vector < Real > b(n);
        b=1.;


//Creacion matriz de Hilbert
        for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
    H[i][j] = Real(1.) / Real(i + j + 1.); //Conversion al tipo Real
cout << H << endl;

//descomposicion LU
        //Creacion objeto llamado mi_lu de la clase de algoritmos LU
        LU < Real > mi_lu(H);
        Matrix < Real > L = mi_lu.getL(); //obtencion matriz L
        cout << L << endl;
Matrix < Real > U = mi_lu.getU(); //Obtencion matriz U
        cout << U << endl;
Real            det = mi_lu.det(); //Obtencion determinante
        cout << "det =" << det << endl;
cout << "L*U = " << L * U << endl;//Producto L * U
```

```
        Matrix < Real > IH = mi_lu.inv(); //Calculo de la inversa
cout << "inversa = " << IH << endl;
Matrix < Real > I = H * IH;
cout << "H*IH = " << I << endl; //comprobacion H * inv(H) = I
        //Solucion de ecuaciones
        Vector < Real > x = mi_lu.solve(b); //Solucion H * x = b;
        cout <<"x = "<<x<<endl;
        return 0;
}
```

Eigenvalue

Si `eig` es un algoritmo de la clase `Eigenvalue` con matriz `A`, y si `B`, `C` y `D` son matrices de las dimensiones adecuadas y `lambda` un vector, las siguientes funciones están definidas:

1. `Eigenvalue <double >eig(A)`: Define un objeto de nombre `eig` perteneciente a la clase `Eigenvalue` y con matriz asociada `A`.
2. `eig.getRealEigenvalues(lambda`: Rellena el vector lambda con las partes reales de los valores propios.
3. `eig.getImagEigenvalues(lambda)`: Rellena el vector lambda con las partes imaginarias de los valores propios.
4. `eig.getD(B)`: Rellena la matriz `B` con la matriz `A` diagonalizada.
5. `eig.getV(C)`: Rellena la matriz `C` con los vectores propios de A, como columnas.

El programa `jama_eigen.cpp` ilustra el uso de las diferentes funciones de la librería JAMA:

```
//Ejemplos con TNT y JAMA
//compilar con g++ - I./templates jama_eigen.cpp
#include <iostream>
#include <iomanip>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;

typedef long double Real;//tipo generico Real

int main()
{
cout << "Entrar dimension" << endl;
int             n;
cin >> n;
```

```
Matrix < Real > H(n, n);
Matrix < Real > E(n, n);

//Creacion matriz Hilbert
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
H[i][j] = Real(1.) / Real(i + j + 1.);
cout << "H=" << H << endl;

//valores y vectores propios
        //Se crea un elemennto val_prop de la clase Eingenvalue
Eigenvalue < Real > val_prop(H);

        //creacion vector de valores propios reales en lambda
        Vector < Real > lambda;

        //relleno de lambda con los valores propios
val_prop.getRealEigenvalues(lambda);
cout << "valores propios=" << lambda << endl;

Matrix < Real > V;//Creacion de matriz de vectores propios

        //vectores propios, se escriben en la matriz V
val_prop.getV(V);//relleno de V

//matriz diagonal, se escribe en D
Matrix < Real > D;

val_prop.getD(D);//relleno de D con la diagonal de H
cout << "Matriz Diagonal= " << D << endl;
cout << " Vectores propios = " << V << endl;

//Comprobacion H * V = V * D
        Matrix < Real > AV = H * V - V * D;

//Comprobacion H * V = V * D
cout << " H*V-V*D= " << AV << endl;
Matrix < Real > I = V * transpose(V);

//Comprobacion ortogonalidad V
cout << "V*transpose(V)=" << I << endl;
I = transpose(V) * H * V;
```

```
//Comprobacion VT * H * V = D
cout << "transpose(V)*H*V=" << I << endl;
        return 0;
}
```

5.3.1. Funciones adicionales de la librería TNT

Con objeto de facilitar la manipulación de matrices, se han definido una serie de funciones que permiten extraer la diagonal, la triangular superior o inferior o una determinada fila o columna. Su empleo se ilustra en el programa `tnt_ejemplos2.cpp`:

```
//Ejemplos funciones adicionales con TNT
//compilar con  g++ - I./templates tnt_ejemplos2.cpp
#include <cmath>
#include <iostream>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;

int main()
{
  int  n; //dimension, que puede ser leida en ejecucion
  cout << "Entrar dimension" << endl;
  cin >> n;

  //Ejempos con matrices y escalares

  //Construccion de tres matrices de double
  Matrix < double >A(n, n), B(n, n), C(n, n);
  for(int i = 0; i < n; i++)
    A[i][i] = double (i); //relleno de la matriz A


//creacion identidad de dimension n
  Matrix<double> I=identity<double>(n);
  cout<<I<<endl;
  B = 2.; //todos los elementos de B = 2.
  cout<<"B= "<<B<<endl;
  C=6./B;
  cout<< "C=6./B "<< C<<endl;


 //extraccion triangular inferior estricta (lower)
```

```
 C=lower(B);
 cout<< "lower(B) ="<<C<<endl;

 //extraccion triangular superior estricta (upper)
 C=upper(B);
 cout<< "upper(B) ="<<C<<endl;

 //extraccion matriz diagonal
 C=diagonal(B);
 cout<< "diagonal(B) ="<<C<<endl;

//division de matrices elemento a elemento
C=upper(B)/B;
cout<<"upper(B)/B= "<<C<<endl;

//normas de matrices
cout<<"normmax(upper(B)/B)= "<<normmax(C)<<endl;
cout<<"norm1(upper(B)/B)= "<<norm1(C)<<endl;
cout<<"normf(upper(B)/B)= "<<normf(C)<<endl; //norma Frobenius

//Extraccion vector diagonal
Vector<double> b= diag(A);
cout<<"diag(A)="<<b<<endl;

//Proyecta fila n de matriz C, empezando por 0  ( row(C,n) )
b=row(C,1);
cout<<b<<endl;
//Proyecta columna n de matriz C, empezando por 0 ( column (C,n) )
b=column(C,1);
cout<<b<<endl;

//Multiplicacion elemento a elemento
cout<<C%B<<endl;

return 0;

}
```

5.4. Raíces de polinomios

Uno de los métodos más eficientes de calcular todas las raíces de un polinomio es diagonalizar una matriz cuya ecuación característica es dicho polinomio. Las raíces son los valores

propios de esta matriz. Si tenemos un polinomio

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

se puede comprobar fácilmente que la siguiente matriz tiene dicho polinomio como ecuación característica:

$$\begin{bmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \cdots & -\frac{a_1}{a_n} & -\frac{a_0}{a_n} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

El programa `raices_pol.cpp` utiliza la clase `Eigenvalue` para calcular las raíces de un polinomio.

```
//Ejemplos con TNT y JAMA
//compilar con g++ -I./templates raices_pol.cpp
#include <iostream>
#include <iomanip>
#include <complex>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;

typedef long double Real;
//tipo generico Real
Real pol(Vector < Real > a, Real x, Real y)
{
//Calculo de un polinomio por el algoritmo de Horner
complex < Real > p = 0.;
int             n = a.size();
complex < Real > z = complex < Real > (x, y);
p = a(n);
for (int i = n - 1; i >= 1; i--)
p = p * z + a(i);
Real            pr = real(p);
return pr;
}


int main()
{
cout << "Entrar dimension" << endl;
```

```
int             n;
cin >> n;
Matrix < Real > A(n, n);
Matrix < Real > E(n, n);
Vector < Real > a(n + 1), lambdar(n), lambdai(n);
//Definicion de a.En caso general leer n y a de fichero
a = 1.;

//Creacion matriz
for (int i = 0; i < n - 1; i++) {
A[0][i] = -a[n - i - 1] / a[n];
A[i + 1][i] = 1.;
}
A[0][n - 1] = -a[0] / a[n];
cout << "A = " << A << endl;

//valores y vectores propios
                //Se crea un elemennto val_prop de la clase Eingenvalue
Eigenvalue < Real > val_prop(A);

                //relleno de lambda con los valores propios
                val_prop.getRealEigenvalues(lambdar);
cout << "parte real raices=" << lambdar << endl;
val_prop.getImagEigenvalues(lambdai);
cout << "parte imaginaria raices=" << lambdai << endl;
//Verificacion de las raices
for (int i = 0; i < n; i++) {
Real            p = pol(a, lambdar[i], lambdai[i]);
cout << p << endl;
}

return 0;
}
```

5.5. Ejercicios

1. Utilizad el programa `gausslu.cpp` para resolver el siguiente sistema de ecuaciones lineales:

$$\begin{aligned} -x_1 + x_2 \quad - 3x_4 &= 4 \\ x_1 + \quad + 3x_3 + x_4 &= 0 \\ x_2 - x_3 - x_4 &= 3 \end{aligned}$$

$$3x_1 + x_3 + 2x_4 = 1$$

2. Escribid un programa que resuelva, utilizando las librerías JAMA y TNT, el siguiente sistema de ecuaciones lineales

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

   El programa debe calcular e imprimir, además, la inversa y el determinante de la matriz de coeficientes, la matriz L, la matriz U, y el producto L*U.

3. Diagonalizad, utilizando el programa `jacobi.cpp`, la matriz simétrica:

$$\begin{bmatrix} 2 & 0 & 1 \\ 0 & 3 & -2 \\ 1 & -2 & -1 \end{bmatrix}$$

   El programa debe de imprimir en fichero los valores y vectores propios.

4. Diagonalizad, utilizando las librerías TNT y JAMA, la matriz:

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix}$$

   Imprimid los valores y vectores propios. Comprobad que para cada valor propio λ, con vector propio v, se satisface:

$$A * v = \lambda v$$

   Comprobad que la matriz V, cuyas columnas son los vectores propios, es ortogonal, es decir,

$$V^T * V = I$$

   Comprobad, además, que

$$V^T * A * V = D$$

   donde D es la matriz diagonalizada.

5. Escribid un programa, al estilo de `gauslu.cpp`, es decir definiendo las matrices como punteros de punteros, que factorice una matriz mediante el algoritmo de Cholesky. Aplicadlo a la matriz del ejercicio anterior. El programa debe de imprimir la matriz triangular M. Comprobad que $M * M^T = A$. Comprobar el resultado con la clase Cholesky de la librería JAMA. Incluid ambos programas, con sus correspondientes resultados, en la memoria de la práctica.

6. Modificad el programa `raices_pol.cpp` para calcular las raíces del polinomio del primer ejercicio de la práctica 3:

$$P(x) = x^5 - x^4 - 5x^3 + 5x^2 + 6x - 6$$

La dimensión del polinomio y los coeficientes deben leerse desde fichero.