```cpp
#include <chrono>
#include "LinearAlgebra.h"
#include "Counter.h"

std::vector<double> operator*(const Matrix &m, const std::vector<double> &n) {
    if (m.columns() > n.size()) {
        std::stringstream fmt;
        fmt << "Tamaño de matriz M es " << m.columns() << ", mientras que vector n es " << n.size();
        throw new std::out_of_range(fmt.str());
    }

    std::vector<double> output(m.rows(), 0.0);

    for (int i = 0; i < m.rows(); ++i) {
        for (int j = 0; j < m.columns(); ++j) {
            output[i] += m(i, j) * n[j];
        }
    }

    return output;
}

/**
* Obtiene el autovalor dominante en modulo de una matriz.
*
* @param A matriz para buscar autoespacios
* @param x vector inicial del algoritmo
* @param norm norma vectorial para actualizar los valores
* @param condition condicion para verificar la convergencia del metodo
*/
EigenPair powerIteration(const Matrix &A, std::vector<double> eigenVector, const Norm &norm, unsigned int iterations) {
    if (A.columns() != A.rows()) {
        throw new std::invalid_argument("La matriz no es cuadrada en el método de la potencia");
    }

    if (iterations <= 0) {
        throw new std::invalid_argument("La cantidad de iteraciones para el método de la potencia debe ser mayor a 0");
    }

    Timer timer("Power Iteration Timer");
    Counter iteration("Power Iteration Iteration Counter");

    double length = norm(eigenVector);

    // Normalizamos el autovector
    for (int j = 0; j < A.rows(); ++j) {
        eigenVector[j] /= length;
    }

    // Verificamos convergencia
    while (iteration < iterations) {
        // Elevamos a potencia
        std::vector<double> temp(eigenVector); // realizamos la copia para luego calcular un delta.

        eigenVector = A * eigenVector;

        // Normalizamos el vector
        length = norm(eigenVector);

        for (int j = 0; j < A.rows(); ++j) {
            eigenVector[j] /= length;
        }


        // Actualizamos el contador
        ++iteration;

        // Verificamos si estamos convergiendo.
        for (int j=0; j < A.rows(); j++)
            temp[j] -= eigenVector[j];
        if (norm(temp) < POWER_ITERATION_DELTA)
            break;
    }

    double eigenValue = 0.0;

```

```
77      for (int i = 0; i < A.rows(); ++i) {
78          for (int j = 0; j < A.rows(); ++j) {
79              eigenValue += eigenVector[i] * eigenVector[j] * A(i, j);
80          }
81      }
82
83      eigenValue /= norm(eigenVector);
84
85      return std::pair<double, std::vector<double>>(eigenValue, eigenVector);
86  }
87
88  /**
89  * Corre deflacion sobre la matriz, devuelve una nueva matriz, que retiene los autovectores de la matriz A,
    excepto por
90  * el autovector de autovalor dominante.
91  *
92  * @param A matriz inicial para la deflacion
93  * @param eigen par de autovalor y autovector
94  * @return nueva matriz con las caracteristicas anunciadas
95  */
96  void deflation(Matrix &A, const EigenPair &eigen) {
97      if (A.columns() != A.rows()) {
98          throw new std::runtime_error("La matriz no es cuadrada en el método de deflación");
99      }
100
101     Timer timer("Deflation Timer");
102
103     for (int i = 0; i < A.rows(); ++i) {
104         for (int j = 0; j < A.columns(); ++j) {
105             A(i, j) -= eigen.first * eigen.second[i] * eigen.second[j];
106         }
107     }
108 }
109
110 /**
111 * Obtiene los primeros k autovectores y autovalores dominantes de la matriz.
112 *
113 * @param A matriz a investigar
114 * @param k cantidad de autovectores/autovalores a obtener
115 * @ret lista ordenada por dominancia decreciente del autopar
116 */
117 std::list<EigenPair> decompose(Matrix deflated, int k, const Norm &norm, unsigned int iterations) {
118     if (k >= deflated.columns()) {
119         std::stringstream fmt;
120         fmt << "Cantidad de autovalores esperado es demasiado grande, " << k << " en una matriz de " <<
    deflated.columns();
121         throw new std::out_of_range(fmt.str());
122     }
123
124     Timer timer("Decompose Timer");
125     std::list<EigenPair> output;
126     // Vector inicial para esta iteracion
127     std::vector<double> x0((unsigned long) deflated.columns(), 0.0);
128
129     for (int i = 0; i < k; ++i) {
130         for (int l = 0; l < deflated.columns(); ++l) {
131             x0[l] = random() % 1337 + 1;
132         }
133
134         // Obtenemos el i-esimo eigenpair dominante
135         EigenPair dominant = powerIteration(deflated, x0, norm, iterations);
136
137         if (dominant.first != dominant.first) {
138             std::cerr << "Error sacando el autovalor " << i << ". Vector: " << std::endl;
139
140             for (int i = 0; i < deflated.columns(); ++i) {
141                 std::cerr << x0[i] << " ";
142             }
143
144             std::cerr << std::endl;
145             --i;
146         } else {
147             // Hacemos deflacion, para el proximo paso
148             deflation(deflated, dominant);
149
150             // Lo guardamos al final de la lista
151             output.push_back(dominant);
152         }
```

```
153      }
154
155      return output;
156  }
157
158  /**
159  * Realiza un cambio de base a las filas de src, guardandolas en dst y utlizando los autovectores indicados.
160  *
161  * @param src matriz con imagenes
162  * @param dst matriz destino
163  * @param l lista de EigenPair
164  */
165
166  void dimensionReduction(const Matrix& src, Matrix& dst, const std::list<EigenPair>& l) {
167      int c = 0;
168
169      for (const EigenPair &ep : l) {
170          const std::vector<double>& eigenVector = ep.second;
171
172          for (int i = 0; i < src.rows(); i++) {
173              double sum = 0.0;
174
175              for (int j = 0; j < src.columns(); j++) {
176                  sum += eigenVector[j] * src(i,j);
177              }
178
179              // los guardamos en fila, asi reutilizamos otros metodos.
180              dst(i,c) = sum;
181          }
182
183          c++;
184      }
185  }
186
187  std::vector<double> operator*(const double &m, const std::vector<double> &n) {
188      std::vector<double> output(n);
189      std::transform(output.begin(), output.end(), output.begin(), [m](const double &x) -> double { return x *
     m; });
190      return output;
191  }
```