به نام خدا

گزارش کار تمرین اول AP

محمدمهدی مالوردی

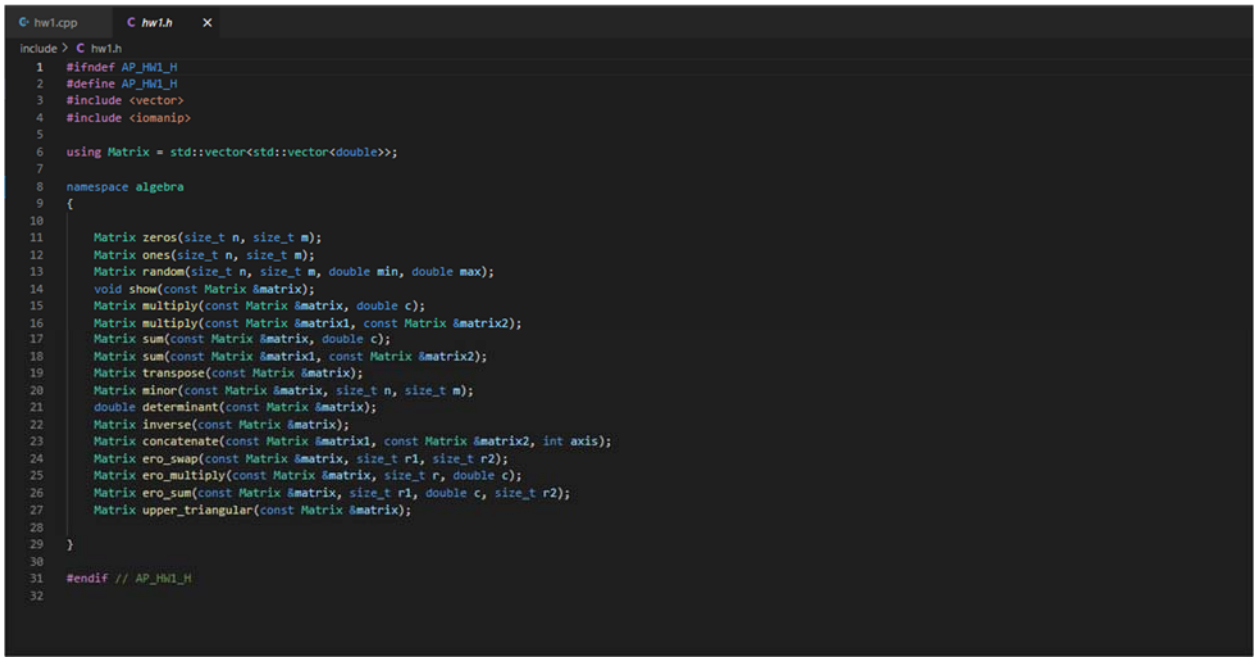
9723079

استاد: امیر جهانشاهی

گروه تدریس یاری: کیان بهزاد و امیر بیات

شرح گزارش

ابتدا داخل فایل هدر ، تمامی توابع را داخل algebra namespace تعریف می کنیم. بعضی از فراخوانی کتابخانه ها را نیز در این فایل انجام می دهیم.



hw1.h

حال تعریف توابع را تک به تک در زیر از فایل hw1.cpp استخراج می کنیم:

توابع zeros و ones

این توابع را به وسیله متد push_back، حلقه های تو در تو و ماتریس کمکی op به صورت زیر پیاده سازی می کنیم:

hw1.cpp | hw1.h

src > hw1.cpp

```cpp
#include <iostream>
#include <random>
#include "hw1.h"
#include <iomanip>
#include <cmath>

Matrix algebra::zeros(size_t n, size_t m)
{

    Matrix output{};

    std::vector<double> op{};

    for (size_t i{}; i < n; i++)
    {

        for (size_t j{}; j < m; j++)
        {

            op.push_back(0);
        }

        output.push_back(op);
        op.clear();
    }

    return output;
}

Matrix algebra::ones(size_t n, size_t m)
{

    Matrix output{};

    std::vector<double> op{};

    for (size_t i{}; i < n; i++)
    {

        for (size_t j{}; j < m; j++)
        {

            op.push_back(1);
        }

        output.push_back(op);
        op.clear();
    }

    return output;
}
```

ones, zeros

توابع random و show

در این توابع نیز مانند توابع قبلی عمل کردیم. برای تولید عدد رندوم از روش گفته شده در صورت سوال(کتابخانه رندوم c++11) استفاده کردیم. برای show هم از setw و setprecision استفاده کردیم.

```
std::random_device rd;
std::mt19937 mt(rd());
std::uniform_real_distribution<double> dist(min, max);
```

برای تولید عدد رندوم

```
53  Matrix algebra::random(size_t n, size_t m, double min, double max)
54  {
55
56      Matrix output{};
57      std::vector<double> op{};
58
59      if (min > max)
60      {
61          throw std::logic_error("Caution: min cannot be greater than max");
62      }
63
64      std::random_device rd;
65      std::mt19937 mt(rd());
66      std::uniform_real_distribution<double> dist(min, max);
67
68      for (size_t i{}; i < n; i++)
69      {
70          for (size_t j{}; j < m; j++)
71          {
72
73              op.push_back(dist(mt));
74          }
75
76          output.push_back(op);
77          op.clear();
78      }
79
80      return output;
81  }
82
83  void algebra::show(const Matrix &matrix)
84  {
85
86      for (size_t i{}; i < matrix.size(); i++)
87      {
88          for (size_t j{}; j < matrix[0].size(); j++)
89          {
90
91              std::cout << std::setw(3) << std::setprecision(3) << matrix[i][j] << " ";
92          }
93
94          std::cout << std::endl;
95      }
96  }
97
```

random, show

توابع multiply

```cpp
Matrix algebra::multiply(const Matrix &matrix, double c)
{

    std::vector<double> op{};
    Matrix output{};
    if (matrix.size() == 0)
    {
        return output;
    }

    for (size_t i{}; i < matrix.size(); i++)
    {
        for (size_t j{}; j < matrix[0].size(); j++)
        {

            op.push_back(c * matrix[i][j]);
        }

        output.push_back(op);
        op.clear();
    }

    return output;
}
```

ضرب عدد ثابت در تک تک درایه های ماتریس

در این جا خیلی راحت و به وسیله دو حلقه تو در تو، عدد ثابت c را در تک تک درایه های ماتریس ضرب می کنیم. باید دقت کنیم که اگر سایز ماتریس ما صفر بود، همان ماتریس تهی رو باید برگردانیم.

```cpp
Matrix algebra::multiply(const Matrix &matrix1, const Matrix &matrix2)
{
    std::vector<double> op{};
    Matrix output{};

    if (matrix1.size() == 0 && matrix2.size() == 0)
    {
        return output;
    }
    size_t n{matrix1.size()};
    size_t m{matrix1[0].size()};
    size_t q{matrix2.size()};
    size_t p{matrix2[0].size()};

    double sum{};

    if (n == 0 || m == 0 || q == 0 || p == 0)
    {

        throw std::logic_error("Caution: multiplication of 2 empty matrix");
    }

    if (m != q)
    {

        throw std::logic_error("Caution: matrices with wrong dimensions cannot be multiplied");
    }
    else
    {

        for (size_t i{}; i < n; i++)
        {
            for (size_t j{}; j < p; j++)
            {
                for (size_t k{}; k < m; k++)
                {
                    sum += (matrix1[i][k] * matrix2[k][j]);
                }
                op.push_back(sum);
                sum = 0;
            }

            output.push_back(op);
            op.clear();
        }
    }

    return output;
}
```

ضرب دو ماتریس n*m

در اینجا نیز علاوه بر بررسی سایز ماتریس که در ضرب عدد ثابت هم انجام دادیم، به وسیله سه حلقه تو در تو که هر کدام از داخلی ترین به بیرونی ترین به ترتیب تعداد جمع های مورد نیاز، تعداد ستون ها و تعداد سطر ها را کنترل می کند. باید حواسمان باشد که در هر دور op و sum را clear و صفر کنیم تا هربار دچار مشکل نشویم.

توابع sum

```cpp
Matrix algebra::sum(const Matrix &matrix, double c)
{

    std::vector<double> op{};
    Matrix output{};

    if (matrix.size() == 0)
    {

        return output;
    }

    for (size_t i{}; i < matrix.size(); i++)
    {
        for (size_t j{}; j < matrix[0].size(); j++)
        {

            op.push_back(c + matrix[i][j]);
        }

        output.push_back(op);
        op.clear();
    }

    return output;
}

```

جمع تک به تک درایه ها با عددی ثابت

مشابه ضرب می باشد، منتهی با فرق این که درایه ها را با عدد ثابت **c** جمع می کنیم.

```cpp
Matrix algebra::sum(const Matrix &matrix1, const Matrix &matrix2)
{

    std::vector<double> op{};
    Matrix output{};
    if (matrix1.empty() && matrix2.empty())
    {
        return output;
    }
    size_t height_of_matrix1{matrix1.size()};

    size_t height_of_matrix2{matrix2.size()};

    if (height_of_matrix1 != height_of_matrix2)
    {

        throw std::logic_error("Caution: matrices with wrong dimensions cannot be summed");
    }

    size_t length_of_matrix1{matrix1[0].size()};
    size_t length_of_matrix2{matrix2[0].size()};
    size_t n{height_of_matrix1}, m{length_of_matrix1};

    if (length_of_matrix1 != length_of_matrix2)
    {
        throw std::logic_error("Caution: matrices with wrong dimensions cannot be summed");
    }

    for (size_t i{}; i < n; i++)
    {
        for (size_t j{}; j < m; j++)
        {

            op.push_back(matrix1[i][j] + matrix2[i][j]);
        }

        output.push_back(op);
        op.clear();
    }

    return output;
}

```

جمع دو ماتریس با یکدیگر

به وسیله دو حلقه تو در تو تک به تک درایه ها را در هر موقعیت با یکدیگر جمع کرده و ماتریس خروجی را تشکی می دهیم.

توابع transpose و minor

```cpp

Matrix algebra::transpose(const Matrix &matrix)
{
    std::vector<double> op{};
    Matrix output{};

    if (matrix.empty())
    {

        return output;
    }

    for (size_t i{}; i < matrix[0].size(); i++)
    {
        for (size_t j{}; j < matrix.size(); j++)
        {

            op.push_back(matrix[j][i]);
        }

        output.push_back(op);
        op.clear();
    }

    return output;
}

Matrix algebra::minor(const Matrix &matrix, size_t n, size_t m)
{

    std::vector<double> op{};
    Matrix output{};

    for (size_t i{}; i < matrix.size(); i++)
    {

        if (i != n)
        {

            for (size_t j{}; j < matrix[0].size(); j++)
            {

                if (j != m)
                {
                    op.push_back(matrix[i][j]);
                }
            }

            output.push_back(op);
            op.clear();
        }
    }

    return output;
}
```

ماتریس transpose و minor

در ماتریس ترانهاده، به کمک دو حلقه تو در تو جای سطر و ستون را عوض می کنیم.

در تابع محاسبه ماینور ها نیز به وسیله ماتریس های کمکی op و output ، درایه هایی که اندیس های سطر و ستون آن ها با اندیس های اعلان شده متفاوت است را در قالب ماتریس خروجی در می آوریم.

تابع determinant

```
297
298  double algebra::determinant(const Matrix &matrix)
299  {
300      if (matrix.empty())
301      {
302          return 1.0;
303      }
304
305      double sum{};
306
307      if (matrix.size() != matrix[0].size())
308      {
309
310          throw std::logic_error("Caution: non-square matrices have no determinant");
311      }
312
313      if (matrix.size() == 2 && matrix[0].size() == 2)
314      {
315
316          sum = matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
317      }
318      else if (matrix.size() == 1 && matrix[0].size() == 1)
319      {
320          sum = matrix[0][0];
321      }
322      else
323      {
324          for (size_t j{}; j < matrix[0].size(); j++)
325          {
326
327              sum += matrix[0][j] * pow(-1, 0 + j) * determinant(minor(matrix, 0, j));
328          }
329      }
330      return sum;
331  }
332
```

تابع محاسبه دترمینان

ابتدا باید دقت کنیم که فقط ماتریس های مربعی دترمینان دارند. همچین دتمرینان ماتریس های 2*2 و 1*1 را جدا گانه محاسبه می نماییم. دترمینان ماتریس empty را نیز برابر یک در نظر می گیریم. حال دترمینان سایر ابعاد را به کمک تابع minor، تابعی بازگشتی از determinant و روش یاد گرفته شده در جبر خطی محاسبه می نماییم.

تابع inverse

```cpp
Matrix algebra::inverse(const Matrix &matrix)
{

    std::vector<double> op{};
    Matrix adj{}, adjT{}, inv{};
    double det{determinant(matrix)};
    if (matrix.empty())
    {
        return adj;
    }

    if (matrix.size() != matrix[0].size())
    {

        throw std::logic_error("Caution: non-square matrices have no inverse");
    }

    if (det == 0)
    {

        throw std::logic_error("Caution: singular matrices have no inverse");
    }

    for (size_t i{}; i < matrix.size(); i++)
    {
        for (size_t j{}; j < matrix[0].size(); j++)
        {

            op.push_back(pow(-1, i + j) * determinant(minor(matrix, i, j)));
        }

        adj.push_back(op);
        op.clear();
    }

    adjT = transpose(adj);
    inv = multiply(adjT, 1 / det);

    return inv;
}
```

تابع محاسبه معکوس ماتریس n*m

به کمک ماینور ها و دترمینان، معکوس ماتریس را محاسبه می کنیم. باید دقت کنیم که ماتریس های غیر مربعی و سینگولار(دترمینان برابر با صفر)، معکوس ندارند.

تابع concatenate

```cpp
Matrix algebra::concatenate(const Matrix &matrix1, const Matrix &matrix2, int axis = 0)
{

    std::vector<double> op{};
    Matrix output{};

    if (axis == 0)
    {

        if (matrix1[0].size() != matrix2[0].size())
        {

            throw std::logic_error("Caution: matrices with wrong dimensions cannot be concatenated");
        }
        for (size_t i{}; i < matrix1.size() + matrix2.size(); i++)
        {
            for (size_t j{}; j < matrix1[0].size(); j++)
            {

                if (i < matrix1.size())
                {
                    op.push_back(matrix1[i][j]);
                }
                else if (i >= matrix1.size() && i < matrix1.size() + matrix2.size())
                {
                    op.push_back(matrix2[i - matrix1.size()][j]);
                }
            }

            output.push_back(op);
            op.clear();
        }
    }

    else if (axis == 1)
    {

        if (matrix1.size() != matrix2.size())
        {

            throw std::logic_error("Caution: matrices with wrong dimensions cannot be concatenated");
        }

        for (size_t i{}; i < matrix1.size(); i++)
        {
            for (size_t j{}; j < matrix1[0].size() + matrix2[0].size(); j++)
            {

                if (j < matrix1[0].size())
                {
                    op.push_back(matrix1[i][j]);
                }
                else if (j >= matrix1[0].size() && j < matrix1[0].size() + matrix2[0].size())
                {
                    op.push_back(matrix2[i][j - matrix1[0].size()]);
                }
            }

            output.push_back(op);
            op.clear();
        }
    }

    return output;
}
```

تابع بهم چسباندن ماتریس ها به صورت عمودی یا افقی

برای این کار نیز متناسب با سایز ماتریس ها و این که به صورت عمودی یا افقی قرار است آن ها را داخل یک ماتریس واحد قرار دهیم، خروجی را در output قرار دادیم.

توابع ero_swap, ero_multiply, ero_sum

```cpp
Matrix algebra::ero_swap(const Matrix &matrix, size_t r1, size_t r2)
{
    std::vector<double> op{};
    Matrix output{matrix};

    if (r1 >= output.size() || r2 >= output.size())
    {

        throw std::logic_error("Caution: r1 or r2 inputs are out of range");
    }

    for (size_t j{}; j < output[0].size(); j++)
    {

        op.push_back(output[r1][j]);
    }
    output[r1].clear();
    for (size_t j{}; j < output[0].size(); j++)
    {

        output[r1].push_back(output[r2][j]);
    }
    output[r2].clear();
    for (size_t j{}; j < output[0].size(); j++)
    {

        output[r2].push_back(op[j]);
    }

    return output;
}

```

تابع ero_swap

در این تابع، پس از این که مطمئن شدیم که r1 , r2 در بازه سطر های موجود هستند، به کمک ماتریس های کمکی op و output و یک حلقه for ، این دو سطر را با یکدیگر جابجا می کنیم.

```cpp
Matrix algebra::ero_multiply(const Matrix &matrix, size_t r, double c)
{
    std::vector<double> op{};
    Matrix output{};

    if (r >= matrix.size())
    {

        throw std::logic_error("Caution: r1 or r2 inputs are out of range");
    }

    for (size_t i{}; i < matrix.size(); i++)
    {
        if (i == r)
        {
            for (size_t j{}; j < matrix[0].size(); j++)
            {

                op.push_back(matrix[i][j] * c);
            }

            output.push_back(op);
            op.clear();
        }

        else
        {

            for (size_t j{}; j < matrix[0].size(); j++)
            {

                op.push_back(matrix[i][j]);
            }

            output.push_back(op);
            op.clear();
        }
    }

    return output;
}
```

تابع ero_multiply

در این تابع، پس از رسیدن به سطر مورد نظر r ، عدد ثابت c را در آن سطر ضرب کرده و جایگزین آن سطر می کنیم. بقیه سطر ها را همانگونه که هستند باقی می گذاریم. در نهایت ماتریس نتیجه را در output می ریزیم.

```cpp
Matrix algebra::ero_sum(const Matrix &matrix, size_t r1, double c, size_t r2)
{
    Matrix output{matrix};
    std::vector<double> op;

    if (r1 >= output.size() || r2 >= output.size())
    {

        throw std::logic_error("Caution: r1 or r2 inputs are out of range");
    }

    for (size_t j{0}; j < output[0].size(); j++)
    {

        op.push_back(c * output[r1][j]);
    }

    for (size_t j{0}; j < output[0].size(); j++)
    {

        output[r2][j] += op[j];
    }

    return output;
}
```

تابع ero_sum

در این تابع، ابتدا از ماتریس کمکی output استفاده کرده و مقدار اولیه آن را برابر با ماتریس ورودی قرار می دهیم. همچنین عدد ثابت c را در سطر r1 ام ماتریس کمکی ضرب می کنیم. در نهایت تغییرات و جابجایی مورد نظر را روی output اعمال کرده و آن را بر میگردانیم.

ماتریس upper_tiangular

```cpp
Matrix algebra::upper_triangular(const Matrix &matrix)
{
    Matrix output{matrix};
    if (output.empty())
    {
        return output;
    }
    if (output.size() != output[0].size())
    {

        throw std::logic_error("Caution: non-square matrices have no upper triangular form");
    }

    for (size_t i{}; i < output.size(); i++)
    {

        for (size_t j{i}; j < output.size() - 1; j++)
        {

            output = ero_sum(output, i, -1 * output[j + 1][i] / output[i][i], j + 1);
        }
    }

    return output;
}
```

تابع تبدیل ماتریس ورودی به ماتریس بالامثلثی

ابتدا باید توجه کنیم که فقط ماتریس های مربعی را می توان بالا مثلثی کرد. در این تابع نیز به کمک ماتریس کمکی output و تابع ero_sum عملیات بالا مثلثی کردن ماتریس ورودی را درون دو حلقه تو در تو و به وسیله فرمول به دست آمده روی کاغذ که روبری output داخل حلقه درونی نوشته شده است، انجام دادیم.

در نهایت 23 تست از 24 تست پاس شدند که تست 24 ام امتیازی بود.