

```

1  #ifndef _TP1_MATRIX_H_
2  #define _TP1_MATRIX_H_ 1
3
4  #include <stdexcept>
5  #include <cassert>
6  #include <utility>
7  #include <iostream>
8  #include "BDouble.h"
9
10 enum Solutions {
11     INFINITE,
12     SINGLE,
13     NONE
14 };
15
16 // Este magic number nos dice cuándo convertir automáticamente una matriz banda en una matriz normal.
17 #define MAGIC_NUMBER 562154
18
19 /*
20 * Matriz Banda.
21 */
22
23 class Matrix {
24     friend std::ostream &operator<<(std::ostream &, const Matrix &);
25
26 public:
27     Matrix(const Matrix &m)
28         : N(m.rows()), M(m.columns()), uband(m.upper_bandwidth()), lband(m.lower_bandwidth()) {
29         int bound = this->lower_bandwidth() + this->upper_bandwidth() + 1;
30         this->matrix = new BDouble *[this->rows()];
31
32         for (int i = 0; i < this->rows(); ++i) {
33             this->matrix[i] = new BDouble[bound];
34
35             for (int j = 0; j < bound; ++j) {
36                 this->matrix[i][j] = m.matrix[i][j];
37             }
38         }
39     }
40
41     Matrix(int N, int M, int lband = MAGIC_NUMBER, int uband = MAGIC_NUMBER)
42         : N(N), M(M), uband(uband), lband(lband) {
43
44
45         if (this->rows() == 0 || this->columns() == 0) {
46             throw new std::out_of_range("Invalid matrix dimension");
47         }
48
49         if (lband > N) {
50             this->lband = N - 1;
51         }
52
53         if (uband > M) {
54             this->uband = M - 1;
55         }
56
57         int bound = this->lower_bandwidth() + this->upper_bandwidth() + 1;
58         this->matrix = new BDouble *[this->rows()];
59
60         for (int i = 0; i < this->rows(); ++i) {
61             this->matrix[i] = new BDouble[bound];
62
63             for (int j = 0; j < bound; ++j) {
64                 this->matrix[i][j] = 0.0;
65             }
66         }
67     }
68
69     inline int rows() const {
70         return this->N;
71     }
72
73     inline int columns() const {
74         return this->M;
75     }
76
77     inline int upper_bandwidth() const {
78         return this->uband;
79     }
80
81     inline int lower_bandwidth() const {
82         return this->lband;
83     }
84
85     BDouble &operator()(const int &i, const int &j) {

```

```

86     if (i >= this->rows() || j >= this->columns()) {
87         throw new std::out_of_range("Index access out of range");
88     }
89
90     if (i <= j + this->lower_bandwidth() && j <= i + this->upper_bandwidth()) {
91         return matrix[i][j - i + this->lower_bandwidth()];
92     } else {
93         throw new std::out_of_range("Out of modifiable range");
94     }
95 }
96
97 const BDouble &operator()(const int &i, const int &j) const {
98     if (i >= this->rows() || j >= this->columns()) {
99         throw new std::out_of_range("Index access out of range");
100     }
101
102     if (i > j + this->lower_bandwidth()) {
103         return zero;
104     } else if (j > i + this->upper_bandwidth()) {
105         return zero;
106     } else {
107         return matrix[i][j - i + this->lower_bandwidth()];
108     }
109 }
110
111 ~Matrix() {
112     for (int i = 0; i < this->rows(); ++i) {
113         delete[] this->matrix[i];
114     }
115
116     delete[] this->matrix;
117 }
118
119 private:
120     // Matrix
121     int N;
122     int M;
123     int uband;
124     int lband;
125     BDouble **matrix;
126 };
127
128 std::ostream &operator<<(std::ostream &os, const Matrix &m) {
129     for (int i = 0; i < m.rows(); ++i) {
130         for (int j = 0; j < m.columns(); ++j) {
131             os << m(i, j) << " ";
132         }
133
134         os << std::endl;
135     }
136
137     os << std::endl;
138
139     return os;
140 }
141
142 /*****
143  * Acá empieza la parte de resolver los sistemas.
144  *****/
145
146 // m tiene que estar triangulada
147 // el usuario libera la memoria
148 std::pair<BDouble *, enum Solutions> backward_substitution(const Matrix &m, BDouble *b) {
149     BDouble *x = new BDouble[m.columns()];
150     enum Solutions solution = SINGLE;
151
152     int N = std::min(m.rows(), m.columns());
153
154     for (int d = N - 1; d >= 0; d--) {
155         if (m(d, d) == 0.0) {
156             x[d] = 1.0;
157             solution = INFINITE;
158         } else {
159             int bound = std::min(m.columns(), d + m.upper_bandwidth() + 1);
160             x[d] = b[d];
161
162             for (int j = d + 1; j < bound; ++j) {
163                 x[d] -= m(d, j) * x[j];
164             }
165
166             x[d] /= m(d, d);
167         }
168     }
169
170     return std::pair<BDouble *, enum Solutions>(x, solution);

```

```

172 }
173
174 // m tiene que estar triangulada
175 // el usuario libera la memoria
176 std::pair<BDouble *, enum Solutions> forward_substitution(const Matrix &m, BDouble *b) {
177     BDouble *x = new BDouble[m.columns()];
178     enum Solutions solution = SINGLE;
179
180     int N = std::min(m.rows(), m.columns());
181
182     for (int d = 0; d < N; ++d) {
183         if (m(d, d) == 0.0) {
184             x[d] = 1.0;
185             solution = INFINITE;
186         } else {
187             int bound = std::max(0, d - m.lower_bandwidth() - 1);
188             x[d] = b[d];
189
190             for (int j = bound; j < d; ++j) {
191                 x[d] -= m(d, j) * x[j];
192             }
193
194             x[d] /= m(d, d);
195         }
196     }
197
198     return std::pair<BDouble *, enum Solutions>(x, solution);
199 }
200
201 std::pair<BDouble *, enum Solutions> gaussian_elimination(Matrix workspace, BDouble *b) {
202     int diagonal = std::min(workspace.columns(), workspace.rows());
203
204     for (int d = 0; d < diagonal; ++d) {
205         // Tenemos algo distinto de cero en la base
206         for (int i = d + 1; i < std::min(workspace.rows(), d + workspace.lower_bandwidth() + 1); ++i) {
207             // Tenemos algo distinto de cero en alguna fila más abajo
208             BDouble coefficient = workspace(i, d) / workspace(d, d);
209
210             // Realizamos el mismo cambio en la solución del sistema
211             b[i] -= coefficient * b[d];
212
213             for (int j = d + 1; j < std::min(d + workspace.upper_bandwidth() + 1, workspace.columns()); ++j) {
214                 // Realizamos la resta a toda la fila.
215                 workspace(i, j) -= coefficient * workspace(d, j);
216             }
217
218             // Setear esto en 0 debería reducir el error posible (por ejemplo, restando números muy chicos)
219             workspace(i, d) = 0.0;
220         }
221     }
222
223     return backward_substitution(workspace, b);
224 }
225
226
227 std::pair<Matrix, Matrix> LU_factorization(const Matrix &A) {
228     // El tamaño de la diagonal
229     int N = std::min(A.rows(), A.columns());
230
231     // Matriz L triangular inferior, U triangular superior
232     Matrix L(A.rows(), A.columns(), A.lower_bandwidth(), 0);
233     Matrix U(A.rows(), A.columns(), 0, A.upper_bandwidth());
234
235
236     //std::cout << "A" << std::endl;
237     //std::cout << A << std::endl;
238     //std::cout << "L" << std::endl;
239     //std::cout << "rows: " << L.rows() << std::endl;
240     //std::cout << "columns: " << L.columns() << std::endl;
241     //std::cout << "upper_band: " << L.upper_bandwidth() << std::endl;
242     //std::cout << "lower_band: " << L.lower_bandwidth() << std::endl;
243     //std::cout << L << L.lower_bandwidth() << std::endl;
244     //std::cout << "U" << std::endl;
245     //std::cout << "rows: " << U.rows() << std::endl;
246     //std::cout << "columns: " << U.columns() << std::endl;
247     //std::cout << "upper_band: " << U.upper_bandwidth() << std::endl;
248     //std::cout << "lower_band: " << U.lower_bandwidth() << std::endl;
249     //std::cout << U << std::endl;
250     // Las inicializamos como la matriz identidad
251     for (int i = 0; i < N; ++i) {
252         L(i, i) = 1.0;
253         U(i, i) = 1.0;
254     }
255
256     // Elegimos, arbitrariamente, que  $L(0,0) * U(0,0) = A(0,0)$ 
257     U(0, 0) = A(0, 0);

```

```

258 L(0, 0) = 1.0;
259
260 if (U(0, 0) == 0.0) {
261     // No podemos factorizar
262     throw new std::out_of_range("Factorization impossible");
263 }
264
265 //Set first row of U and first column of L
266 int M = std::min(A.upper_bandwidth(), A.lower_bandwidth());
267 //std::cout << "First cycle init..." << std::endl;
268 for (int i = 1; i <= M; i++) {
269     //std::cout << "i: " << i << std::endl;
270     U(0, i) = A(0, i) / L(0, 0);
271     L(i, 0) = A(i, 0) / U(0, 0);
272 }
273 //std::cout << "First cycle end..." << std::endl;
274
275 //Set rows/columns from 1 to n-1
276 //std::cout << "second cycle init..." << std::endl;
277 for (int i = 1; i < N - 1; i++) {
278     //std::cout << "i: " << i << std::endl;
279     U(i, i) = A(i, i);
280     //Aprovechamos banda
281     int bound = std::min(A.lower_bandwidth(), A.upper_bandwidth());
282     for (int h = 1; h <= bound; h++) {
283         //std::cout << "h: " << h << std::endl;
284         if (i >= h) {
285             U(i, i) -= L(i, i - h) * U(i - h, i);
286         }
287     }
288     U(i, i) /= L(i, i);
289
290     if (U(i, i) == 0.0) {
291         // No podemos factorizar
292         throw new std::out_of_range("Factorization impossible");
293     }
294
295     //Estamos abusando de que las matrices van a tener la misma banda
296     //inferior y superior... esto podria no ser asi.
297     int M = std::min(A.upper_bandwidth(), A.lower_bandwidth());
298     for (int k = 1; k <= M; k++) {
299         int j = i + k;
300         //std::cout << "k: " << k << std::endl;
301         if (j < U.columns()) {
302             //std::cout << "A(" << i << ", " << j << ") " << std::endl;
303             U(i, j) = A(i, j);
304         }
305
306         if (j < L.rows()) {
307             //std::cout << "A(" << j << ", " << i << ") " << std::endl;
308             L(j, i) = A(j, i);
309         }
310     }
311
312
313     //Aprovechamos banda
314     int bound = std::min(A.lower_bandwidth(), A.upper_bandwidth());
315     //std::cout << "bound: " << bound << std::endl;
316     for (int h = 1; h <= bound - k; h++) {
317         //std::cout << "h: " << h << std::endl;
318         if (i >= h) {
319             if (j < U.columns()) {
320                 //std::cout << "U(" << i << ", " << j << ") " << std::endl;
321                 //std::cout << "L(" << i << ", " << i-h << ") " << std::endl;
322                 //std::cout << "U(" << i-h << ", " << j << ") " << std::endl;
323                 U(i, j) -= L(i, i - h) * U(i - h, j); // i° ROW OF U
324             }
325             if (j < L.rows()) {
326                 //std::cout << "L(" << j << ", " << i << ") " << std::endl;
327                 //std::cout << "L(" << j << ", " << i-h << ") " << std::endl;
328                 //std::cout << "U(" << i-h << ", " << i << ") " << std::endl;
329                 L(j, i) -= L(j, i - h) * U(i - h, i); // j° COLUMN OF L
330             }
331         }
332     }
333 }
334 //std::cout << "cycle end..." << std::endl;
335
336 if (j < U.columns()) {
337     //std::cout << "U(" << i << ", " << j << ") " << std::endl;
338     U(i, j) /= L(i, i);
339 }
340 if (j < L.rows()) {
341     //std::cout << "L(" << j << ", " << i << ") " << std::endl;
342     L(j, i) /= U(i, i);
343 }

```

```

344     }
345 }
346 }
347 }
348 //std::cout << "second cicle end..." << std::endl;
349
350 //Set last position
351 U(N - 1, N - 1) = A(N - 1, N - 1);
352
353 int bound = std::min(A.lower_bandwidth(), A.upper_bandwidth());
354 for (int h = 1; h <= bound; h++) {
355     U(N - 1, N - 1) -= L(N - 1, N - 1 - h) * U(N - 1 - h, N - 1);
356 }
357 U(N - 1, N - 1) /= L(N - 1, N - 1);
358
359 return std::pair<Matrix, Matrix>(L, U);
360 }
361
362 /**
363  * - A matrix original del sistema.
364  * - L matriz triangular inferior de la descomposicion.
365  * - U matriz triangular superior de la descomposicion.
366  * - i fila del elemento a modificar.
367  * - j columna del elemento a modificar.
368  * - a nuevo valor de la posicion (i, j)
369  */
370 std::pair<BDouble *, enum Solutions> sherman_morrison(Matrix &A, Matrix &L, Matrix &U,
371 int i, int j, BDouble a, BDouble *b) {
372     int N = std::min(A.rows(), A.columns());
373
374     //Sherman-Morrison formula vectors
375     //Altered system: A2 = (A + uv')
376     BDouble *u = new BDouble[N];
377     BDouble *v = new BDouble[N];
378
379     for (int k = 0; k < N; k++) {
380         u[k] = 0.0;
381         v[k] = 0.0;
382     }
383     //Column vector
384     u[i] = 1.0;
385
386     //Row vector
387     v[j] = a - A(i, j);
388
389     //From Sherman-Morrison
390     // A^-1 b = y <=> Ay = b
391     // A^-1 u = z <=> Az = u
392
393     //First we solve:
394     // L y2 = b and L z2 = u
395     BDouble *y2;
396     BDouble *z2;
397
398     std::pair<BDouble *, enum Solutions> solution;
399     solution = forward_substitution(L, b);
400     y2 = solution.first;
401     solution = forward_substitution(U, u);
402     z2 = solution.first;
403
404     //Then we solve:
405     // U y = y2 and U z = z2
406     BDouble *y;
407     BDouble *z;
408     solution = backward_substitution(L, y2);
409     y = solution.first;
410     solution = backward_substitution(U, z2);
411     z = solution.first;
412     delete[] y2;
413     delete[] z2;
414
415     //Finally x = y - z * [(v' y)/(1 + v' z)]
416     BDouble *x = new BDouble[N];
417
418     //First we calculate k = (v' y)/(1 + v' z) (scalar value)
419     BDouble vy = 0.0;
420     BDouble vz = 1.0;
421     for (int h = 0; h < N; h++) {
422         vy += v[h] * y[h];
423         vz += v[h] * z[h];
424     }
425
426     //Finally we calculate x = y + z * k
427     for (int h = 0; h < N; h++) {
428         x[h] = y[h] - (z[h] * (vy / vz));
429     }

```

```

430     delete[] y;
431     delete[] z;
432
433     return std::pair<BDouble *, enum Solutions>(x, SINGLE);
434 }
435
436 std::pair<BDouble *, enum Solutions> sherman_morrison(
437     Matrix &L,
438     Matrix &U,
439     BDouble *u,
440     BDouble *v, BDouble *b) {
441     int N = std::min(L.rows(), L.columns());
442
443     //Sherman-Morrison formula vectors
444     //Altered system: A2 = (A + uv')
445
446     //From Sherman-Morrison
447     // A^-1 b = y <=> Ay = b
448     // A^-1 u = z <=> Az = u
449
450     //First we solve:
451     // L y2 = b and L z2 = u
452     BDouble *y2;
453     BDouble *z2;
454
455     std::pair<BDouble *, enum Solutions> solution;
456     solution = forward_substitution(L, b);
457     y2 = solution.first;
458     solution = forward_substitution(L, u);
459     z2 = solution.first;
460
461     //Then we solve:
462     // U y = y2 and U z = z2
463     BDouble *y;
464     BDouble *z;
465     solution = backward_substitution(U, y2);
466     y = solution.first;
467     solution = backward_substitution(U, z2);
468     z = solution.first;
469     delete[] y2;
470     delete[] z2;
471
472     //Finally x = y - z * [(v' y)/(1 + v' z)]
473     BDouble *x = new BDouble[N];
474
475     //First we calculate k = (v' y)/(1 + v' z) (scalar value)
476     BDouble vy = 0.0;
477     BDouble vz = 1.0;
478     for (int h = 0; h < N; h++) {
479         vy += v[h] * y[h];
480         vz += v[h] * z[h];
481     }
482     BDouble k = (vy / vz);
483
484     //Finally we calculate x = y - z * k
485     for (int h = 0; h < N; h++) {
486         x[h] = y[h] - (z[h] * k);
487     }
488     delete[] y;
489     delete[] z;
490
491     return std::pair<BDouble *, enum Solutions>(x, SINGLE);
492 }
493
494
495 #endif // _TP1_MATRIX_H_

```