Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 23-06-2020

Last Name: ....
First Name: ....
Student ID: ....
Signature: ....

Exam duration: 2 hours (Online version)

Students can use a pen or a pencil for answering questions.

Students are NOT permitted to use books, course notes, calculators, mobile phones, and similar connected devices.

Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.

Writing on the cheat sheet is NOT allowed.

Exercise 1: _____ Exercise 2: _____ Exercise 3: _____

Exercise 1 (15 points)

You have to implement a library which provides facilities to use Matlab like vectors and matrices in C++. In particular, you have to implement column and row vectors and 2D matrices of double type. Elements indexing will follow the C++ convention, i.e., the first element in a vector has index 0, but your vectors and matrices can grow as in Matlab. So, for example if the current content of a matrix is the following:

$$m = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

after the assignment :

`m(2,4)= 7`

the content of the matrix becomes:

$$m = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 3.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 7.0 \end{pmatrix}$$

In particular, you have to provide the implementation (i.e., both declarations and definition) of the class `MatlabLike2DMatrix`:

1. defining the data structures you need
2. implement a **constructor** that, given the number of rows and columns, will create the initial data structure which will be filled with 0.0s
3. implement the `operator ()` to read and write the individual elements and to introduce additional rows and columns as needed. Provide both the const e non const version
4. implement the `operator *` which returns a copy of the initial matrix obtained as the product with a given scalar
5. implement an overloaded version of the `operator *` which computes the traditional rows by columns multiplication of two matrices

Additionally, you have to

6. provide the declarations of the classes `MatlabLikeColVector` and `MatlabLikeRowVector`
7. implement the **constructor** for one of the two

8. implement the **operator ()** to access the individual element of one of the two. Provide both the const e non const version

9. finally, if `rv` and `cv` are a row vector and a column vector variables, respectively, each including two doubles, given your implementation will be:

   MatlabLike2DMatrix m = rv * cv;

   a proper assignment?

Remarks

Duplicated code should be avoided: if a functionality needed by a function is already implemented as a method, it should be reused.
Provide the implementation of other required methods or functions, if any.
You don't need to cope with error conditions (e.g., assume that the matrices involved in a multiplication have the proper size).

Solution 1

The header file of the class MatlabLike2DMatrix is as follows:

```
#ifndef MATLABLIKELIB_MATLABLIKE2DMATRIX_H
#define MATLABLIKELIB_MATLABLIKE2DMATRIX_H

#include <vector>
#include <iostream>


using std::vector;
using std::cout;
using std::endl;

class MatlabLike2DMatrix {

protected:
   vector<vector<double>> data;
   void resize(size_t new_rows, size_t new_cols);

public:
   MatlabLike2DMatrix(size_t rows, size_t cols);


   double & operator () (size_t i, size_t j);
   double operator () (size_t i, size_t j) const;

   MatlabLike2DMatrix operator * (double scalar) const;
   void print(void) const;

   size_t n_cols(void) const {
      return data[0].size();
   }
   size_t n_rows(void) const {
      return data.size();
   }

};

MatlabLike2DMatrix operator * (const MatlabLike2DMatrix & m1, const MatlabLike2DMatrix & m2);
```

```
#endif //MATLABLIKELIB_MATLABLIKE2DMATRIX_H
```

The proposed data structure is a simple vector of vectors. The **operator ()** is overloaded in order to return a constant copy of an element and a reference, respectively. The **operator *** for the scalar is implemented as a member function while its version for multiplying two matrices is implemented as an helper function. `resize()` is a core method that is used both by the constructor and by the **operator ()** non const version . Its goal is to implement the growing functionality. The `n_cols()` and `n_rows()` methods , finally, return the matrix number of columns and rows and are usefull to implement the matrices multiplication.

The class implementation is reported below:

```
#include "MatlabLike2DMatrix.h"

double & MatlabLike2DMatrix::operator()(size_t i, size_t j) {
    // if we are indexing an additional row, lets resize data matrix
    if (i>= data.size())
        resize(i+1,data[0].size());
    // if we are indexing an additional col,
    // let's resize data matrix, without changing its row number
    if (j>=data[i].size())
        resize(data.size(), j+1);
    return data[i][j];
}

double  MatlabLike2DMatrix::operator()(size_t i, size_t j) const {
    return data.at(i).at(j);
}


MatlabLike2DMatrix MatlabLike2DMatrix::operator*(double scalar) const {
    MatlabLike2DMatrix result(data.size(), data[0].size());
    for (size_t i=0; i<data.size(); ++i)
        for (size_t j =0; j <data[i].size(); ++j)
            result.data[i][j] = data[i][j] * scalar;
    return result;
}

MatlabLike2DMatrix::MatlabLike2DMatrix(size_t rows, size_t cols){
    resize(rows,cols);
}

void MatlabLike2DMatrix::print() const {
    for (size_t i= 0; i< data.size(); ++i) {
        for (size_t j = 0; j < data[i].size(); ++j)
            cout << data[i][j] << " ";
        cout << endl;
    }
}
void MatlabLike2DMatrix::resize(size_t new_rows, size_t new_cols){
    // if additional rows are requested enlarge data matrix
    if (new_rows >= data.size())
        data.resize(new_rows);

    // Then resize all rows (this has no effect if new_cols is
    // as before
    for (size_t i = 0; i <data.size(); ++i)
        data[i].resize(new_cols);

}
```

```
MatlabLike2DMatrix operator*(const MatlabLike2DMatrix &  m1,  const MatlabLike2DMatrix & m2) {
   MatlabLike2DMatrix result(m1.n_rows(), m2.n_cols());

   for (size_t i =0; i < m1.n_rows(); ++i)
      for (size_t j=0; j< m2.n_cols(); ++j)
         for (size_t k=0; k< m1.n_cols(); ++k)
            result(i,j) += m1(i,k)*m2(k,j);

   return result;
}
```

There are few remarkable points in the proposed implentation:

- the resize() method first allocates rows and then allocates individual elements for each row.
- the non const version of operator() first checks if an additional row is indexed and in that case the matrix is resized accordingly. If the user is indexing an additional column, then the data matrix is resized taking care to not change the rows number.

MatlabLikeColVector and MatlabLikeRowVector are specialization of MatlabLike2DMatrix. For example MatlabLikeColVector declaration and definition are:

```
#ifndef MATLABLIKELIB_MATLABLIKECOLVECTOR_H
#define MATLABLIKELIB_MATLABLIKECOLVECTOR_H

#include "MatlabLike2DMatrix.h"

class MatlabLikeColVector : public MatlabLike2DMatrix {

   void resize(size_t n_elems);

public:
   MatlabLikeColVector(size_t n_elems);

   double & operator () (size_t i);

};



#endif //MATLABLIKELIB_MATLABLIKECOLVECTOR_H
```

and

```
#include "MatlabLikeColVector.h"

void MatlabLikeColVector::resize(size_t n_elems) {
   MatlabLike2DMatrix::resize(n_elems,1);
}

double &MatlabLikeColVector::operator()(size_t i) {
   return MatlabLike2DMatrix::operator()(i,0);
}

MatlabLikeColVector::MatlabLikeColVector(size_t n_elems) : MatlabLike2DMatrix(n_elems,1) {}
```

MatlabLikeColVector can be implemented easily by relying on its super-class methods by fixing the row index to 0 or specifying that there is a single row available. MatlabLikeRowVector can be defined similarly. Finally, the assignment:

```
MatlabLike2DMatrix m = rv * cv;
```

is correct. Thanks to inheritance, both `rv` and `cv` are also `MatlabLike2DMatrix` objects and then the **`operator *`** returns a 2x2 matrix.

Exercise 2 (9 points)

You have to develop **a parallel function** which computes the cosine similarity of two non-zero n-dimensional vectors (assume n significantly larger than the number of available processes). Cosine similarity is popular in machine learning (e.g., in recommender systems) to compute the distance of two *entities* and it is defineed as follows:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

where A_i and B_i are the components of vectors $\mathbf{A}$ and $\mathbf{B}$, respectively.

You can rely on the following class `nd_vector` which stores a vector in $\mathbb{R}^n$; an object of this class can be initialized both by copy and by providing the size, n. Furthermore, it provides a **size** method to read n and an unchecked indexing operator to access the values.

```
namespace numeric
{
  class nd_vector
  {
    typedef std::vector<double> container_type;

    container_type x;

  public:
    typedef container_type::value_type value_type;
    typedef container_type::size_type size_type;
    typedef container_type::pointer pointer;
    typedef container_type::const_pointer const_pointer;
    typedef container_type::reference reference;
    typedef container_type::const_reference const_reference;

    explicit nd_vector (size_type n = 0);
    nd_vector (std::initializer_list<double>);


    size_type
    size (void) const;

    void
    read (std::ifstream & input_stream);

    void
    print() const;


    reference
    operator [] (size_type);
    value_type
    operator [] (size_type) const;

    pointer
    data (void);
    const_pointer
```

```
    data (void) const;
  };
}

#endif
```

The prototype of the function is the following:

```
double cosine_similarity (const nd_vector & a, const nd_vector & b);
```

You can assume that the size of vectors is a multiple of the number of available cores.

Solution 2

The implementation of the function cosine_similarity is reported below:

```
double
cosine_similarity (const numeric::nd_vector & v1, const numeric::nd_vector & v2){
  double local_dot_term = 0;
  double local_v1_norm_term = 0;
  double local_v2_norm_term = 0;

  int rank;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  int size;
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  // commpute partial terms
  for (numeric::nd_vector::size_type i = rank; i < v1.size(); i+= size ){
    local_dot_term += v1[i]*v2[i];
    local_v1_norm_term += v1[i]* v1[i];
    local_v2_norm_term += v2[i]* v2[i];
  }

  double global_dot_term;
  double global_v1_norm_term;
  double global_v2_norm_term;
  // aggregate partial terms

  MPI_Allreduce (&local_dot_term, &global_dot_term, 1, MPI_DOUBLE,
            MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce (&local_v1_norm_term, &global_v1_norm_term, 1, MPI_DOUBLE,
            MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce (&local_v2_norm_term, &global_v2_norm_term, 1, MPI_DOUBLE,
            MPI_SUM, MPI_COMM_WORLD);
  global_v1_norm_term = sqrt(global_v1_norm_term);
  global_v2_norm_term = sqrt(global_v2_norm_term);

  return global_dot_term/(global_v1_norm_term*global_v2_norm_term);



}
```

Since the two input vectors are already available to the callee, the implementation relies on the cyclic block partitioning scheme. Three MPI_Allreduce() opeations are needed to provide the partial results to all processess to compute the final return value.

Exercise 3 (8 points)

The class Account keeps track of the total amount (amount instance variable) of money (in dollars) stored in a bank account and it provides methods to deposit and withdraw a certain amount (n) of dollars. Note that trying to withdraw more money than the amount stored in the account results in withdrawing no money and the amount is left unchanged.

Two subclasses of Account are also given: RiskyAccount and PremiumAccount.

In RiskyAccount when withdrawing more money than the ones stored, a penalty of 2% of the total amount is subtracted from amount itself and stored in instance variable blocked. When depositing an amount n greater than blocked not only n is added to amount but also the blocked money are merged back into amount and blocked is set to 0.

On the other hand, PremiumAccount account gives a bonus equal to the 1% of n for each deposit.

Given the provided source code, you have to:

1. **List and motivate** the program **output**
2. **List and motivate** the **expected output** of the program if **keyword virtual** is added in the declaration of methods **deposit** and **withdraw** in class **Account**, and **keyword override** is added in the same methods of the subclasses.

Provided source code:

- **account.h**

```
class Account {
protected:
    float amount;
public:
    Account(float initial = 0): amount(initial) {};
    void deposit(float);
    float withdraw(float);
    float getAmount();
};

class RiskyAccount : public Account {
private:
    float blocked;
public:
    RiskyAccount(float initial = 0): Account(initial), blocked(0) {};
    void deposit(float);
    float withdraw(float);
};

class PremiumAccount : public Account {
public:
    PremiumAccount(float initial = 0): Account(initial) {};
    void deposit(float);
};
```

- **account.cpp**

```
void Account::deposit(float n) {
    cout << "Acc-Dep" << endl;
    if (n > 0)
        amount += n;
}

float Account::withdraw(float n) {
    cout << "Acc-Wit" << endl;
    if (n <= 0 || n > amount)
        return 0;
```

```
    amount -= n;
    return n;
}

float Account::getAmount(){
    return amount;
}

float RiskyAccount::withdraw(float n)  {
    cout << "Ris-Wit" << endl;
    if (n <= 0) return 0;
    if (n > amount){
        float penalty = amount * 0.02;
        amount -= penalty;
        blocked += penalty;
        return 0;
    }
    amount -= n;
    return n;
}

void RiskyAccount::deposit(float n) {
    cout << "Ris-Dep" << endl;
    Account::deposit(n);
    if (n >= blocked){
        amount += blocked;
        blocked = 0;
    }
}

void PremiumAccount::deposit(float n) {
    cout << "Pre-Dep" << endl;
    if (n > 0)
        amount += n*1.01;
}
```

- **main.cpp**

```
int main() {
    Account *a = new Account();
    a->deposit(1000);
    a->withdraw(500);
    a->withdraw(700);

    cout << "---"<< endl;

    Account *b = new RiskyAccount();
    b->deposit(1000);
    b->withdraw(1300);
    b->deposit(10);

    cout << "---"<< endl;

    PremiumAccount *c = new PremiumAccount(1000);
    c->deposit(5000);
    c->withdraw(1000);

    cout << "---"<< endl;
```

```
    RiskyAccount d(100);
    d.withdraw(150);
    d.deposit(10);

    cout << "---"<< endl;

    cout << "a: " << a->getAmount() << "$ b: " << b->getAmount() << "$ c: "
        << c-> getAmount() << "$ d: " << d.getAmount() << "$" << endl;

    delete a; delete b; delete c;

    return 0;
}
```

Solution 3

1. Being not declared as virtual, methods `deposit` and `withdraw` are called by using the static type of the used account. Therefore, `RiskyAccount` $*b$ is used as a normal account since its static type is `Account`. In the case of `PremiumAccount` $*c$, method `withdraw` is not overridden and therefore the `Account` implementation is called. As a result no penalty is blocked on account $*b$ (since it behaves as a normal account) while a 50$ bonus is added to account $*c$ when depositng 5000$. Finally, variable d always uses methods of class `RiskyAccount` having static and dynamic types equals to `RiskyAccount`.

   ```
   Acc-Dep
   Acc-Wit
   Acc-Wit
   ---
   Acc-Dep
   Acc-Wit
   Acc-Dep
   ---
   Pre-Dep
   Acc-Wit
   ---
   Ris-Wit
   Ris-Dep
   Acc-Dep
   ---
   a: 500$ b: 1010$ c: 5050$ d: 110$
   ```

2. When methods are declared as virtual and override is added in the subclasses (however, not that this is not striclty required) the dynamic type is considered. While a, c and d behave as before, account $*b$ uses `RiskyAccount::deposit/withdraw` and a penalty of 20$ is blocked when withdrawing 1300$ having an `amount` of 1000$. The blocked money are then not restored since a deposit of 10$ is not sufficient. Note also that `RiskyAccount::deposit` calls `Account::deposit` as shown in the output log below.

   ```
   Acc-Dep
   Acc-Wit
   Acc-Wit
   ---
   Ris-Dep
   Acc-Dep
   Ris-Wit
   Ris-Dep
   Acc-Dep
   ---
   Pre-Dep
   Acc-Wit
   ```

Ris-Wit
Ris-Dep
Acc-Dep
---
a: 500$ b: 990$ c: 5050$ d: 110$