```hpp
/*
 * La classe IndexInvalide représente l'exception qui représente une
 * erreur d'indice dans l'accès à un élément d'une matrice
 */

#pragma once

#include <exception>

class IndexInvalide : public std::exception {
protected:
  std::string msg;
public:
  IndexInvalide(const int i) {
    this->msg = "Indice invalide : " + std::to_string(i);
  }
  IndexInvalide(const int i1, const int i2, const int j1, const int j2) {
    this->msg = "Indices invalides : " + std::to_string(i1) + "<=" + std::to_string(i2)
 +
      std::to_string(j1) + "<=" + std::to_string(j2);
  }
  IndexInvalide(const IndexInvalide &e) : msg(e.msg) {}
  // redéfinition de what
  virtual const char* what() const noexcept override {
    return msg.c_str();
  }
};
```



```hpp
/*
 * La classe MatriceAbstraiteDouble est une classe abstraite pour
 * représenter des matrices M x N de réels double
 *
 */
#pragma once

#include <iostream>
#include <sstream>
#include <cassert>
#include <iomanip>

#include "matriceAbstraite.hpp"

class matriceAbstraiteDouble : public matriceAbstraite<double> {

protected:
  static constexpr double DEFAULT_RATIO = 10.0; // 10 % ratio matrice creuse pas
 défaut
  double ratio =  DEFAULT_RATIO;

public:
  /*
   * antécédent : m>0 et n>0
   * rôle : initialise le nombre de lignes, de  colonnes
   *        et d'éléments de la matrice courante
   */
  matriceAbstraiteDouble(const int m, const int n) : matriceAbstraite<double>(m,
n) {}

  /*
   * rôle : constructeur par défaut
   */
  matriceAbstraiteDouble() {}

  /*
   * rôle : renvoie le ratio courante pour une matrice creuse
   */
  double getRatio() const { return this->ratio; }

  /*
   * rôle : moidifie le ratio courante pour une matrice creuse
   */
  void setRatio(const double r) { this->ratio = r; }

  /*
   * rôle : renvoie le nombre de double différents de 0.0
   *        dans la matrice courante
   */
  virtual int nbNonZeros() const =0;

  /*
   * rôle : renvoie la matrice somme de la matrice courante et de la matrice m
   */
  virtual matriceAbstraiteDouble &plus(const matriceAbstraiteDouble &m) const =0
;

  /*
   * rôle : teste si la matrice courante est creuse ou pas
   */
  bool estCreuse() const {
    return (100.0*nbNonZeros()/matriceAbstraite::nbElem())<=this->getRatio();
  }

  /*
   * rôle : surcharge de l'opérateur + pour renvoyer ma matrice la somme de
```

```
    *        de la matrice  courante et de la matrice m
    */
  inline matriceAbstraiteDouble &operator+(const matriceAbstraiteDouble &m) cons
t {
    return this->plus(m);
  }
};
```



```
/*
 * La classe MatriceAbstraite est une classe abstraite pour
 * représenter des matrices M x N génériques
 *
 */

#pragma once

#include <cassert>
#include <sstream>
#include <iostream>
#include <iomanip>

template<typename T>
class matriceAbstraite {

protected:
  int nbElt;     // le nombre d'éléments de la matrice courante
  int nbL, nbC;  // son nombre de lignes et de colonnes

public:
  /*
   * antécédent : m>0 et n>0
   * rôle : initialise le nombre de lignes, de  colonnes
   *        et d'éléments de la matrice courante
   */
  matriceAbstraite(const int m, const int n) :nbL(m), nbC(n), nbElt(m*n) {
    assert(m>0 && n>0);
  }

  /*
   * rôle : constructeur par défaut
   */
  matriceAbstraite() {}

  /*
   * rôle : constructeur de copie de la matrice abstraite courante
   */
  matriceAbstraite(const matriceAbstraite<T> &m) {
    this->nbL = m.nbL;
    this->nbC = m.nbC;
    this->nbElt = m.nbElt;
  }

  /*
   * Rôle : destructeur virtuel de la matrice courante
   */
  virtual ~matriceAbstraite() {};

  /*
   * rôle : renvoie le nombre de lignes de la matrice courante
   */
  inline int nbLig() const { return this->nbL; }

  /*
   * rôle : renvoie le nombre de colonnes de la matrice courante
   */
  inline int nbCol() const { return this->nbC; }

  /*
   * rôle : renvoie le nombre d'élément de la matrice courante
   */
  inline int nbElem() const { return this->nbElt; }

  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
```

```
   *               j>=0 and j<matriceAbstraite::nbLig()
   *  rôle : renvoie l'élément d'indice (i,j) de la matrice courante
   */
  virtual T get(const int i, const int j) const =0;

  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
   *              j>=0 and j<matriceAbstraite::nbLig()
   *  rôle : met x à l'indice (i,j) dans la matrice courante
   */
  virtual void set(const int i, const int j, const T& x) =0;

  /*
   * rôle : renvoie la sous-matrice ([i1;i2] , [j1;j2]) de la matrice courante
   */
  virtual matriceAbstraite<T> *subMat(const int i1, const int i2,  const int j1,
 const int j2) const=0;

  /*
   * rôle : renvoie la représentation sous forme de chaîne de
   *        caractères de la matrice courante
   */
  std::string toString() const {
    std::ostringstream s;
    for (int i=0; i<this->nbL; i++) {
      for (int j=0; j<this->nbC; j++)
        s << this->get(i,j) <<  " ";
      //
      s << "\n";
    }
    return s.str();
  }

  /*
   * rôle : écrit sur dans le fichier de texte f la matrice courante
   */
  friend std::ostream&operator<<(std::ostream &f, const matriceAbstraite<T>& m)
  {
    return f << m.toString();
  }

  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
   *              j>=0 and j<matriceAbstraite::nbLig()
   * rôle       : renvoie l'élément d'indice (i,j) de la matrice courante
   */
  inline T operator() (int i, int j) const {
    return this->get(i,j);
  }
};
```



```
#include "matriceDouble.hpp"
#include "matriceCreuse.hpp"

/*
 * rôle : copie la matriceCreuse m vers la matriceCruse courante
 */
void matriceCreuse::dupliquer(const matriceCreuse &m) {
  matriceAbstraite::nbL = m.nbLig();
  matriceAbstraite::nbC = m.nbCol();
  this->lesElements = m.lesElements;
}

/*
 * rôle : initialise la matriceCreuse courante avec la matriceDouble m
 */
matriceCreuse::matriceCreuse(const matriceDouble &m) : matriceAbstraiteDouble(m.
nbLig(),m.nbCol())
{
  for (int i=0; i<m.nbLig(); i++)
    for (int j=0; j<m.nbCol(); j++) {
      double x =  m.get(i,j);
      if (x!=0.0) this->set(i, j, x);
    }
}

/*
 * antécédent : i>=0 and i<matriceAbstraite::nbLig()
 *              j>=0 and j<matriceAbstraite::nbLig()
 * rôle : renvoie le double d'indice (i,j) dans la matrice courante
 */
double matriceCreuse::get(const int i, const int j) const {
  assert(i>=0 and i<matriceAbstraite::nbL);
  assert(j>=0 and j<matriceAbstraite::nbC);
  // rechercher dans la map un élément d'indice (i,j)
  auto r=this->lesElements.find(std::make_pair(i,j));
  if (r != this->lesElements.end())
    // ok trouvé
    return r->second;
  // else non trouvé
  return 0.0;
}

/*
 * antécédent : i>=0 and i<matriceAbstraite::nbLig()
 *              j>=0 and j<matriceAbstraite::nbLig()
 * rôle : met le double x à l'indice (i,j) dans la matrice courante
 */
void matriceCreuse::set(const int i, const int j, const double &x) {
  assert(i>=0 and i<matriceAbstraite::nbL);
  assert(j>=0 and j<matriceAbstraite::nbC);
  //
  std::pair<int, int> index = std::make_pair(i,j);
  if (x!=0.0)
    // mettre x dans la map (éventuellement écrase la valeur précédente)
    this->lesElements[index] = x;
  else
    // x==0 => supprimer l'élément d'indice index s'il exsite
    this->lesElements.erase(index);
}

/*
 * rôle : renvoie la sous-matrice creuse ([i1;i2] , [j1;j2]) de la matrice coura
nte
 */
matriceAbstraiteDouble *matriceCreuse::subMat(const int i1, const int i2, const
int j1, const int j2)
```

```
  const
{
  if (i1<0 or i1>=matriceAbstraite::nbLig()) throw new IndexInvalide(i1);
  if (i2<0 or i2>=matriceAbstraite::nbLig()) throw new IndexInvalide(i2);
  if (j1<0 or j1>=matriceAbstraite::nbCol()) throw new IndexInvalide(j1);
  if (j2<0 or j2>=matriceAbstraite::nbCol()) throw new IndexInvalide(j2);
  if (i1>i2 or j1>j2) throw new IndexInvalide(i1, i2, j1, j2);
  int nbl = i2-i1+1;
  int nbc = j2-j1+1;
  matriceAbstraiteDouble *m = new matriceCreuse(nbl, nbc);
  // copier les éléments de this dans m
  int o_i = i1;
  for (int i=0; i<nbl; i++, o_i++)  {
    int o_j = j1;
    for (int j=0; j<nbc; j++, o_j++)
      m->set(i,j, (*this)(o_i, o_j));
  }
  return m;
}

/*
 * rôle : renvoie la matrice somme de la matrice courante et de la matrice m
 */
matriceAbstraiteDouble &matriceCreuse::plus(const matriceAbstraiteDouble &m) con
st {
  matriceAbstraiteDouble *res = new matriceCreuse(matriceAbstraite::nbL, matrice
Abstraite::nbC) ;
  for (int i=0; i<matriceAbstraite::nbL; i++)
    for (int j=0; j<matriceAbstraite::nbC; j++)
      res->set(i,j, m.get(i,j)+this->get(i,j));
  //
  return *res;
}

/*
 * rôle : affecte la matriceCreuse m à la matrice courante et renvoie la l'adres
se de *this
 */
matriceCreuse &matriceCreuse::operator=(const matriceCreuse &m) {
  this->dupliquer(m);
  return *this;
}

/*
 * rôle : renvoie le nombre de double différents de 0.0
 *        dans la matrice courante
 */
int matriceCreuse::nbNonZeros() const {
  return this->lesElements.size();
}
```



```
/*
 * La classe matriceCreuse représente des matrices M x N de réels double
 * dont seules les valeurs différentes de 0 sont mémorisées
 *
 */
#pragma once

#include <iostream>
#include <utility>
#include <map>

#include "matriceAbstraiteDouble.hpp"
#include "IndexInvalide.hpp"

class matriceDouble; // référence en avant

class matriceCreuse : public matriceAbstraiteDouble {

protected:

  std::map<std::pair<int,int>, double> lesElements; // les éléments différents d
e 0

  /*
   * rôle : copie la matriceCreuse m vers la matriceCreuse courante
   */
  void dupliquer(const matriceCreuse &m);

public:
  /*
   * antécédent : m>0 et n>0
   * rôle : initialise la matrice courante m x n
   */
  matriceCreuse(const int m, const int n) : matriceAbstraiteDouble(m,n) {}

  /*
   * rôle : initialise la matriceCreuse courante avec la matriceCreuse m
   */
  matriceCreuse(const matriceCreuse &m) {
      this->dupliquer(m);
  }

  /*
   * rôle : initialise la matriceCreuse courante avec la matriceDouble m
   */
  matriceCreuse(const matriceDouble &m);

  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
   *              j>=0 and j<matriceAbstraite::nbLig()
   * rôle : renvoie le double d'indice (i,j) dans la matrice courante
   */
  virtual double get(const int i, const int j) const override;

  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
   *              j>=0 and j<matriceAbstraite::nbLig()
   * rôle : met le double x à l'indice (i,j) dans la matrice courante
   */
  virtual void set(const int i, const int j, const double &x) override;

  /*
   * rôle : renvoie la sous-matrice creuse ([i1;i2] , [j1;j2]) de la matrice cou
rante
   */
  virtual matriceAbstraiteDouble *subMat(const int i1, const int i2, const int j
```

```
1, const int j2)
    const override;

  /*
   * rôle : renvoie la matrice somme de la matrice courante et de la matrice m
   */
  virtual matriceAbstraiteDouble &plus(const matriceAbstraiteDouble &m) const ov
erride;

  /*
   * rôle : affecte la matriceCreuse m à la matrice courante et renvoie la l'adr
esse de *this
   */
  matriceCreuse &operator=(const matriceCreuse &m);

  /*
   * rôle : renvoie le nombre de double différents de 0.0
   *        dans la matrice courante
   */
  virtual int nbNonZeros() const override;
};
```



```
#include <cassert>

#include "matriceDouble.hpp"
#include "matriceCreuse.hpp"
#include "IndexInvalide.hpp"

/*
 * rôle : copie la matriceDouble m vers la matriceDouble courante
 */
void matriceDouble::dupliquer(const matriceDouble &m) {
  this->mat = new double[matriceAbstraite::nbElt = m.nbElem()];
  matriceAbstraite::nbL = m.nbLig();
  matriceAbstraite::nbC = m.nbCol();
  for (int i=0; i<m.nbElem(); i++)
    this->mat[i] = m.mat[i];
}

/*
 * rôle : initialise la matriceDouble courante avec la matriceCreuse m
 */
matriceDouble::matriceDouble(const matriceCreuse &m) :
  matriceAbstraiteDouble(m.nbLig(),m.nbCol()),
  mat(new double[m.nbLig()*m.nbCol()])
{
  for (int i=0; i<m.nbLig(); i++)
    for (int j=0; j<m.nbCol(); j++)
      this->set(i, j, m.get(i,j));
}

/*
 * antécédent : i>=0 and i<matriceAbstraite::nbLig()
 *              j>=0 and j<matriceAbstraite::nbLig()
 * rôle : renvoie le double d'indice (i,j) dans la matrice courante
 */
double matriceDouble::get(const int i, const int j) const {
  assert(i>=0 and i<matriceAbstraite::nbLig());
  assert(j>=0 and j<matriceAbstraite::nbCol());
  return this->mat[matriceAbstraite::nbCol()*i+j];
}

/*
 * antécédent : i>=0 and i<matriceAbstraite::nbLig()
 *              j>=0 and j<matriceAbstraite::nbLig()
 * rôle : met le double x à l'indice (i,j) dans la matrice courante
 */
void matriceDouble::set(const int i, const int j, const double &x) {
  assert(i>=0 and i<matriceAbstraite::nbLig());
  assert(j>=0 and j<matriceAbstraite::nbCol());
  this->mat[matriceAbstraite::nbCol()*i+j] = x;
}

/*
 * rôle : renvoie la sous-matrice ([i1;i2] , [j1;j2]) de la matrice courante
 */
matriceAbstraiteDouble *matriceDouble::subMat(const int i1, const int i2, const
int j1, const int j2) const
{
  if (i1<0 or i1>=matriceAbstraite::nbL) throw new IndexInvalide(i1);
  if (i2<0 or i2>=matriceAbstraite::nbL) throw new IndexInvalide(i2);
  if (j1<0 or j1>=matriceAbstraite::nbC) throw new IndexInvalide(j1);
  if (j2<0 or j2>=matriceAbstraite::nbC) throw new IndexInvalide(j2);
  if (i1>i2 or j1>j2) throw new IndexInvalide(i1, i2, j1, j2);
  int nbl = i2-i1+1;
  int nbc = j2-j1+1;
  matriceAbstraiteDouble *m = new matriceDouble(nbl, nbc);
  // copier les éléments de this dans m
```

```
  int o_i = i1;
  for (int i=0; i<nbl; i++, o_i++)  {
    int o_j = j1;
    for (int j=0; j<nbc; j++, o_j++)
      m->set(i,j, (*this)(o_i, o_j));
  }
  return m;
}

/*x2
 * rôle : renvoie la matrice somme de la matrice courante et de la matrice m
 */
matriceAbstraiteDouble &matriceDouble::plus(const matriceAbstraiteDouble &m) con
st {
  assert(m.nbLig() == matriceAbstraite::nbLig());
  assert(m.nbCol() == matriceAbstraite::nbCol());
  matriceAbstraiteDouble *res = new matriceDouble(matriceAbstraite::nbLig(), mat
riceAbstraite::nbCol()) ;
  for (int i=0; i<matriceAbstraite::nbLig(); i++)
    for (int j=0; j<matriceAbstraite::nbCol(); j++)
      res->set(i,j, m.get(i,j)+this->get(i,j));
  //
  return *res;
}

/*
 * rôle : affecte la matriceDouble m à la matrice courante et renvoie la l'adres
se de *this
 */
int matriceDouble::nbNonZeros() const {
  int nbElts;
  for (int i=0; i<matriceAbstraite::nbElem(); i++)
    if (this->mat[i]!=0.0) nbElts++;
  return nbElts;
}

/*
 * rôle : affecte la matriceDouble m à la matrice courante et renvoie la l'adres
se de *this
 */
const matriceDouble &matriceDouble::operator=(const matriceDouble &m) {
  delete [] this->mat;
  this->dupliquer(m);
  return *this;
}
```



```
/*
 * La classe matriceDouble représente des matrices M x N de réels double
 *
 */
#pragma once

#include <iostream>
#include <sstream>
#include <cassert>
#include <iomanip>

#include "matriceAbstraiteDouble.hpp"
#include "matriceCreuse.hpp"
#include "IndexInvalide.hpp"

class matriceDouble : public matriceAbstraiteDouble {

protected:
  double *mat; // tableau pour contenir les éléments de la matrice courante
  /*
   * rôle : copie la matriceCreuse m vers la matriceCruse courante
   */
  void dupliquer(const matriceDouble &m);

public:
  /*
   * antécédent : m>0 et n>0
   * rôle : initialise la matrice courante m x n à la valeur v
   */
  matriceDouble(const int m, const int n, const double v=0.0) :
    matriceAbstraiteDouble(m,n), mat(new double[m*n])  {
    for (int i=0; i<matriceAbstraite::nbElem(); i++)
      this->mat[i] = v;
  }

  /*
   * rôle : initialise la matriceDouble courante avec la matriceDouble m
   */
  matriceDouble(const matriceDouble &m) {
    this->dupliquer(m);
  }

  /*
   * rôle : initialise la matriceDouble courante avec la matriceCreuse m
   */
  matriceDouble(const matriceCreuse &m);

  /*
   * rôle : détruit la matriceDouble courante
   */
  ~matriceDouble() override { delete [] mat; }

  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
   *              j>=0 and j<matriceAbstraite::nbLig()
   * rôle : renvoie le double d'indice (i,j) dans la matrice courante
   */

  virtual double get(const int i, const int j) const override;
  /*
   * antécédent : i>=0 and i<matriceAbstraite::nbLig()
   *              j>=0 and j<matriceAbstraite::nbLig()
   * rôle : met le double x à l'indice (i,j) dans la matrice courante
   */
  virtual void set(const int i, const int j, const double &x) override;
```

```
  /*
   * rôle : renvoie la sous-matrice ([i1;i2] , [j1;j2]) de la matrice courante
   */
  virtual matriceAbstraiteDouble *subMat(const int i1, const int i2,
                                         const int j1, const int j2) const override;

  /*
   * rôle : renvoie la matrice somme de la matrice courante et de la matrice m
   */
  virtual matriceAbstraiteDouble &plus(const matriceAbstraiteDouble &m) const ov
erride;

  /*
   * rôle : affecte la matriceDouble m à la matrice courante et renvoie la l'adr
esse de *this
   */
  const matriceDouble &operator=(const matriceDouble &m);

  /*
   * rôle : affecte la matriceDouble m à la matrice courante et renvoie la l'adr
esse de *this
   */
  virtual int nbNonZeros() const override;
};
```