

Reconocimiento de dígitos manuscritos a partir de la factorización SVD

Métodos Numéricos, Departamento de Computación, Universidad de Buenos Aires

Guillermo Gallardo Diez, Damian Eliel Aleman, y Luis Scoccola

Reconocimiento de dígitos manuscritos a partir de la factorización SVD

Métodos Numéricos, Departamento de Computación, Universidad de Buenos Aires

ÍNDICE

I. Introducción teórica	2
II. Desarrollo	2
A. Descomposición SVD	2
A.1. Método de la potencia y deflación	3
A.2. Método QR para matrices simétricas tridiagonales y método de la potencia inversa	3
B. Reconociendo dígitos usando las componentes principales	4
B.1. kVecinos	4
B.2. kVecinos ponderados	4
B.3. Distancia al promedio de las componentes	4
C. Tests	5
C.1. Error de los autovectores calculados	5
C.2. Diferencias en el tiempo de ejecución para obtener autovectores	5
C.3. Cantidad de dígitos reconocidos en función de el método, la cantidad de componentes utilizada y la precisión de los autovectores	5
C.4. Influencia del tamaño de la base de datos	6
C.5. Dígitos mejor reconocidos en función del método	6
III. Resultados	6
IV. Discusión	6
V. Conclusiones	6

VI. Apéndices	7
A. Enunciado	7
B. Código relevante y modo de uso	9
B.1. Conversión de datos usando Matlab	9
B.2. Modo de uso del programa	9
B.3. Clase <i>Reconocedor</i>	9
B.4. Clase <i>Matriz</i>	18
C. Tablas	31

Resumen

En el presente trabajo estudiaremos el reconocimiento automatizado de dígitos manuscritos mediante la comparación de las componentes principales de una imagen de un dígito manuscrito con las de una serie de imágenes de entrenamiento. Compararemos performance y calidad de resultados de dos métodos distintos para hallar autovectores de matrices. i?????SLASMDL:KSD

Index Terms

I. INTRODUCCIÓN TEÓRICA

SE llama factorización SVD (*singular value decomposition*) de una matriz A (de dimensiones arbitrarias) a un producto de tres matrices $U\Sigma T^t = A$. Donde U y T son ortogonales y Σ tiene coeficientes nulos en todo elemento $\sigma_{i,j}, i \neq j$ (usualmente llamada diagonal, si bien sus dimensiones son iguales a las de A). Si los coeficientes de la diagonal de Σ tienen la particularidad de estar ordenados de la forma $\sigma_{i,i} \geq \sigma_{j,j}, i < j$, entonces esta factorización es única. Una propiedad importante de esta factorización es que las matrices U y V tienen como columnas los autovectores de AA^t y A^tA respectivamente. Los elementos de la diagonal de Σ se denominan valores singulares de X .

La factorización puede ser interpretada de varias formas y suele brindar mucha información acerca de la matriz. En este caso queremos calcular las componentes principales de un cierto vector que representa una imagen de un dígito manuscrito. Esto lo logramos calculando la factorización SVD de la matriz de covarianza entre píxeles X^tX . Para conseguir la matriz X se parte de una base de datos x_1, \dots, x_n con imágenes de dígitos manuscritos de dimensiones iguales. Luego se interpreta cada imagen x_i como un vector fila y se calcula, índice a índice, el vector promedio μ de todas las imágenes. La matriz X se obtiene al poner como filas los vectores $x_i - \mu$ y dividiendo por $\sqrt{n-1}$.

Las k componentes principales de una imagen dada se obtienen realizando el producto interno entre el vector que representa la imagen y los k autovectores que se corresponden con los k autovalores de mayor módulo de la matriz de covarianza entre píxeles. Es decir, las k componentes principales de una imagen x son los primeros k coeficientes del vector que resulta de multiplicar V^tx , suponiendo que se realizó la factorización SVD de X de manera tal que los valores singulares de X se encuentran ordenados de mayor a menor.

Una vez obtenidas las componentes principales de la imagen que quiere reconocerse, pueden compararse con las componentes principales de los dígitos de la base de datos de diversas maneras. Típicamente se toma la distancia correspondiente a la norma euclídea entre vectores para buscar la imagen cuyas componentes principales se encuentren más cercanas a las de la imagen que quiere reconocerse.

II. DESARROLLO

A. Descomposición SVD

DE la introducción se deduce que no es necesario calcular de forma completa toda la descomposición en valores singulares. Basta tener las primeras k columnas de V . Es decir necesitamos

un método para obtener los autovectores que se corresponden con los autovalores de mayor módulo de una matriz. Es importante notar que la matriz con la que trabajamos es simétrica con coeficientes reales y por lo tanto podremos utilizar métodos optimizados. Para obtener los autovectores implementamos dos estrategias:

- **Método de la Potencia y deflación:** Suponiendo que tenemos autovalores $\lambda_1 \geq \dots \geq |\lambda_n|$ y los primeros k son distintos dos a dos, obtenemos λ_1 (y su autovector asociado), mediante el método de la potencia. Luego construimos una matriz que tenga como autovalores $|\lambda_2| \geq \dots \geq |\lambda_n| \geq 0$. De esta forma inductiva calculamos los primeros k autovalores con sus correspondientes autovectores.
- **Método QR para matrices simétricas tridiagonales y método de la potencia inversa:** Asumiendo que partimos de una matriz simétrica, obtenemos una matriz de hesseberg (y en este caso tridiagonal, por ser simétrica) semejante, mediante reflexiones de Householder. Luego, mediante el algoritmo QR calculamos sus autovalores. Finalmente usamos esta aproximación de los autovalores para calcular los primeros k autovectores con el método de la potencia inversa.

A continuación se explican las implementaciones con más detalle.

A.1 Método de la potencia y deflación

El método de la potencia es un método iterativo. Se parte con un vector inicial v_0 y se genera una sucesión de vectores v_i realizando, en cada iteración,

$$v_{i+1} = \frac{Av_i}{\|Av_i\|_\infty}$$

Asumiendo que la matriz tiene un autovalor $\bar{\lambda}$ de módulo estrictamente mayor al resto de los autovalores y que v_0 no es ortogonal al autovector \bar{v} asociado a dicho autovalor, la sucesión converge a \bar{v} .

El método no depende fuertemente de la norma que se usa para renormalizar en cada iteración. La única diferencia es que el autovector obtenido estará normalizado con la norma usada. En nuestro caso usamos la norma infinito porque ya que se calcula de manera más rápida computacionalmente.

Una vez que tenemos el autovalor $\bar{\lambda}$ obtenemos una matriz A' que conserva los mismos autovalores y autovectores exceptuando $\bar{\lambda}$ y \bar{v} , realizando:

$$A' = A - \bar{\lambda}\bar{v}\bar{v}^t$$

A.2 Método QR para matrices simétricas tridiagonales y método de la potencia inversa

Como la matriz de covarianza es simétrica, utilizamos el algoritmo 9,5 de [2] para obtener una matriz tridiagonal semejante a ésta. El método se basa en transformaciones de Householder. Una vez obtenida usamos el algoritmo 9,6 para calcular sus autovalores. Dado que trabajamos con matrices tridiagonales los autovalores pueden calcularse rápidamente y con mucha precisión.

Luego utilizamos el método inverso de la potencia. Este es bastante similar al método de la potencia pero permite calcular cualquier par autovalor-autovector asumiendo las mismas hipótesis

que el método de la potencia con el agregado de tener una aproximación razonable de los autovalores correspondientes a los autovectores que quieren calcularse.

Partiendo de la matriz A , un autovector v_0 y una aproximación a un autovalor λ obtenemos una sucesión v_i de autovectores que converge al autovector asociado al autovalor λ . Para esto planteamos la iteración:

$$v_{i+1} = (A - \lambda Id)^{-1} v_i$$

Notemos que este es el concepto matemático. En nuestra implementación no invertimos la matriz, sino que resolvemos un sistema de ecuaciones en cada iteración. Para alivianar el tiempo de cómputo realizamos una descomposición LU de la matriz del sistema que nos sirve para todas las iteraciones que se realicen con un mismo autovalor λ .

Dado que se usan transformaciones ortogonales, el método QR suele brindar resultados de mucha calidad.

B. Reconociendo dígitos usando las componentes principales

PARA reconocer dígitos a partir de sus componentes principales implementamos dos métodos. Cada uno de los metodos puede ser usado con la cantidad de componentes principales que se desee y por lo tanto no se hará referencia a dicha cantidad en la explicación. En la parte tests experimentaremos con diversas cantidades y buscaremos la más adecuada para cada método.

B.1 kVecinos

Se calculan las componentes principales c de la imagen a evaluar y las componentes c_i de todas las imágenes x_i de las bases de datos. Se toman las distancias euclídeas $\|c - c_i\|$ y se cuentan las apariciones de cada dígito $(0, \dots, 9)$ en el conjunto de los k c_i más cercanos a c . El dígito que registre mayor frecuencia se tomará como el dígito escrito en la imagen a evaluar.

B.2 kVecinos ponderados

En un principio pensamos calcular el promedio de las distancias $\|c - c_j\|$. Donde c_j representa el conjunto de componentes principales de las imágenes del dígito j . De esta manera se está tomando la distancia promedio a todos los dígitos 1, a los dígitos 2, etc.

Vimos que este método no daba resultados razonables y parametrizamos el mismo, para dar más granularidad a la hora de utilizarlo. En lugar de promediar la distancia a todos los dígitos de la base de datos, lo hacemos únicamente con los k dígitos que se encuentren más cerca.

B.3 Distancia al promedio de las componentes

Para cada dígito $0, \dots, 9$ se iteran todos los c_i y se generan diez \bar{c}_j correspondientes con el promedio componente a componente de los c_i de los dígitos j . Luego se calcula la distancia de c a cada uno de los \bar{c}_j y se toma como dígito escrito al j tal que $\|c - \bar{c}_j\|$ sea mínimo.

C. Tests

C.1 Error de los autovectores calculados

Para los dos métodos utilizados para calcular autovectores diseñamos un experimento que busca cuantificar el error máximo que presentan los resultados. Se calculan los primeros k autovectores v_i y se normalizan con la norma euclídea. Luego se realiza Av_i para cada i y se normaliza el resultado, obteniendo \hat{v}_i . Finalmente se toman las diferencias $\|v_i - \hat{v}_i\|$. Mientras más precisos sean los resultados del algoritmo, más pequeñas serán estas distancias. De esta manera podemos comparar el comportamiento de los dos algoritmos con distintas tolerancias de entrada. La definición anterior esta sujeta a un problema: un autovector se encuentra contenido en una recta y esa recta contiene dos vectores de igual norma. Ambos vectores son autovectores y la única diferencia entre ellos es el sentido. Teniendo esto en consideración podría ocurrir que la distancia calculada esté por arriba que $\sqrt{2}$ e incluso que sea igual a 2^1 . Por este motivo, cuando se obtiene una distancia mayor a $\sqrt{2}$ se multiplica el vector por -1 y se la recalcula.

Esperamos obtener resultados más precisos al utilizar la estrategia *QR-potencia Inversa*. Basamos esta hipótesis en que se usan matrices ortogonales para calcular los autovalores. Estas matrices tienen el número de condición mas bajo posible, 1. Una vez que tenemos aproximaciones de los autovalores de buena calidad el método de la potencia inversa brinda buenas aproximaciones de los autovectores.

C.2 Diferencias en el tiempo de ejecución para obtener autovectores

Dado que ambos métodos son esencialmente distintos cabe preguntarse si sus complejidades son distintas. Teóricamente notamos una diferencia importante: el método de la potencia inversa resuelve un sistema de ecuaciones por cada autovector y por ende tiene una complejidad de $\Omega(n^3)$, donde n representa la dimensión de la matriz. Por otro lado el método de la potencia solo realiza una multiplicación entre matriz y vector y por lo tanto tendrá una complejidad de $\Omega(n^2)$.

Por este motivo esperamos ver una diferencia significativa en el tiempo de ejecución de ambas estrategias.

C.3 Cantidad de dígitos reconocidos en función de el método, la cantidad de componentes utilizada y la precisión de los autovectores

Comenzamos buscando buenos parámetros para los métodos de reconocimiento de dígitos. Para esto calculamos autovectores con la mejor precisión posible y experimentamos con la cantidad de componentes entre 20 y 100. Una vez que obtenemos los parámetros que se comportan mejor para cada método analizamos la diferencia de comportamiento de los métodos al variar la cantidad de componentes principales y la precisión con la cual se calculan los autovectores.

¹ En el caso en que obtengamos el mismo vector pero cambiado de signo.

C.4 Influencia del tamaño de la base de datos

Basándonos en el test anterior comparamos el *hit rate* logrado por los métodos de reconocimiento de dígitos al variar la cantidad de imágenes en la base de datos.

C.5 Dígitos mejor reconocidos en función del método

Dado que implementamos más de un método para el reconocimiento de dígitos, nos preguntamos si los mismos funcionan mejor en distintos dígitos. Para responder al interrogante calculamos el *hit rate* de cada dígito por separado y comparamos los resultados de los distintos métodos.

III. RESULTADOS

IV. DISCUSIÓN

V. CONCLUSIONES

REFERENCIAS

- [1] Cátedra de Métodos numéricos,
Tercer Trabajo Práctico,
Primer cuatrimestre 2013
- [2] Richard L. Burden,
Numerical Analysis,
9th ed.

VI. APÉNDICES

A. Enunciado

Laboratorio de Métodos Numéricos - Primer Cuatrimestre 2013

Trabajo Práctico Número 3: OCR+SVD

Introducción

El reconocimiento óptico de caracteres (OCR, por sus siglas en inglés) es el proceso por el cual se traducen o convierten imágenes de dígitos o caracteres (sean éstos manuscritos o de alguna tipografía especial) a un formato representable en nuestra computadora (por ejemplo, ASCII). Esta tarea puede ser más sencilla (por ejemplo, cuando tratamos de determinar el texto escrito en una versión escaneada a buena resolución de un libro) o tornarse casi imposible (recetas indecifrables de médicos, algunos parciales manuscritos de alumnos de métodos numéricos, etc).

El objetivo del trabajo práctico es implementar un método de reconocimiento de dígitos manuscritos basado en la descomposición en valores singulares, y analizar empíricamente los parámetros principales del método.

Como instancias de entrenamiento, se tiene un conjunto de n imágenes de dígitos manuscritos en escala de grises del mismo tamaño y resolución (varias imágenes de cada dígito). Cada una de estas imágenes sabemos a qué dígito se corresponde. En este trabajo consideraremos la popular base de datos MNIST, utilizada como referencia en esta área de investigación².

Para $i = 1, \dots, n$, sea $x_i \in \mathbb{R}^m$ la i -ésima imagen de nuestra base de datos almacenada por filas en un vector, y sea $\mu = (x_1 + \dots + x_n)/n$ el promedio de las imágenes. Definimos $X \in \mathbb{R}^{n \times m}$ como la matriz que contiene en la i -ésima fila al vector $(x_i - \mu)^t / \sqrt{n-1}$, y

$$X = U\Sigma V^t$$

a su descomposición en valores singulares, con $U \in \mathbb{R}^{n \times n}$ y $V \in \mathbb{R}^{m \times m}$ matrices ortogonales, y $\Sigma \in \mathbb{R}^{n \times m}$ la matriz diagonal conteniendo en la posición (i, i) al i -ésimo valor singular σ_i . Siendo v_i la columna i de V , definimos para $i = 1, \dots, n$ la *transformación característica* del dígito x_i como el vector $\mathbf{tc}(x_i) = (v_1^t x_i, v_2^t x_i, \dots, v_k^t x_i) \in \mathbb{R}^k$, donde $k \in \{1, \dots, m\}$ es un parámetro de la implementación. Este proceso corresponde a extraer las k primeras *componentes principales* de cada imagen. La intención es que $\mathbf{tc}(x_i)$ resuma la información más relevante de la imagen, descartando los detalles o las zonas que no aportan rasgos distintivos.

Dada una nueva imagen x de un dígito manuscrito, que no se encuentra en el conjunto inicial de imágenes de entrenamiento, el problema de reconocimiento consiste en determinar a qué dígito corresponde. Para esto, se calcula $\mathbf{tc}(x)$ y se compara con $\mathbf{tc}(x_i)$, para $i = 1, \dots, n$.

Enunciado

Se pide implementar un programa que lea desde archivos las imágenes de entrenamiento de distintos dígitos manuscritos y que, utilizando la descomposición en valores singulares, se calcule la transformación característica de acuerdo con la descripción anterior. Para ello se deberá implementar

² <http://yann.lecun.com/exdb/mnist/>

algún método de estimación de autovalores/autovectores. Dada una nueva imagen de un dígito manuscrito, el programa deberá determinar a qué dígito corresponde. El formato de los archivos de entrada y salida queda a elección del grupo. Si no usan un entorno de desarrollo que incluya bibliotecas para la lectura de archivos de imágenes, sugerimos que utilicen imágenes en formato RAW.

Se deberán realizar experimentos para medir la efectividad del reconocimiento, analizando tanto la influencia de la cantidad k de componentes principales seleccionadas como la influencia de la precisión en el cálculo de los autovalores.

Fecha de entrega

- *Formato electrónico*: viernes 21 de junio de 2013, hasta las 23:59 hs., enviando el trabajo (informe+código) a `metnum.lab@gmail.com`. El subject del email debe comenzar con el texto [TP3] seguido de la lista de apellidos de los integrantes del grupo.
- *Formato físico*: lunes 24 de junio de 2013, de 18 a 20hs (en la clase de la práctica).

B. Código relevante y modo de uso

B.1 Conversión de datos usando Matlab

Las instancias utilizadas fueron bajadas de `http://yann.lecun.com/exdb/mnist/`. Debe utilizarse el *script* `generarPuntoDat.sh` que se encuentra en `imagenes/MNIST_Matlab` para convertir las instancias bajadas a archivos legibles por nuestra implementación. Al ejecutar el *script* se verá el modo de uso.

B.2 Modo de uso del programa

Si bien al ejecutar el programa `tp3.exe` que se encuentra en `ejecutables/` se imprimen las instrucciones de uso cabe destacar algunos puntos:

B.3 Clase *Reconocedor*

Esta clase permite utilizar cualquiera de los métodos para el calculo de autovalores y combinarlo con cualquiera de los método de reconocimiento de dígitos.

```

1 #include "reconocedor.h"
2
3 #define TAMANO_IMAGEN 784
4 #define MIN_SIGNIFICATIVO 0.01f
5
6 using namespace std;
7
8 // PARA COMPARAR TUPLAS POR LA SEGUNDA COMPONENTE
9 bool compararTupla ( pair<int,double> i, pair<int,double> j ) { return ( i.second < j.second
10 ); }
11
12 Reconocedor::Reconocedor( char *puntoDat )
13 {
14     instanciaAbierta = false;
15     tcsCalculados = false;
16
17     int scan;
18     FILE *data = fopen( puntoDat, "r" );
19
20     scan = fscanf( data, "%d\n", &cantidad );
21
22     imagenes = new Matriz<double>( cantidad, TAMANO_IMAGEN );
23     labels = new int[cantidad];
24     int numerito;
25
26     for ( int i=0 ; i<cantidad ; ++i ) {
27         scan = fscanf( data, "%d ", &numerito );
28         labels[i] = numerito;
29     }
30
31     for ( int i=0 ; i<cantidad ; ++i ) {
32         for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
33             scan = fscanf( data, "%d ", &numerito );
34             (*imagenes)[i][j] = (double) numerito;
35         }
36     }
37
38     fclose( data );
39

```

```

40     Matriz<double> X (*imagenes);
41
42
43     // tomamos promedio
44     double promedio[TAMANO_IMAGEN];
45     // arreglo limpio
46     memset( promedio, 0, sizeof(double)*TAMANO_IMAGEN );
47
48     for ( int i=0 ; i<cantidad ; ++i )
49         for ( int j=0 ; j<TAMANO_IMAGEN ; ++j )
50             promedio[j] += (*imagenes)[i][j];
51
52     double cant = (double) TAMANO_IMAGEN;
53
54     for ( int i=0 ; i<TAMANO_IMAGEN ; ++i )
55         promedio[i] /= cant;
56
57     double denominador = sqrt( cant - 1.f );
58
59     // calculo matriz X
60
61     for ( int i=0 ; i<cantidad ; ++i )
62         for ( int j=0 ; j<TAMANO_IMAGEN ; ++j )
63             (*imagenes)[i][j] = ((*imagenes)[i][j] - promedio[j]) / denominador;
64
65     covarianza = new Matriz<double> (TAMANO_IMAGEN, TAMANO_IMAGEN);
66     covarianza->cargarTranspuestaPorMat(X);
67
68
69     promediosCalculados = false;
70
71     cantAutovectores = 0;
72 }
73
74 Reconocedor::Reconocedor( char *puntoDat, char *matCovarianza )
75 {
76     instanciaAbierta = false;
77     tcsCalculados = false;
78
79     int scan;
80     FILE *data = fopen( puntoDat, "r" );
81
82     scan = fscanf( data, "%d\n", &cantidad );
83
84     imagenes = new Matriz<double>( cantidad, TAMANO_IMAGEN );
85     labels = new int[cantidad];
86
87     int numerito;
88
89     for ( int i=0 ; i<cantidad ; ++i ) {
90         scan = fscanf( data, "%d ", &numerito );
91         labels[i] = numerito;
92     }
93
94     for ( int i=0 ; i<cantidad ; ++i ) {
95         for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
96             scan = fscanf( data, "%d ", &numerito );
97             (*imagenes)[i][j] = (double) numerito;
98         }
99     }
100
101     fclose( data );
102
103
104     //matriz de covarianza
105
106     FILE *dataCov = fopen( matCovarianza, "r" );

```

```

107
108     int dim1, dim2;
109     scan = fscanf( dataCov, "%d %d", &dim1, &dim2 );
110
111     assert( dim1==TAMANO_IMAGEN && dim2==TAMANO_IMAGEN );
112
113     covarianza = new Matriz<double> (dim1, dim2);
114
115     double numeritoDoubel;
116     for ( int i=0 ; i<TAMANO_IMAGEN; ++i ) {
117         for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
118             scan = fscanf( dataCov, "%lf ", &numeritoDoubel );
119             (*covarianza)[i][j] = numeritoDoubel;
120         }
121     }
122
123     fclose( dataCov );
124
125     promediosCalculados = false;
126
127     cantAutovectores = 0;
128 }
129
130 Reconocedor::~Reconocedor()
131 {
132     delete imagenes;
133     delete[] labels;
134
135     delete covarianza;
136
137     if (cantAutovectores>0) delete autovectores;
138
139     if (tcsCalculados) delete tcs;
140
141     if (instanciaAbierta) {
142         delete aEvaluar;
143         delete[] labels_aEvaluar;
144     }
145
146     if (promediosCalculados)
147         delete promediosTcs;
148 }
149
150 void Reconocedor::guardarCovarianza( char *nombre )
151 {
152     FILE *guardar = fopen(nombre, "w");
153
154     //tamano de la matriz
155     fprintf(guardar, "%d %d\n", covarianza->cantFil(), covarianza->cantCol());
156
157     //coeficientes de la matriz
158     for ( uint i=0 ; i<covarianza->cantFil() ; ++i ) {
159         for ( uint j=0 ; j<covarianza->cantCol() ; ++j )
160             fprintf(guardar, "%d ", (int) (*covarianza)[i][j]);
161
162         fprintf(guardar, "\n");
163     }
164
165     fclose(guardar);
166 }
167
168 void Reconocedor::abrir_instancia_a_evaluar( char *archivo, int primero, int ultimo )
169 {
170     FILE *data = fopen( archivo, "r" );
171     int scan;
172     scan = fscanf( data, "%d\n", &cantidad );
173

```

```

174     if ( ultimo > cantidad ) {
175         printf("pediste mas indices de los que hay, esto va a explotar\n");
176         return;
177     }
178
179     int cuantosPidio = ultimo - primero + 1;
180
181     aEvaluar = new Matriz<double>( cuantosPidio, TAMANO_IMAGEN );
182     labels_aEvaluar = new int[cuantosPidio];
183
184     int numerito;
185
186     int indice = 0;
187     for ( int i=0 ; i<cantidad ; ++i ) {
188         scan = fscanf( data, "%d ", &numerito );
189
190         if ( i+1 >= primero && i+1 <= ultimo ) {
191             labels_aEvaluar[indice] = numerito;
192             indice++;
193         }
194     }
195     indice = 0;
196     for ( int i=0 ; i<cantidad ; ++i ) {
197         for ( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
198             scan = fscanf( data, "%d ", &numerito );
199
200             if ( i+1 >= primero && i+1 <= ultimo ) {
201                 (*aEvaluar)[indice][j] = (double) numerito;
202             }
203         }
204         if ( i+1 >= primero && i+1 <= ultimo )
205             indice++;
206     }
207
208     fclose( data );
209
210     instanciaAbierta = true;
211
212     calcular_tcs();
213 }
214
215 void Reconocedor::calcularAutovectores_QR( int maxIterQR, int maxIterInvPotencia, double
    tolerancia, int cuantosAutovec )
216 {
217
218     // printf("Me pidieron esta tolerancia: %f\n", tolerancia);
219
220     if ( (uint) cuantosAutovec > covarianza->cantFil() ) {
221         printf("pediste mas autovalores que la cantidad de dimensiones de la matriz, esto
            va a explotar\n");
222         return;
223     }
224
225     cantAutovectores = cuantosAutovec;
226
227     autovectores = new Matriz<double> (TAMANO_IMAGEN, cantAutovectores);
228
229     Matriz<double> temp (*covarianza);
230     //temp.contieneNaN();
231     temp.Householder( tolerancia );
232     //temp.contieneNaN();
233
234     vector<double> autoval;
235     int iter = maxIterQR;
236     temp.QR( tolerancia, iter, autoval );
237
238     //ya tengo los autovalores

```

```

239     sort(autoval.begin(),autoval.end() );
240
241     //   for ( int i=0 ; i<autoval.size() ; ++i ) {
242     //       printf("%f\n", autoval[i]);
243     //   }
244
245     Matriz<double> x(TAMANO_IMAGEN, 1);
246
247     for (int i=0 ; i<cuantosAutovec ; ++i ) {
248         // agarro los autovalores de menor a mayor!
249         double autovalActual = autoval[autoval.size()-i-1];
250
251         if ( fabs(autovalActual) < MIN_SIGNIFICATIVO ) break;
252
253         //   printf("%f %f\n", autovalActual, autoval[autoval.size()-i-2]);
254         //   __BITACORA
255
256
257         // le paso un autovector que este lejos de 0
258         for( int j=0 ; j<TAMANO_IMAGEN; ++j ) {
259             x[j][0] = 1.f;
260         }
261
262         covarianza->potenciaInversa(autovalActual, x, tolerancia, maxIterInvPotencia);
263     x.normalizar();
264
265         // lo guardo en mi matriz de autovectores
266         for( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
267             (*autovectores)[j][i] = x[j][0];
268         }
269     }
270 }
271
272 void Reconocedor::calcularAutovectores_potencia( int maxIterPotSim, double tolerancia, int
    cuantosAutovec )
273 {
274     if ( (uint) cuantosAutovec > covarianza->cantFil() ) {
275         printf("pediste mas autovalores que la cantidad de dimensiones de la matriz, esto
            va a explotar\n");
276         return;
277     }
278
279     double autovalActual;
280
281     autovectores = new Matriz<double> (TAMANO_IMAGEN, cuantosAutovec);
282
283     Matriz<double> autovectorActual(TAMANO_IMAGEN, 1);
284
285     for( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
286         autovectorActual[j][0] = 1.f;
287     }
288
289
290     for (int i=0 ; i<cuantosAutovec ; ++i ) {
291
292         covarianza->potenciaSimple(autovalActual,autovectorActual,tolerancia,
            maxIterPotSim);
293
294         if ( fabs(autovalActual) < MIN_SIGNIFICATIVO )
295             break;
296         else
297             cantAutovectores++;      //calculo un autovector
298
299
300         // lo guardo en mi matriz de autovectores
301         for( int j=0 ; j<TAMANO_IMAGEN ; ++j ) {
302             (*autovectores)[j][i] = autovectorActual[j][0];

```



```

303         }
304
305         covarianza->deflacion(autovalActual, autovectorActual);
306     }
307 }
308
309 int Reconocedor::reconocer_kVecinos( int cantComponentes, int k, int indice_imagen )
310 {
311     if (!instanciaAbierta) {
312         printf("No se elegio instancia a evaluar\n");
313         return -1;
314     }
315
316     if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
317         printf("pediste una imagen fuera de rango\n");
318         return -1;
319     }
320
321     if ( cantComponentes > cantAutovectores ) {
322         printf("pediste mas componenes principales que la cantidad de autovectores que
323             calculaste\n");
324         return -1;
325     }
326
327     //me lo dan partiendo de 1
328     //pero yo lo uso desde 0
329     indice_imagen--;
330
331     int cantIm = imagenes->cantFil();
332
333     vector< pair<int,double> > distancias;
334
335     Matriz<double> resta(cantComponentes,1);
336
337     for ( int i=0 ; i<cantIm ; ++i ) {
338         for( int j=0 ; j<cantComponentes ; ++j )
339             resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*tcs)[i][j];
340
341         double distancia = resta.norma();
342
343         pair<int,double> labelDistancia;
344         labelDistancia.first = labels[i];
345         labelDistancia.second = distancia;
346
347         // printf("label %d, a distancia %f\n", labelDistancia.first, labelDistancia.second)
348         ;
349         distancias.push_back(labelDistancia);
350     }
351
352
353     sort( distancias.begin(), distancias.end(), compararTupla );
354
355     vector<int> frecuencias(10, 0);
356
357     for (int i=0 ; i<k ; ++i ) {
358         frecuencias[ distancias[i].first ]++;
359     }
360
361     int mejor = 0;
362     int digito = -1;
363
364     for (int i=0 ; i<10 ; ++i ) {
365         // printf("%d ", frecuencias[i]);
366         if ( frecuencias[i] > mejor ) {
367             mejor = frecuencias[i];

```

```

368         digito = i;
369     }
370 }
371
372     return digito;
373 }
374
375 int Reconocedor::reconocer_distanciaMedia( int cantComponentes, int indice_imagen )
376 {
377     if (!instanciaAbierta) {
378         printf("No se elegio instancia a evaluar\n");
379         return -1;
380     }
381
382     if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
383         printf("pediste una imagen fuera de rango\n");
384         return -1;
385     }
386
387     if ( cantComponentes > cantAutovectores ) {
388         printf("pediste mas componenes principales que la cantidad de autovectores que
389             calculaste\n");
390         return -1;
391     }
392
393     indice_imagen--;
394
395     int cantIm = imagenes->cantFil();
396
397     vector<double> distanciasMedias(10, 0.f);
398     vector<int> cantApariciones(10, 0);
399
400     Matriz<double> resta(cantComponentes,1);
401
402     for ( int i=0 ; i<cantIm ; ++i ) {
403
404         for( int j=0 ; j<cantComponentes ; ++j )
405             resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*tcs)[i][j];
406
407         double distancia = resta.norma();
408
409         //aparecion un a vez mas
410         cantApariciones[ labels[i] ]++;
411         //con esta distancia
412         distanciasMedias[ labels[i] ] += distancia;
413     }
414
415     double mejor = INFINITY;
416     int digito = -1;
417
418     for (int i=0 ; i<10 ; ++i ) {
419         printf("%d ", frecuencias[i]);
420
421         double mediaActual = distanciasMedias[i]/(double)cantApariciones[i];
422         if ( mediaActual < mejor ) {
423             mejor = mediaActual;
424             digito = i;
425         }
426     }
427
428     return digito;
429 }
430
431 int Reconocedor::reconocer_kVecinosPonderados( int cantComponentes, int k, int indice_imagen
432 )
433 {

```

```

433     if (!instanciaAbierta) {
434         printf("No se elegio instancia a evaluar\n");
435         return -1;
436     }
437
438     if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
439         printf("pediste una imagen fuera de rango\n");
440         return -1;
441     }
442
443     if ( cantComponentes > cantAutovectores ) {
444         printf("pediste mas componenes principales que la cantidad de autovectores que
445             calculaste\n");
446         return -1;
447     }
448
449     //me lo dan partiendo de 1
450     //pero yo lo uso desde 0
451     indice_imagen--;
452
453     int cantIm = imagenes->cantFil();
454
455     //first LABEL
456     //second DISTANCIA
457     vector< pair<int,double> > distancias;
458
459     Matriz<double> resta(cantComponentes,1);
460     for ( int i=0 ; i<cantIm ; ++i ) {
461         for( int j=0 ; j<cantComponentes ; ++j )
462             resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*tcs)[i][j];
463
464         double distancia = resta.norma();
465
466         pair<int,double> labelDistancia;
467         labelDistancia.first = labels[i];
468         labelDistancia.second = distancia;
469
470         distancias.push_back(labelDistancia);
471     }
472
473
474     sort( distancias.begin(), distancias.end(), compararTupla );
475
476     vector<int> frecuencias(10, 0);
477     vector<double> distanciasMedias(10, 0.f);
478
479     for (int i=0 ; i<k ; ++i ) {
480         int labelActual = distancias[i].first;
481
482         frecuencias[ labelActual ]++;
483         distanciasMedias[ labelActual ] += distancias[i].second;
484     }
485
486     double mejor = INFINITY;
487     int digito = -1;
488
489     for (int i=0 ; i<10 ; ++i ) {
490
491         // printf("%d ", frecuencias[i]);
492         double mediaActual = (frecuencias[i]==0)? INFINITY : distanciasMedias[i]/(double)
            frecuencias[i];
493
494         if ( mediaActual < mejor ) {
495             mejor = mediaActual;
496             digito = i;
497         }

```

```

498     }
499
500     return digito;
501 }
502
503 int Reconocedor::reconocer_digitoMedio( int cantComponentes, int indice_imagen )
504 {
505     if (!instanciaAbierta) {
506         printf("No se elegio instancia a evaluar\n");
507         return -1;
508     }
509
510     if (!promediosCalculados) {
511         printf("No promediaste antes de llamar a la funcion, esto va a explotar\n");
512         return -1;
513     }
514
515     if ( (uint) indice_imagen > aEvaluar->cantFil() ) {
516         printf("pediste una imagen fuera de rango\n");
517         return -1;
518     }
519
520     if ( cantComponentes > cantAutovectores ) {
521         printf("pediste mas componenes principales que la cantidad de autovectores que
522             calculaste\n");
523         return -1;
524     }
525
526     indice_imagen--;
527
528     vector<double> distanciasMedias(10, 0.f);
529
530     Matriz<double> resta(cantComponentes,1);
531
532     for ( int i=0 ; i<10 ; ++i ) {
533         for( int j=0 ; j<cantComponentes ; ++j ) {
534             resta[j][0] = (*tcs_aEvaluar)[indice_imagen][j] - (*promediosTcs)[i][j];
535         }
536
537         double distancia = resta.norma();
538
539         //con esta distancIA
540         distanciasMedias[i] = distancia;
541     }
542
543     double mejor = INFINITY;
544     int digito = -1;
545     for (int i=0 ; i<10 ; ++i ) {
546
547         double mediaActual = distanciasMedias[i];
548
549         //printf("%f ", distanciasMedias[i]);
550         if ( mediaActual < mejor ) {
551             mejor = mediaActual;
552             digito = i;
553         }
554     }
555     //printf("\n");
556
557     return digito;
558 }
559
560 void Reconocedor::promediarTcs( int cantComponentes )
561 {
562     promediosTcs = new Matriz<double> (10, cantComponentes);
563

```

```

564 //inicializo
565 for ( uint i=0 ; i<10 ; ++i )
566     for ( int j=0 ; j<cantComponentes ; ++j )
567         (*promediosTcs)[i][j] = 0.f;
568
569
570 vector<int> apariciones(10, 0);
571
572 int cuantasImagenes = tcs->cantFil();
573
574 for ( int i=0 ; i<cuantasImagenes ; ++i ) {
575     for ( int j=0 ; j<cantComponentes ; ++j ) {
576         (*promediosTcs)[ labels[i] ][j] += (*tcs)[i][j];
577         apariciones[ labels[i] ]++;
578     }
579 }
580
581
582 for ( int i=0 ; i<10 ; ++i ) {
583     for ( int j=0 ; j<cantComponentes ; ++j ) {
584         if (apariciones[i] == 0) {
585             (*promediosTcs)[i][j] = INFINITY;
586         } else {
587             (*promediosTcs)[i][j] /= apariciones[i];
588         }
589     }
590 }
591
592 promediosCalculados = true;
593 }
594
595 void Reconocedor::calcular_tcs()
596 {
597     int cantIm = imagenes->cantFil();
598     int componentes = cantAutovectores;
599
600     tcs = new Matriz<double>( cantIm, componentes );
601
602     int cantImEval = aEvaluar->cantFil();
603
604     tcs_aEvaluar = new Matriz<double>( cantImEval, componentes );
605
606     tcs->cargarMultiplicacion( *imagenes, *autovectores );
607     tcs_aEvaluar->cargarMultiplicacion( *aEvaluar, *autovectores );
608
609     tcsCalculados = true;
610 }

```

B.4 Clase *Matriz*

```

1  #ifndef __MATRIZ_H
2  #define __MATRIZ_H
3
4  #include "master_header.h"
5
6  #define COMPARAR_DOUBLE(a, b) (fabs( (a) - (b) ) < 1e-9)
7  #define SIGNO_DOUBLE( a ) ((fabs( (a) ) > 1e-9) ? ((a) > 0) ? 1.f : -1.f) : 1.f)
8
9  template< typename T >
10 class Matriz
11 {
12     private:
13         uint n,m;
14         uint triangulada;
15

```

```

16 public:
17     T ** matriz;
18
19     Matriz(int _n, int _m) : n(_n), m(_m)
20     {
21         matriz = new T*[n];
22         for( uint i = 0; i < n; ++i)
23             matriz[i] = new T[m];
24         triangulada = 0;
25     }
26
27     Matriz( Matriz<T> &B )
28     {
29         n = B.n;
30         m = B.m;
31         triangulada = 0;
32         uint i, j;
33
34         matriz = new T*[n];
35
36         for( i=0 ; i<n ; ++i )
37             matriz[i] = new T[m];
38
39         for ( i=0 ; i<n ; ++i )
40             for ( j=0 ; j<m ; ++j )
41                 matriz[i][j] = B[i][j];
42     }
43
44
45     ~Matriz()
46     {
47         for(uint i = 0; i < n; ++i)
48             delete [] matriz[i];
49
50         delete [] matriz;
51     }
52
53     uint cantFil() { return n; }
54     uint cantCol() { return m; }
55
56     T* operator[](uint i)
57     {
58         if(i >= n) return NULL;
59         return matriz[i];
60     }
61
62     void cargarSuma( Matriz<T> &A, Matriz<T> &B)
63     {
64         assert( n==A.n && m==A.m && A.n==B.n && A.m==B.m );
65
66         uint i, j;
67         for ( i=0 ; i<n ; ++i )
68             for ( j=0 ; j<m ; ++j )
69                 matriz[i][j] = A[i][j] + B[i][j];
70     }
71
72     void cargarResta( Matriz<T> &A, Matriz<T> &B)
73     {
74         assert( n==A.n && m==A.m && A.n==B.n && A.m==B.m );
75
76         uint i, j;
77         for ( i=0 ; i<n ; ++i )
78             for ( j=0 ; j<m ; ++j )
79                 matriz[i][j] = A[i][j] - B[i][j];
80     }
81
82     void restarle( Matriz<T> &A )

```

```

83     {
84         assert( n==A.n && m==A.m );
85
86         uint i,j;
87         for ( i=0 ; i<n ; ++i )
88             for ( j=0 ; j<m ; ++j )
89                 matriz[i][j] -= A[i][j];
90     }
91
92
93 void cargarMultiplicacion( Matriz<T> &A, Matriz<T> &B )
94 {
95     assert( n==A.n && A.m==B.n && B.m==m );
96
97     uint i,j,k;
98
99     for ( i=0 ; i<n ; ++i ) {
100         for ( j=0 ; j<m ; ++j ) {
101             double sum = 0;
102             for ( k=0 ; k<A.m ; ++k )
103                 sum += (double) A[i][k] * (double) B[k][j];
104
105             matriz[i][j] = (T) sum;
106         }
107     }
108 }
109
110 void cargarTranspuestaPorMat( Matriz<T> &A )
111 {
112     uint i,j,k;
113     for ( i=0 ; i<A.m ; ++i ) {
114         for ( j=0 ; j<A.m ; ++j ) {
115             double sum = 0.f;
116             for ( k=0 ; k<A.n ; ++k )
117                 sum += (double) A[k][i] * (double) A[k][j];
118
119             matriz[i][j] = (T) sum;
120         }
121     }
122 }
123
124 void transponer( Matriz<T> &B )
125 {
126     assert( n==B.m && m==B.n );
127
128     uint i,j;
129
130     for ( i=0 ; i<n ; ++i )
131         for ( j=0 ; j<m ; ++j )
132             matriz[i][j] = B[j][i];
133 }
134
135 void imprimirMatriz()
136 {
137     uint i,j;
138
139     for ( i=0 ; i<n ; ++i ) {
140         for ( j=0 ; j<m ; ++j )
141             printf("%f\t", (double) matriz[i][j]);
142
143         printf("\n");
144     }
145
146     printf("\n");
147 }
148
149 void contieneNaN()

```

```

150 {
151     uint i,j;
152
153     for ( i=0 ; i<n ; ++i )
154         for ( j=0 ; j<m ; ++j )
155             assert( !(matriz[i][j]!=matriz[i][j]));
156 }
157
158 void guardarArchivo(char * archivo)
159 {
160     //DEBUG - NO ME SAQUES
161
162     FILE *nuevo = fopen( archivo , "w" );
163
164     int i;
165
166     fprintf( nuevo, "%d\n", n );
167
168     for ( i=0 ; i<n ; ++i )
169         fprintf( nuevo, "%f ", matriz[i][0] );
170
171     fprintf( nuevo, "\n" );
172
173     fclose( nuevo );
174 }
175
176 void Householder ( double tol )
177 {
178     // la matriz de entrada es la que llama a la funcion, se llama matriz
179     std::vector<double> v(n, 0.f);
180     std::vector<double> u(n, 0.f);
181     std::vector<double> z(n, 0.f);
182     std::vector<double> y(n, 0.f);
183
184     for (uint k = 0; k < n-2 ; k++) // reveer los indices hasta donde van
185     {
186         double q = 0.f;
187         for (uint j=k+1; j<n ; j++) {
188             q += ( matriz[j][k])*(matriz[j][k]) );
189         }
190
191         // si la columna son todos ceros,
192         // no hay nada que hacer
193         // paso a la proxima
194         if( COMPARAR_DOUBLE(q, 0.f) ) continue;
195
196         // ahora q es la sumatoria
197
198         double alfa = - sqrt(q) * SIGNO_DOUBLE(matriz[k+1][k]);
199
200         double rsq = alfa*alfa - alfa * matriz[k+1][k];
201
202         v[k] = 0.f;
203         v[k+1] = matriz[k+1][k] - alfa;
204
205         for (uint j = k+2; j<n; j++)
206             v[j] = matriz[j][k];
207
208         double suma;
209         for (uint j=k; j<n ; j++)
210         {
211             suma=0.f;
212             for (uint i = k+1; i<n ; i++)
213                 suma += matriz[j][i] * v[i];
214
215             u[j] = (1.f/rsq) * suma;
216         }
217
218         double prod = 0.f;

```



```

217         for (uint i=k+1; i<n ; i++)
218             prod += v[i] * u[i];
219
220         for (uint j=k; j<n ; j++)
221             z[j] = u[j] - (prod/(2.f*rsq)) * v[j];
222
223         for (uint l=k+1; l<n-1 ; l++)
224         {
225             for (uint j=l+1; j<n ; j++)
226             {
227                 matriz[j][l] = matriz[j][l] - v[l] * z[j] - v[j] * z[l];
228                 matriz[l][j] = matriz[j][l];
229             }
230
231             matriz[l][l] = matriz[l][l] - 2.f * v[l] * z[l];
232         }
233
234         matriz[n-1][n-1] = matriz[n-1][n-1] - 2.f * v[n-1] * z[n-1];
235
236         for (uint j=k+2; j<n ; j++)
237         {
238             matriz[k][j] = 0.f;
239             matriz[j][k] = 0.f;
240         }
241
242         matriz[k+1][k] = matriz[k+1][k] - v[k+1] * z[k];
243         matriz[k][k+1] = matriz[k+1][k];
244     }
245 }
246
247 void submatriz(int desde1, int hasta1, int desde2, int hasta2, Matriz<T> &salida)
248 {
249     for (int i = desde1-1; i < hasta1 ; i++)          //hago desde-1 para que se
250                                                         corresponda con los indices de la bibliografia
251         for (int j = desde2-1; j < hasta2; j++)
252             salida[i-(desde1-1)][j-(desde2-1)] = matriz [i][j];
253 }
254
255 void dameDiagonales( std::vector<double> &As, std::vector<double> &Bs )
256 {
257     As.push_back( matriz[0][0] );
258     Bs.push_back( 0.f );
259
260     // printf("%f\n",matriz[0][0] );
261
262     for ( uint i=1 ; i<n ; ++i ) {
263         As.push_back(matriz[i][i]);
264         Bs.push_back(matriz[i][i-1]);
265         // printf("%f\t%f\n",matriz[i][i], matriz[i][i-1]);
266     }
267 }
268
269 void QR( double tol, int maxIter, std::vector<double> &autoval )
270 {
271     std::vector<double> As;
272     std::vector<double> Bs;
273
274     dameDiagonales(As,Bs);
275
276     double shift = 0.f;
277
278     QR_rec( As, Bs, tol, maxIter, autoval, shift );
279 }
280
281 void QR_rec( std::vector<double> &As, std::vector<double> &Bs, double tol, int &
282             maxIter, std::vector<double> &autoval, double shift )

```

```

282 {
283     int tam = As.size();
284
285     ///#if DEBUG
286     //printf("llamado con diag de tamano: %d\n", n);
287     ///#endif
288
289     std::vector<double> Cs(tam, 0.f);    // cosetamos
290     std::vector<double> Ds(tam, 0.f);
291     std::vector<double> Qs(tam, 0.f);
292     std::vector<double> Rs(tam, 0.f);
293     std::vector<double> Ss(tam, 0.f);    // senos
294     std::vector<double> Xs(tam, 0.f);
295     std::vector<double> Ys(tam, 0.f);
296     std::vector<double> Zs(tam, 0.f);
297
298     while ( maxIter > 0 ) {
299
300         if (tam==0) return;
301
302
303         // busco la seguidilla mas larga de ceros
304         // en la subdiagonal empezando desde el final
305
306         while ( fabs(Bs[tam-1])<tol && tam>0 ) {
307             // el ultimo b es suficientemente chico, tengo un autoval
308             autoval.push_back( As[tam-1]+shift );
309             tam--;
310         }
311
312         if (tam==0) return;
313
314         // busco la seguidilla mas larga de ceros
315         // en la subdiagonal empezando desde el principio
316
317         int inicio = 1;
318         while ( fabs(Bs[inicio]) < tol && inicio<tam) {
319             // el primer b es suficientemetne chico, tengo un autoval
320             autoval.push_back( As[inicio-1]+shift );
321             inicio++;
322         }
323         if (inicio==tam) return;
324
325         ///////// reacomodo
326         inicio--;
327         if ( inicio>0 ) {
328             for ( int j=0; j<tam-inicio ; ++j ) {
329                 As[j] = As[j+inicio];
330                 Bs[j] = Bs[j+inicio];
331             }
332         }
333         tam -= inicio;
334
335
336         ////////// casos Base
337         if ( tam==0 ) return;
338
339         if ( tam==1 ) {
340             autoval.push_back( As[0] + shift );
341             return;
342         }
343
344         ////////// posible llamado recursivo
345         for ( int j=2 ; j<tam-1 ; ++j ) {
346             if ( fabs(Bs[j]) < tol ) {
347
348                 //si tengo un numero pequeno parto la matriz

```

```

349                                     //para seguir teniendo una convergencia rapida
350
351                                     //construimos los As y Bs para el llamado recursivo
352                                     std::vector<double> As1;
353                                     std::vector<double> Bs1;
354                                     std::vector<double> As2;
355                                     std::vector<double> Bs2;
356
357                                     for ( int k=0 ; k<tam ; ++k ) {
358
359                                         if ( k<j-1 ) {
360                                             As1.push_back(As[k]);
361                                             Bs1.push_back(Bs[k]);
362                                         } else {
363                                             As2.push_back(As[k]);
364                                             Bs2.push_back(Bs[k]);
365                                         }
366                                     }
367
368                                     QR_rec( As1, Bs1, tol, maxIter, autoval, shift );
369                                     QR_rec( As2, Bs2, tol, maxIter, autoval, shift );
370
371                                     return;
372                                 }
373
374
375
376                                     //////////// iteracion QR
377                                     double b = -(As[tam-2] + As[tam-1]);
378                                     double c = As[tam-1]*As[tam-2] - Bs[tam-1]*Bs[tam-1];
379                                     double d = sqrt(b*b - 4.f*c);
380
381                                     double mu1, mu2;
382                                     if ( !COMPARAR_DOUBLE(b,0.f) && b>0.f ) {
383                                         mu1 = -2.f*c / (b+d);
384                                         mu2 = -(b+d)/2.f;
385                                     } else {
386                                         mu1 = (d-b)/2.f;
387                                         mu2 = 2.f*c / (d-b);
388                                     }
389
390                                     //caso Base n=2
391                                     if ( tam==2 ) {
392                                         autoval.push_back( mu1 + shift );
393                                         autoval.push_back( mu2 + shift );
394                                         return;
395                                     }
396
397                                     double sigma;
398                                     sigma = (fabs(mu1-As[tam-1]) < fabs(mu2-As[tam-1])) ? mu1 : mu2;
399
400                                     shift += sigma;
401
402                                     for ( int j=0 ; j<tam ; ++j )
403                                         Ds[j] = As[j] - sigma;
404
405                                     Xs[0] = Ds[0];
406                                     Ys[0] = Bs[1];
407
408                                     for ( int j=1 ; j<tam ; ++j ) {
409                                         Zs[j-1] = sqrt( Xs[j-1]*Xs[j-1] + Bs[j]*Bs[j]);
410
411                                         Cs[j] = Xs[j-1] / Zs[j-1];
412
413                                         Ss[j] = Bs[j] / Zs[j-1];
414
415                                         Qs[j-1] = Cs[j] * Ys[j-1] + Ss[j] * Ds[j];

```

```

416         Xs[j] = -Ss[j] * Ys[j-1] + Cs[j] * Ds[j];
417
418         if ( j != tam ) {
419             Rs[j-1] = Ss[j] * Bs[j+1];
420             Ys[j] = Cs[j] * Bs[j+1];
421         }
422     }
423
424     Zs[tam-1] = Xs[tam-1];
425
426     As[0] = Ss[1] * Qs[0] + Cs[1] * Zs[0];
427
428     Bs[1] = Ss[1] * Zs[1];
429
430     for ( int j=1 ; j<tam-1 ; ++j ) {
431         As[j] = Ss[j+1] * Qs[j] + Cs[j] * Cs[j+1] * Zs[j];
432         Bs[j+1] = Ss[j+1] * Zs[j+1];
433     }
434
435     As[tam-1] = Cs[tam-1] * Zs[tam-1];
436
437     maxIter--;
438 }
439
440 //         printf("Paso el max de iteraciones y QR no converge\n");
441     }
442
443 void multiplicarEscalar ( double beta)
444 {
445     for (uint i = 0; i < n ; i++)
446         for (uint j = 0; j < m ; j++)
447             matriz[i][j] = matriz[i][j] * beta;
448 }
449
450 void copiar (Matriz<T> &entrada)
451 {
452     for (uint i = 0; i < n ; i++)
453         for (uint j = 0; j < m ; j++)
454             matriz[i][j] = entrada[i][j];
455 }
456
457 void identidad( double factor )
458 {
459     for (uint i = 0; i < n ; i++)
460         for (uint j = 0; j < m ; j++)
461             matriz[i][j] = (i==j) ? factor : 0.f;
462 }
463
464 double distanciaAutovector( Matriz<T> & autovector ){
465
466     assert( autovector.cantFil() == n && autovector.cantCol() == 1 );
467
468     Matriz<double> autov(autovector);
469     Matriz<double> multi(autovector);
470     Matriz<double> autovN(autovector);
471
472     autov.normalizar();
473     autovN.normalizar();
474
475     multi.cargarMultiplicacion(*this, autov);
476     multi.normalizar();
477
478     multi.cargarResta(autov,multi);
479
480     double rtrn = multi.norma();
481
482     if( rtrn > sqrt(2) ){

```

```

483         multi.cargarMultiplicacion(*this, autovN);
484         multi.normalizar();
485         multi.multiplicarEscalar(-1.);
486
487         multi.cargarResta(autovN,multi);
488
489         rtrn = multi.norma();
490     }
491
492     return rtrn;
493 }
494
495 double norma()
496 {
497     double res = 0;
498     for (uint i = 0; i < n; i++)
499         res += matriz[i][0] * matriz[i][0];
500
501     return sqrt(res);
502 }
503
504 void normalizar()
505 {
506     double norm = this->norma();
507     this->multiplicarEscalar( 1.f/norm );
508 }
509
510 double normaInf()
511 {
512     double res = fabs(matriz[0][0]);
513     for (uint i = 0; i < n; i++){
514         res = ( res > fabs(matriz[i][0]) ) ? res : fabs(matriz[i][0]);
515     }
516     return res;
517 }
518
519 int menorP()
520 {
521     int p = 0;
522     double res = matriz[0][0];
523     for (uint i = 0; i<n; i++){
524         if ( fabs(res) < fabs(matriz[i][0]) )
525         {
526             res = matriz[i][0];
527             p = i;
528         }
529     }
530     return p;
531 }
532
533 void deflacion(double autovalor, Matriz<T> &autovector)
534 {
535
536     int n = autovector.cantFil();
537     Matriz<double> temp(n,n);
538
539     Matriz<double> autovectorN( autovector );
540     Matriz<double> autovectorNT(1,n);
541
542     autovectorN.normalizar();
543
544     autovectorNT.transponer(autovectorN);
545
546     temp.cargarMultiplicacion(autovectorN,autovectorNT);
547     temp.multiplicarEscalar(autovalor);
548
549     this->cargarResta(*this, temp);

```

```

550     }
551
552 void potenciaSimple(double &guess, Matriz <T> &x, double tol, int maxIter )
553 {
554     assert( x.cantFil() == n && x.cantCol()==1 );
555
556     Matriz<double> resta(n,1);
557     Matriz<double> mult(1,1);
558     Matriz<double> y(n,1);
559
560     int p = x.menorP();
561     double xp = x[p][0];
562     x.multiplicarEscalar(1.f/xp);
563
564     double mu_0 = 0;
565     double mu_1 = 0;
566     double mu_s = 0;
567
568     uint k = 0;
569     while (k<n){
570
571         y.cargarMultiplicacion(*this,x);
572
573         p = y.menorP();
574
575         double mu = y[p][0];
576         mu_s = mu_0 - (mu_1 - mu_0)*(mu_1 - mu_0)/( mu - 2*mu_1 + mu_0 );
577
578         // printf("matrix Y");
579         y.multiplicarEscalar(1.f/mu);
580
581         resta.cargarResta(x,y);
582
583         x.copiar( y );
584         double err = resta.normaInf();
585
586         // printf("%f\n", err);
587
588         if ( fabs(err)<fabs(tol) && k >= 4 )
589         {
590             guess = mu_s;
591             //printf("Termino Bien en %d iteraciones \n", k);
592             return;
593         }
594
595         k++;
596         mu_0 = mu_1;
597         mu_1 = mu;
598     }
599
600     // printf("se llego a la maxima cant de iteraciones");
601
602     guess = mu_s;
603 }
604
605 void potenciaInversa(double &guess, Matriz <T> &x, double tol, int maxIter )
606 {
607     assert( x.cantFil() == n && x.cantCol()==1 );
608
609     Matriz<double> resta(n,1);
610     Matriz<double> mult(1,1);
611     Matriz<double> y(n,1);
612     Matriz<double> P(n,n); // asumo que P, L y U son cuadradas
613     Matriz<double> L(n,n);
614     Matriz<double> U(n,n);
615     Matriz<double> B(n,n);
616     Matriz<double> qident(n,n);

```

```

617
618
619         qident.identidad( guess );
620
621 //         x.imprimirMatriz();
622
623         int p = x.menorP();
624         double xp = x[p][0];
625         x.multiplicarEscalar(1.f/xp);
626
627         B.cargarResta(*this,qident);
628         B.factorizacionLU(P,L,U);
629
630         uint k = 0;
631         while (k<n) {
632
633             // resuelvo el sistema de ecuaciones
634             y.resolverLU(x,P,L,U);
635
636             p = y.menorP();
637             double mu = y[p][0];
638
639             y.multiplicarEscalar(1.f/mu);
640             resta.cargarResta(x,y);
641
642             x.copiar( y );
643             double err = resta.normaInf();
644
645 //             printf("%f\n", err);
646
647             if ( fabs(err)<fabs(tol) )
648             {
649                 // printf("i: %d - e: %f\n",k,err);
650                 mu = 1.f / mu + guess;
651                 guess = mu;
652
653                 //printf("Itere %d veces\n", k);
654
655                 return;
656             }
657
658             k++;
659         }
660 //         printf("se llego a la maxima cant de iteraciones\n");
661     }
662
663     void factorizacionLU( Matriz<T> &P, Matriz<T> &L, Matriz<T> &U )
664     {
665         uint i,k,j;
666
667         for(uint i=0;i<n;i++){
668             for(k=0;k<n;k++){
669                 U[i][k] = matriz[i][k];
670                 P[i][k] = (i==k) ? 1.f : 0.f;
671             }
672         }
673
674         for (i=0;i<n;i++){
675
676             U.pivoteoParcial(i,P,L);
677             L[i][i]=1.f;
678             for(j=i+1;j<n;j++){
679                 assert( !COMPARAR_DOUBLE( U[i][i], 0.f ) );
680                 double mji = U[j][i] / U[i][i];
681                 for (k=i+1;k<n;k++){
682                     U[j][k] = (double)U[j][k] - mji * (double)U[i][k];
683

```

```

684         }
685         U[j][i] = 0.f;
686         L[j][i] = (T) mji;
687     }
688 }
689
690
691 void resolverLU(Matriz<T> &entrada, Matriz<T> &P, Matriz<T> &L, Matriz<T> &U)
692 {
693     // Tengo  $A x = b$ , como  $P A = L U$ , tengo  $L U x = P t b$ 
694     // sea  $Y = U x$ , entonces  $L Y = P t b$ , lo resuelvo con SustitucionAtras
695     // despues para obtener  $x$  tengo que  $U x = Y$ , lo resuelvo con forward
        substitution
696
697     Matriz<double> Y(n,1);
698     Matriz<double> res(n,1);
699     Matriz<double> b2(entrada.cantFil(), entrada.cantCol());
700     /***** Transpongo P y lo multiplico por b *****/
701
702     b2.cargarMultiplicacion(P, entrada);
703
704
705     /*****Hago  $L Y = b2$  *****/
706     forwardSubstituion(L, b2, Y);
707
708     /*****Hago  $U x = Y$  *****/
709     backwardSubstitution(U, Y, res);
710
711     for(uint i=0; i<n; i++){
712         matriz[i][0] = res[i][0];
713     }
714 }
715
716 void forwardSubstituion( Matriz<T> &L, Matriz<T> &B ,Matriz<T> &res )
717 {
718     Matriz<double> resD(res.n, res.m);
719     //resD[0][0]=B[0][0];
720
721     for( uint f=0; f < n ; f++ ){
722         resD[f][0] = B[f][0];
723         for(uint j=0; j<f; j++){
724             assert( !COMPARAR_DOUBLE(L[f][f], 0.f) );
725             resD[f][0] -= L[f][j]*resD[j][0];
726         }
727         resD[f][0] /= L[f][f];
728     }
729
730     for(uint i=0; i<res.n; i++){
731         res[i][0] = (T) resD[i][0];
732     }
733 }
734
735 void backwardSubstitution( Matriz<T> &U, Matriz<T> &B ,Matriz<T> &res )
736 {
737     Matriz<double> resD(res.n, res.m);
738     resD[n-1][0] = B[n-1][0]/U[n-1][n-1];
739
740     int f = n -2;
741
742     for( f=n-2; f >= 0 ; f-- ){
743         resD[f][0] = B[f][0];
744         for(uint j=f+1; j<n; j++){
745             assert( !COMPARAR_DOUBLE(U[f][f], 0.f) );
746             resD[f][0] -= U[f][j]*resD[j][0];
747         }
748         resD[f][0] /= U[f][f];
749     }

```



```

750
751         for(uint i=0;i<res.n;i++){
752             res[i][0] = (T) resD[i][0];
753         }
754     }
755
756     void pivoteoParcial( int i,Matriz<T> &P,Matriz<T> &L)
757     {
758         uint k;
759         int max=i;
760
761         for (k=i;k<n;k++){
762             max = (fabs(matriz[k][i]) > fabs(matriz[max][i])) ? k :max;
763         }
764
765         if( max != i ){
766
767             for (k=0;k<n;k++){
768                 //cambio de filas
769                 T temp=matriz[i][k];
770                 matriz[i][k]= matriz[max][k];
771                 matriz[max][k] = temp;
772
773                 temp=P[i][k];
774                 P[i][k]= P[max][k];
775                 P[max][k] = temp;
776
777                 temp=L[i][k];
778                 L[i][k]= L[max][k];
779                 L[max][k] = temp;
780             }
781         }
782     }
783 };
784
785 #endif

```

C. Tablas