```
    1: #include "means.h"
    2: #include <vector>
    3: #include <iostream>
    4: using namespace std;
    5:
    6: int main(int argc, char **argv)
    7: {
    8:     double arithmetic_mean, geometric_mean, harmonic_mean;
    9:
   10:     // Fixed version
   11:     calculate_means(1, 4, 16, arithmetic_mean, geometric_mean, harmonic_mean);
   12:     cout << arithmetic_mean << ", " << geometric_mean << ", " << harmonic_mean << en
dl;
   13:
   14:     calculate_means(2, 3, 5, arithmetic_mean, geometric_mean, harmonic_mean);
   15:     cout << arithmetic_mean << ", " << geometric_mean << ", " << harmonic_mean << en
dl;
   16:
   17:     calculate_means(1000, 4000, 16000, arithmetic_mean, geometric_mean, harmonic_mea
n);
   18:     cout << arithmetic_mean << ", " << geometric_mean << ", " << harmonic_mean << en
dl;
   19:     cout << "-------------------------------" << endl;
   20:
   21:
   22:
   23:     // Scalable version
   24:     calculate_means(vector<int> {1, 4, 16}, arithmetic_mean, geometric_mean, harmoni
c_mean);
   25:     cout << arithmetic_mean << ", " << geometric_mean << ", " << harmonic_mean << en
dl;
   26:
   27:     calculate_means(vector<int> {2, 3, 5}, arithmetic_mean, geometric_mean, harmonic
_mean);
   28:     cout << arithmetic_mean << ", " << geometric_mean << ", " << harmonic_mean << en
dl;
   29:
   30:     calculate_means(vector<int> {1000, 4000, 16000}, arithmetic_mean, geometric_mean
, harmonic_mean);
   31:     cout << arithmetic_mean << ", " << geometric_mean << ", " << harmonic_mean << en
dl;
   32:
   33:
   34:     return 0;
   35: }
```

```cpp
#include "../ex1A_mean_values/means.h"
#include <cmath>
#include <vector>

void calculate_means(int a, int b, int c, double &am, double &gm, double &hm)
{
    am = (a + b + c)/3.0;
    gm = exp((log(a)+log(b)+log(c))/3);
    hm = 3.0/(1.0/a + 1.0/b + 1.0/c);
}

void calculate_means(std::vector<int> numbers, double &am, double &gm, double &hm)
{
    int n = numbers.size();

    am = 0.;
    gm = 0.;
    hm = 0.;

    for (int i = 0; i < n; ++i)
    {
        am += numbers[i];
        gm += log(numbers[i]);
        hm += 1.0/numbers[i];
    }

    am /= n;
    gm = exp(gm/n);
    hm = n/hm;
}
```

```
    1: #pragma once
    2: #include <vector>
    3:
    4: /**
    5:   This function calculates arithmetic mean, geometric mean and harmonic mean of thre
e integers.
    6:   @param[in]    a           first integer
    7:   @param[in]    b           second integer
    8:   @param[in]    c           third integer
    9:   @param[out]   am          arithmetic mean
   10:   @param[out]   gm          geometric mean
   11:   @param[out]   hm          harmonic mean
   12: */
   13: void calculate_means(int a, int b, int c, double &am, double &gm, double &hm);
   14:
   15: /**
   16:   This function calculates arithmetic mean, geometric mean and harmonic mean of an i
nteger vector.
   17:   @param[in]    numbers     vector containing integers
   18:   @param[out]   am          arithmetic mean
   19:   @param[out]   gm          geometric mean
   20:   @param[out]   hm          harmonic mean
   21: */
   22: void calculate_means(std::vector<int> numbers, double &am, double &gm, double &hm);
   23:
```

```cpp
#include "../ex1A_mean_values/means.h"
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;


int main(int argc, char **argv)
{
    // read vector from file
    vector<int> data_vector = {};

    ifstream input_stream("data_1.txt");

    int line;
    while(input_stream >> line)
    {
        data_vector.push_back(line);
    }
    data_vector.shrink_to_fit();


    // calculate minimum and maximum
    vector<int>::iterator min_it = min_element(data_vector.begin(), data_vector.end(
));
    vector<int>::iterator max_it = max_element(data_vector.begin(), data_vector.end(
));

    // calculate arithmetic mean, geometric mean and harmonic mean
    double am, gm, hm;
    calculate_means(data_vector, am, gm, hm);


    // calculate standard deviation
    double sd = 0.;
    int n = data_vector.size();
    for (int i = 0; i < n; ++i)
    {
        sd += pow(data_vector[i] - am, 2);
    }
    sd = sqrt(sd/n);


    // print results
    cout << "minimum: " << *min_it << endl;
    cout << "maximum: " << *max_it << endl;
    cout << "arithmetic mean: " << am << endl;
    cout << "geometric mean: " << gm << endl;
    cout << "harmonic mean: " << hm << endl;
    cout << "standard deviation: " << sd << endl;

    return 0;
}
```

```cpp
#include "special_sum.h"
#include "../utils/timing.h"
#include <iostream>
#include <chrono>
#include <stack>
using namespace std;

int main(int argc, char **argv)
{
    // check results and compare speeds
    for(size_t n : {15, 1001, 1432987})
    {
        cout << "n = " << n << endl;
        size_t sum_1, sum_2;

        tic();
        for(size_t i = 0; i < 1000; ++i)
            sum_1 = special_sum_loop(n);
        double time_1 = toc();

        tic();
        for(size_t i = 0; i < 1000; ++i)
            sum_2 = special_sum_noloop(n);
        double time_2 = toc();

        cout << "loop: " << sum_1 << "\t\tDuration: " << time_1 << endl;
        cout << "no loop: " << sum_2 << "\t\tDuration: " << time_2 << endl << "-----
----------------------------------------" << endl;
    }

    return 0;
}
```

```cpp
#include "special_sum.h"

size_t gauss_sum(size_t n)
{
    return (n*(n+1))/2;
}

size_t special_sum_loop(size_t n)
{
    size_t sum = 0;
    for (size_t i = 1; i < n+1; ++i)
    {
        if (i % 3 == 0 || i % 5 == 0)
        {
            sum += i;
        }
    }
    return sum;
}

size_t special_sum_noloop(size_t n)
{
    size_t factor_3 = gauss_sum(n/3);   // dividing int by int automatically gets rounded off
    size_t factor_5 = gauss_sum(n/5);
    size_t factor_15 = gauss_sum(n/15);

    return factor_3*3 + factor_5*5 - factor_15*15;
}
```

```
    1: #include <cstddef>
    2:
    3: /**
    4:   This function returns the sum of all positive integers less or equal n which are a
 multiples of 3 or of 5, WITH using a loop.
    5:   @param[in]    n
    6:   @param[out]   M
    7: */
    8: size_t special_sum_loop(size_t n);
    9:
   10:
   11: /**
   12:   This function returns the sum of all positive integers less or equal n which are a
 multiples of 3 or of 5, WITHOUT using a loop.
   13:   Example: For n=15, we have 60 = 3+5+6+9+10+12+15 = (1+2+3+4+5)*3 + (1+2+3)*5 - 1*1
5
   14:   Formula: M = (\sum_{i=1}^{k_3} i)*3 + (\sum_{i=1}^{k_5} i)*5 - (\sum_{i=1}^{k_15}
i)*15
   15:   @param[in]    n
   16:   @param[out]   M
   17: */
   18: size_t special_sum_noloop(size_t n);
```

```cpp
#include "mylib.h"
#include <cmath>
#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
    for(size_t i = 1; i < 8; ++i)
    {
        size_t n = pow(10,i);
        vector<double> x(n);
        for (size_t k = 0; k < n; ++k)
            x[k] = 1.0/(k + 1);


        // compute scalar products
        double sum_1 = scalar(x, x);
        double sum_2 = Kahan_skalar(x, x);

        // compute error
        double err_1 = abs(sum_1 - pow(M_PI,2)/6);
        double err_2 = abs(sum_2 - pow(M_PI,2)/6);

        cout << "n = " << n << endl;
        cout << "Normal scalar product: " << sum_1 << "\terror: " << err_1 << endl;
        cout << "Kahan scalar product: " << sum_2 << "\terror: " << err_2 << endl;
        cout << endl;
    }



    return 0;
}
```

```cpp
#include "mylib.h"
#include <cassert>        // assert()
#include <cmath>
#include <vector>
using namespace std;

double scalar(vector<double> const &x, vector<double> const &y)
{
    assert(x.size() == y.size());
    size_t const N = x.size();
    double sum = 0.0;
    for (size_t i = 0; i < N; ++i)
    {
        sum += x[i] * y[i];
    }
    return sum;
}



double Kahan_skalar(vector<double> const &x, vector<double> const &y)
{
    double sum = 0;
    double c = 0;
    size_t n = x.size();
    for (size_t i = 0; i < n; ++i)
    {
        double z = x[i]*y[i] - c;     // c is the part that got lost in the last iteration
        double t = sum + z;           // when adding sum + z, the lower digits are lost if sum is large
        c = (t - sum) - z;            // now we recover the lower digits to add in the next iteration
        sum = t;
    }
    return sum;
}
```

```
 1: #ifndef FILE_MYLIB
 2: #define FILE_MYLIB
 3: #include <vector>
 4:
 5: /**     Inner product
 6:         @param[in] x    vector
 7:         @param[in] y    vector
 8:         @return             resulting Euclidian inner product <x,y>
 9: */
10: double scalar(std::vector<double> const &x, std::vector<double> const &y);
11:
12: /**     Inner product using BLAS routines
13:         @param[in] x    vector
14:         @param[in] y    vector
15:         @return             resulting Euclidian inner product <x,y>
16: */
17: double scalar_cblas(std::vector<double> const &x, std::vector<double> const &y);
18: float scalar_cblas(std::vector<float> const &x, std::vector<float> const &y);
19:
20:
21: /**     L_2 Norm of a vector
22:         @param[in] x    vector
23:         @return             resulting Euclidian norm <x,y>
24: */
25: double norm(std::vector<double> const &x);
26:
27: double Kahan_skalar(std::vector<double> const &x,  std::vector<double> const &y);
28:
29:
30: #endif
```

```cpp
#include "../utils/timing.h"
#include <iostream>
#include <random>
#include <chrono>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;


size_t random_integer(int lower_bound, int upper_bound)
{
    unsigned seed = chrono::system_clock::now().time_since_epoch().count();

    minstd_rand0 generator (seed);

    return lower_bound + generator() % (upper_bound - lower_bound + 1);
}

int main(int argc, char **argv)
{
    // start with generating a sorted vector/list
    size_t n = 10000;
    vector<int> x_vec = {};
    list<int> x_list = {};
    for(size_t k = 0; k < n; ++k)
    {
        x_vec.push_back(k + 1);
        x_list.push_back(k + 1);
    }


    // insert new random entries such that the container stays sorted
    tic();
    for(size_t i = 0; i < n; ++i)
    {
        size_t new_entry = random_integer(1,n);
        auto it = lower_bound(x_vec.begin(), x_vec.end(), new_entry);
        x_vec.insert(it, new_entry);
    }
    double time_1 = toc();

    tic();
    for(size_t i = 0; i < n; ++i)
    {
        size_t new_entry = random_integer(1,n);
        auto it = lower_bound(x_list.begin(), x_list.end(), new_entry);
        x_list.insert(it, new_entry);
    }
    double time_2 = toc();


    // check results
    cout << "New vector is sorted: " << std::boolalpha << is_sorted(x_vec.begin(), x_vec.end()) << "\tsize: " << x_vec.size() << "\tduration: " << time_1 << endl;
    cout << "New list is sorted: " << std::boolalpha << is_sorted(x_list.begin(), x_list.end()) << "\tsize: " << x_list.size() << "\tduration: " << time_2 << endl;


    return 0;
}
```

```cpp
#include "goldbach.h"

size_t single_goldbach(size_t k)
{
    const std::vector<size_t> relevant_primes = get_primes(k);
    size_t m = relevant_primes.size();

    size_t counter = 0;

    for(size_t i = 0; i < m; ++i)
    {
        for(size_t j = i; j < m; ++j)
        {
            if(relevant_primes[i] + relevant_primes[j] == k)
                counter += 1;
        }
    }

    return counter;
}


std::vector<size_t> count_goldbach(size_t n)
{
    const std::vector<size_t> relevant_primes = get_primes(n);
    size_t m = relevant_primes.size();

    std::vector<size_t> counter_vector(n + 1, 0);


    for(size_t i = 0; i < m; ++i)
    {
        for(size_t j = i; j < m; ++j)
        {
            size_t sum = relevant_primes[i] + relevant_primes[j];
            if(sum <= n)
                ++counter_vector[relevant_primes[i] + relevant_primes[j]];
        }
    }

    return counter_vector;
}
```

```
    1: #pragma once
    2: #include "mayer_primes.h"
    3: #include <iostream>
    4: #include <vector>
    5: #include <iterator>
    6: #include <cassert>
    7:
    8:
    9: /**
   10:   This function returns the number of possible decompositions of an integer into a s
um of two prime numbers.
   11:   @param[in]    k            first integer
   12:   @param[out]   count        number of decompositions
   13: */
   14: size_t single_goldbach(size_t k);
   15:
   16: /**
   17:   This function returns the number of possible decompositions into a sum of two prim
e numbers of all even integers in the interval [4,n].
   18:   @param[in]    n                   upper integer bound
   19:   @param[out]   count_vector        vector containing the number of decompositions f
or a natural number the corresponding index
   20: */
   21: std::vector<size_t> count_goldbach(size_t n);
```

```cpp
#include "../utils/timing.h"
#include "goldbach.h"
#include <iostream>
#include <algorithm>
using namespace std;


int main(int argc, char **argv)
{
    cout << "Check: 694 has "<< single_goldbach(694) << " decompositions." << endl << "----------------------------------------" << endl;


    for(size_t n : {10000, 100000, 400000, 1000000, 2000000})
    {
        tic();

        auto goldbach_vector = count_goldbach(n);

        auto max_it = max_element(goldbach_vector.begin(), goldbach_vector.end());
        size_t max_number = distance(goldbach_vector.begin(), max_it);

        double time = toc();

        cout << "The number " <<  max_number << " has " << *max_it << " decompositions. Duration: " << time << endl;
    }

    /*
    The number 9240 has 329 decompositions. Duration: 0.00572876
    The number 99330 has 2168 decompositions. Duration: 0.3342
    The number 390390 has 7094 decompositions. Duration: 4.23734
    The number 990990 has 15594 decompositions. Duration: 29.5817
    The number 1981980 has 27988 decompositions. Duration: 135.985
    */


    return 0;
}
```

```
    1: #pragma once
    2:
    3: #include <cstring> //memset
    4: #include <vector>
    5: //using namespace std;
    6:
    7: /** \brief Determines all prime numbers in interval [2, @p max].
    8:  *
    9:  *  The sieve of Eratosthenes is used.
   10:  *
   11:  *  The implementation originates from <a href="http://code.activestate.com/recipes/
576559-fast-prime-generator/">Florian Mayer</a>.
   12:  *
   13:  * \param[in]   max end of interval for the prime number search.
   14:  * \return  vector of prime numbers @f$2,3,5, ..., p<=max @f$.
   15:  *
   16:  * \copyright
   17:  * Copyright (c) 2008 Florian Mayer          (adapted by Gundolf Haase 2018)
   18:  *
   19:  * Permission is hereby granted, free of charge, to any person obtaining a copy
   20:  * of this software and associated documentation files (the "Software"), to deal
   21:  * in the Software without restriction, including without limitation the rights
   22:  * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
   23:  * copies of the Software, and to permit persons to whom the Software is
   24:  * furnished to do so, subject to the following conditions:
   25:  *
   26:  * The above copyright notice and this permission notice shall be included in
   27:  * all copies or substantial portions of the Software.
   28:  *
   29:  *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLI
ED,
   30:  * INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   31:  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
   32:  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
   33:  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
   34:  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOF
TWARE.
   35:  *
   36:  */
   37: template <class T>
   38: std::vector<T> get_primes(T max)
   39: {
   40:     std::vector<T> primes;
   41:     char *sieve;
   42:     sieve = new char[max / 8 + 1];
   43:     // Fill sieve with 1
   44:     memset(sieve, 0xFF, (max / 8 + 1) * sizeof(char));
   45:     for (T x = 2; x <= max; x++)
   46:     {
   47:         if (sieve[x / 8] & (0x01 << (x % 8))) {
   48:             primes.push_back(x);
   49:             // Is prime. Mark multiplicates.
   50:             for (T j = 2 * x; j <= max; j += x)
   51:             {
   52:                 sieve[j / 8] &= ~(0x01 << (j % 8));
   53:                         }
   54:         }
   55:         }
   56:     delete[] sieve;
   57:     return primes;
   58: }
   59:
   60: //----------------------------------------------------------------
   61: //int main()        // by  Florian Mayer
   62: //{g++ -O3  -std=c++14 -fopenmp main.cpp && ./a.out
   63: //    vector<unsigned long> primes;
   64: //    primes = get_primes(10000000);
   65: //    // return 0;
```

```
66: //    // Print out result.
67: //    vector<unsigned long>::iterator it;
68: //    for(it=primes.begin(); it < primes.end(); it++)
69: //        cout << *it << " ";
70: //
71: //    cout << endl;
72: //    return 0;
73: //}
74:
```

```cpp
#pragma once
#include "sigmoid.h"
#include <iostream>
#include <vector>
using namespace std;

class DenseMatrix
{
    private:
        vector<double> M;
        size_t n;
        size_t m;


    public:
        vector<double> Mult(const vector<double> &x) const
        {
            vector<double> y(n,0);
            for(size_t i = 0; i < n; ++i)   // iterate row
            {
                for(size_t j = 0; j < m; ++j)   // iterate column
                {
                    y[i] += M[i*m + j]*x[j];
                }
            }
            return y;
        }

        vector<double> MultT(const vector<double> &y) const
        {
            vector<double> x(m,0);
            for(size_t j = 0; j < m; ++j)   // iterate column
            {
                for(size_t i = 0; i < n; ++i)   // iterate row
                {
                    x[j] += M[i*m + j]*y[i];
                }
            }
            return x;
        }

        void Print() const
        {
            for(size_t i = 0; i < n; ++i)   // iterate row
            {
                for(size_t j = 0; j < m; ++j)   // iterate column
                {
                    cout << M[i*m + j] << "  ";
                }
                cout << endl;
            }
            cout << endl;
        }


        DenseMatrix(size_t n, size_t m)
        {
            this->n = n;
            this->m = m;
            M = vector<double>(n*m);
            size_t nm = max(n,m);

            for(size_t i = 0; i < n; ++i)   // iterate row
            {
                for(size_t j = 0; j < m; ++j)   // iterate column
                {
                    M[i*m + j] = sigmoid(x_entry(i,nm))*sigmoid(x_entry(j,nm));
                }
```

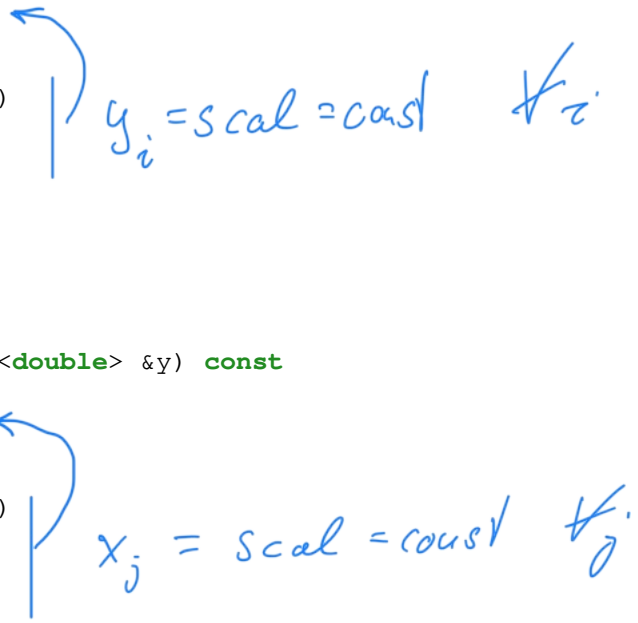
Access pattern

```
    69:             }
    70:         }
    71: };
```

```cpp
#include "../utils/timing.h"
#include "DenseMatrix.h"
#include "ProductMatrix.h"
#include <algorithm>

int main()
{
    // b) ----------------------------------------------------------------------------
    DenseMatrix const M(5,3);
    vector<double> const u{{1, 2, 3}};
    vector<double> f1 = M.Mult(u);
    vector<double> const v{{-1, 2, -3, 4, -5}};
    vector<double> f2 = M.MultT(v);

    M.Print();

    for(size_t i = 0; i < f1.size(); ++i)
        cout << f1[i] << endl;
    cout << endl;


    for(size_t j = 0; j < f2.size(); ++j)
        cout << f2[j] << "  ";
    cout << endl << "-------------------------------------------------" << endl;

    // c) ----------------------------------------------------------------------------
    size_t n = pow(10,3);
    DenseMatrix const M_1(n,n);
    vector<double> x(n, 1.0);

    size_t n_loops = 100;
    vector<double> y_1;
    vector<double> y_2;

    double time_1 = 0;
    double time_2 = 0;

    tic();
    for(int l = 0; l < n_loops; ++l)
        y_1 = M_1.Mult(x);
    time_1 += toc();

    tic();
    for(int l = 0; l < n_loops; ++l)
        y_2 = M_1.MultT(x);
    time_2 += toc();

    vector<double> error_vec(n,0);
    for(int i = 0; i < n; ++i)
        error_vec[i] = abs(y_1[i] - y_2[i]);
    double sup_error = *max_element(error_vec.begin(), error_vec.end());


    cout << "n = " << n << endl;
    cout << "Average duration for Mult: " << time_1/n_loops << endl;
    cout << "Average duration for MultT: " << time_2/n_loops << endl;
    cout << "sup-error: " << sup_error << endl;
    cout << "-------------------------------------------------" << endl;

    // d) ----------------------------------------------------------------------------
    vector<double> u_M(n,0);
    for(int i = 0; i < n; ++i)
        u_M[i] = sigmoid(x_entry(i, n));

    ProductMatrix const M_2(u_M, u_M);
```

```
66:
67:     time_1 = 0;
68:     time_2 = 0;
69:
70:     tic();
71:     for(int l = 0; l < n_loops; ++l)
72:         y_1 = M_2.Mult(x);
73:     time_1 += toc();
74:
75:     tic();
76:     for(int l = 0; l < n_loops; ++l)
77:         y_2 = M_2.MultT(x);
78:     time_2 += toc();
79:
80:     for(int i = 0; i < n; ++i)
81:         error_vec[i] = abs(y_1[i] - y_2[i]);
82:     sup_error = *max_element(error_vec.begin(), error_vec.end());
83:
84:
85:     cout << "n = " << n << endl;
86:     cout << "Average duration for Mult: " << time_1/n_loops << endl;
87:     cout << "Average duration for MultT: " << time_2/n_loops << endl;
88:     cout << "sup-error: " << sup_error << endl;
89:     cout << "-------------------------------------------------" << endl;
90:
91:
92:     return 0;
93: }
```

```cpp
#pragma once
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

class ProductMatrix
{
    private:
        vector<double> u;
        vector<double> v;
        size_t n;
        size_t m;

    public:
        vector<double> Mult(const vector<double> &x) const
        {
            vector<double> y(n,0);
            for(int i = 0; i < n; ++i)
            {
                for(int j = 0; j < m; ++j)
                {
                    y[i] += v[j]*x[j];
                }
                y[i] *= u[i];
            }
            return y;
        }

        vector<double> MultT(const vector<double> &y) const
        {
            vector<double> x(m,0);
            for(int j = 0; j < m; ++j)
            {
                for(int i = 0; i < n; ++i)
                {
                    x[j] += y[i]*u[i];
                }
                x[j] *= v[j];
            }
            return x;
        }

        ProductMatrix(const vector<double> &u, const vector<double> &v)
        {
            n = u.size();
            m = v.size();
            this->u = u;
            this->v = v;

        }
};
```



```
 1: #pragma once
 2: #include <cmath>
 3:
 4: double sigmoid(double x)
 5: {
 6:     return 1./(1. + exp(-x));
 7: }
 8:
 9: double x_entry(size_t k, size_t nm)
10: {
11:     return (10.*k)/(nm - 1) - 5.;
12: }
```

```
    1: //
    2: //    Gundolf Haase, Oct 18 2024
    3: //
    4: #pragma once
    5:
    6: #ifdef __NVCC__
    7: #include <cuda_runtime.h>
    8: #endif
    9: #ifdef _OPENMP
   10: #include <omp.h>
   11: #endif
   12: #include <iostream>
   13: #include <unordered_map>
   14:
   15: //#####################################
   16: // G.Haase
   17: // See https://sourceforge.net/p/predef/wiki/Compilers/
   18: //     http://www.cplusplus.com/doc/tutorial/preprocessor/
   19: //  also:  export OMP_DISPLAY_ENV=VERBOSE
   20: //#####################################
   21: /**     Checks for compilers, its versions, threads etc. on CPU
   22:  *
   23:         @param[in] argc number of command line arguments
   24:         @param[in] argv command line arguments as array of C-strings
   25: */
   26: template <class T>
   27: void check_env(T argc, char const *argv[])
   28: {
   29:     std::cout << "\n#########################################################################
#######\n";
   30:     std::cout << "Code    :";
   31:     for (T k = 0; k < argc; ++k) std::cout << "  " << argv[k];
   32:     std::cout << std::endl;
   33:
   34:     // compiler:      https://sourceforge.net/p/predef/wiki/Compilers/
   35:     std::cout <<    "Compiler:  ";
   36: #if defined __INTEL_COMPILER
   37: #pragma message(" ##########  INTEL  ###############")
   38:     std::cout << "INTEL " << __INTEL_COMPILER;
   39:     // Ignore warnings for #pragma acc   unrecognice
   40: #pragma warning disable 161
   41:     // Ignore warnings for #pragma omp   unrecognice
   42: #pragma warning disable 3180
   43:
   44: #elif defined __NVCC__
   45: //#pragma message(" ##########  NVCC    ###############")
   46: //  https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/
   47:     std::cout << "NVCC " << __CUDACC_VER_MAJOR__ << "." << __CUDACC_VER_MINOR__ ;
   48: #elif defined  __clang__
   49: #pragma message(" ##########  CLANG    ###############")
   50:     std::cout << "CLANG " << __clang_major__ << "." << __clang_minor__ << "."; // <<
 __clang_patchlevel__;
   51: #elif defined __GNUC__
   52: //#pragma message(" ##########  Gnu    ###############")
   53:     std::cout << "Gnu " <<  __GNUC__  << "." << __GNUC_MINOR__ << "." << __GNUC_PATC
HLEVEL__;
   54: #else
   55: #pragma message(" ##########  unknown Compiler   ###############")
   56:     std::cout << "unknown";
   57: #endif
   58:     std::cout << "  C++ standard: " << __cplusplus << std::endl;
   59:
   60:     // Parallel environments
   61:     std::cout <<    "Parallel:  ";
   62: #if defined MPI_VERSION
   63: #ifndef __GNUC__
   64: #pragma message(" ##########  MPI    ###############")
   65: #endif
```

```
   66: #ifdef OPEN_MPI
   67:     std::cout << "OpenMPI ";
   68: #else
   69:     std::cout << "MPI ";
   70: #endif
   71:     std::cout << MPI_VERSION << "." << MPI_SUBVERSION << "    ";
   72: #endif
   73:
   74: #ifdef _OPENMP
   75: //https://www.openmp.org/specifications/
   76: //https://stackoverflow.com/questions/1304363/how-to-check-the-version-of-openmp-on-
linux
   77:     std::unordered_map<unsigned, std::string> const map{
   78:         {200505, "2.5"}, {200805, "3.0"}, {201107, "3.1"}, {201307, "4.0"}, {201511,
 "4.5"}, {201611, "5.0"}, {201811, "5.0"}};
   79: #ifndef __GNUC__
   80: #pragma message(" ##########  OPENMP    ###############")
   81: #endif
   82:     std::cout << "OpenMP ";
   83:     try {
   84:         std::cout << map.at(_OPENMP);
   85:     }
   86:     catch (...) {
   87:         std::cout << _OPENMP;
   88:     }
   89:     #pragma omp parallel
   90:     {
   91:         #pragma omp master
   92:         {
   93:             const int nn = omp_get_num_threads();           // OpenMP
   94:             std::cout << " ---> " <<  nn << " Threads   ";
   95:         }
   96:         #pragma omp barrier
   97:     }
   98:
   99: #endif
  100: #ifdef _OPENACC
  101: #pragma message(" ##########  OPENACC    ###############")
  102:     std::cout << "OpenACC   ";
  103: #endif
  104:     std::cout << std::endl;
  105:     std::cout << "Date     :  " << __DATE__ << "  " << __TIME__;
  106:     std::cout << "\n###################################################################
#######\n";
  107: }
  108:
  109:
  110:
  111: /** @brief  Lists basic properties of GPU
  112:  */
  113: inline
  114: void printGPUInfo()
  115: {
  116:     using std::cout, std::endl, std::boolalpha;
  117: #ifdef __NVCC__
  118:     cout <<"\n++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n";
  119:     // https://devblogs.nvidia.com/how-query-device-properties-and-handle-errors-cud
a-cc/
  120:     int nDevices;
  121:     cudaError_t err = cudaGetDeviceCount(&nDevices);
  122:     if (err != cudaSuccess) printf("%i : %s\n", err, cudaGetErrorString(err));
  123:
  124:     //  https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html
  125:     cudaDeviceProp prop;
  126:     // https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/grou
p__CUDART__DEVICE_g5aa4f47938af8276f08074d09b7d520c.html
  127:     err = cudaGetDeviceProperties (&prop, 0);
  128:     if (err != cudaSuccess) printf("%i : %s\n", err, cudaGetErrorString(err));
```

```
  129:     cout << "We work on " << nDevices << " x " << prop.name << " GPU \n    with "
  130:          << prop.multiProcessorCount << " Multiprocessors (SME)"
  131:          << ", Compute capability " << prop.major << "." << prop.minor << endl;
  132:     cout << "global Mem: " << prop.totalGlobalMem/1024/1000/1000 << " GB" << endl;
  133:     cout << "shared Mem per SME: " << prop.sharedMemPerMultiprocessor/1024 << "kB"
  134:          << "   shared Mem per Block: " << prop.sharedMemPerBlock/1024 << " kB"
  135:          << "   32b-Registers per Block: " << prop.regsPerBlock << endl;
  136:     cout << "global L2 cache: " << prop.l2CacheSize/1024 << " kB"
  137:          << "   local L1 cache supported: " << boolalpha << bool(prop.localL1CacheSu
pported)  << endl;
  138:     cout << "max Threads per Block:" << prop.maxThreadsPerBlock << endl;
  139:     cout <<"++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n";
  140: #else
  141:     cout << endl << "No compiler support for GPU" << endl;
  142: #endif
  143: }
  144:
  145:
```

```
    1: #include "timing.h"              // tic(), toc()
    2: #include "info.h"
    3: #include "std_lib_facilities.h" // overload [] : by Stroustrup
    4: #include <iostream>
    5: using namespace std;
    6:
    7: int main(int argc , char const *argv[])
    8: {
    9:     cout << "Check my own headers\n" << endl;
   10:
   11:     tic();                      // start timer
   12:     check_env(argc, argv);          // info on compiler
   13:     printGPUInfo();                 // info on GPU. Only in NVCC_
   14:     double tt(toc());           // end timer
   15:     cout << "\n\nSeconds: " << tt << endl;
   16:
   17:     vector<double> aa(5,-1.2345);
   18:     try
   19:     {
   20:         cout << "\nThe next line should result in an exception\n";
   21:         cout << aa[5] << endl;// Stroustrups overloading should check the index rang
e
   22:     }
   23:     catch (Range_error &re)
   24:     {
   25:         cout << "Index out of range!!!\n";
   26:     }
   27:
   28:     return 0;
   29: }
   30:
```

```
    1: /*
    2: std_lib_facilities.h
    3: */
    4:
    5: /*
    6: simple "Programming: Principles and Practice using C++ (second edition)" course head
er to
    7: be used for the first few weeks.
    8: It provides the most common standard headers (in the global namespace)
    9: and minimal exception/error support.
   10:
   11: Students: please don't try to understand the details of headers just yet.
   12: All will be explained. This header is primarily used so that you don't have
   13: to understand every concept all at once.
   14:
   15: By Chapter 10, you don't need this file and after Chapter 21, you'll understand it
   16:
   17: Revised April 25, 2010: simple_error() added
   18:
   19: Revised November 25 2013: remove support for pre-C++11 compilers, use C++11: <chrono
>
   20: Revised November 28 2013: add a few container algorithms
   21: Revised June 8 2014: added #ifndef to workaround Microsoft C++11 weakness
   22: Revised Febrary 2 2015: randint() can now be seeded (see exercise 5.13).
   23: Revised August 3, 2020: a cleanup removing support for ancient compilers
   24: */
   25:
   26: #ifndef H112
   27: #define H112 080315L
   28:
   29:
   30: #include<iostream>
   31: #include<iomanip>
   32: #include<fstream>
   33: #include<sstream>
   34: #include<cmath>
   35: #include<cstdlib>
   36: #include<string>
   37: #include<list>
   38: #include <forward_list>
   39: #include<vector>
   40: #include<unordered_map>
   41: #include<algorithm>
   42: #include <array>
   43: #include <regex>
   44: #include<random>
   45: #include<stdexcept>
   46:
   47: //------------------------------------------------------------------------------
   48:
   49:
   50: typedef long Unicode;
   51:
   52: //------------------------------------------------------------------------------
   53:
   54: using namespace std;
   55:
   56: template<class T> string to_string(const T& t)
   57: {
   58:         ostringstream os;
   59:         os << t;
   60:         return os.str();
   61: }
   62:
   63: struct Range_error : out_of_range {     // enhanced vector range error reporting
   64:         int index;
   65:         Range_error(int i) :out_of_range("Range error: " + to_string(i)), index(i) {
 }
```

```
   66: };
   67:
   68:
   69: // trivially range-checked vector (no iterator checking):
   70: template< class T> struct Vector : public std::vector<T> {
   71:         using size_type = typename std::vector<T>::size_type;
   72:
   73: /* #ifdef _MSC_VER
   74:         // microsoft doesn't yet support C++11 inheriting constructors
   75:         Vector() { }
   76:         explicit Vector(size_type n) :std::vector<T>(n) {}
   77:         Vector(size_type n, const T& v) :std::vector<T>(n, v) {}
   78:         template <class I>
   79:         Vector(I first, I last) : std::vector<T>(first, last) {}
   80:         Vector(initializer_list<T> list) : std::vector<T>(list) {}
   81: */
   82:         using std::vector<T>::vector;   // inheriting constructor
   83:
   84:         T& operator[](unsigned int i) // rather than return at(i);
   85:         {
   86:                 if (/*i<0 || */ this->size() <= i) throw Range_error(i);
   87:                 return std::vector<T>::operator[](i);
   88:         }
   89:         const T& operator[](unsigned int i) const
   90:         {
   91:                 if (/*i<0 || */ this->size() <= i) throw Range_error(i);
   92:                 return std::vector<T>::operator[](i);
   93:         }
   94: };
   95:
   96: // disgusting macro hack to get a range checked vector:
   97: #define vector Vector
   98:
   99: // trivially range-checked string (no iterator checking):
  100: struct String : std::string {
  101:         using size_type = std::string::size_type;
  102:         //      using string::string;
  103:
  104:         char& operator[](unsigned int i) // rather than return at(i);
  105:         {
  106:                 if (/*i<0 || */ size() <= i) throw Range_error(i);
  107:                 return std::string::operator[](i);
  108:         }
  109:
  110:         const char& operator[](unsigned int i) const
  111:         {
  112:                 if (/*i<0 || */ size() <= i) throw Range_error(i);
  113:                 return std::string::operator[](i);
  114:         }
  115: };
  116:
  117:
  118: namespace std {
  119:
  120:         template<> struct hash<String>
  121:         {
  122:                 size_t operator()(const String& s) const
  123:                 {
  124:                         return hash<std::string>()(s);
  125:                 }
  126:         };
  127:
  128: } // of namespace std
  129:
  130:
  131: struct Exit : runtime_error {
  132:         Exit() : runtime_error("Exit") {}
  133: };
```

```
134:
135: // error() simply disguises throws:
136: inline void error(const string& s)
137: {
138:         throw runtime_error(s);
139: }
140:
141: inline void error(const string& s, const string& s2)
142: {
143:         error(s + s2);
144: }
145:
146: inline void error(const string& s, int i)
147: {
148:         ostringstream os;
149:         os << s << ": " << i;
150:         error(os.str());
151: }
152:
153:
154: template<class T> char* as_bytes(T& i)  // needed for binary I/O
155: {
156:         void* addr = &i;         // get the address of the first byte
157:         // of memory used to store the object
158:         return static_cast<char*>(addr); // treat that memory as bytes
159: }
160:
161:
162: inline void keep_window_open()
163: {
164:         cin.clear();
165:         cout << "Please enter a character to exit\n";
166:         char ch;
167:         cin >> ch;
168:         return;
169: }
170:
171: inline void keep_window_open(string s)
172: {
173:         if (s == "") return;
174:         cin.clear();
175:         cin.ignore(120, '\n');
176:         for (;;) {
177:                 cout << "Please enter " << s << " to exit\n";
178:                 string ss;
179:                 while (cin >> ss && ss != s)
180:                         cout << "Please enter " << s << " to exit\n";
181:                 return;
182:         }
183: }
184:
185:
186:
187: // error function to be used (only) until error() is introduced in Chapter 5:
188: inline void simple_error(string s)      // write ``error: s and exit program
189: {
190:         cerr << "error: " << s << '\n';
191:         keep_window_open();             // for some Windows environments
192:         exit(1);
193: }
194:
195: // make std::min() and std::max() accessible on systems with antisocial macros:
196: #undef min
197: #undef max
198:
199:
200: // run-time checked narrowing cast (type conversion). See ???.
201: template<class R, class A> R narrow_cast(const A& a)
```

```
  202: {
  203:         R r = R(a);
  204:         if (A(r) != a) error(string("info loss"));
  205:         return r;
  206: }
  207:
  208: // random number generators. See 24.7.
  209:
  210: inline default_random_engine& get_rand()
  211: {
  212:         static default_random_engine ran;       // note: not thread_local
  213:         return ran;
  214: };
  215:
  216: inline void seed_randint(int s) { get_rand().seed(s); }
  217:
  218: inline int randint(int min, int max) { return uniform_int_distribution<>{min, max}(g
et_rand()); }
  219:
  220: inline int randint(int max) { return randint(0, max); }
  221:
  222: //inline double sqrt(int x) { return sqrt(double(x)); } // to match C++0x
  223:
  224: // container algorithms. See 21.9.   // C++ has better versions of this:
  225:
  226: template<typename C>
  227: using Value_type = typename C::value_type;
  228:
  229: template<typename C>
  230: using Iterator = typename C::iterator;
  231:
  232: template<typename C>
  233: // requires Container<C>()
  234: void sort(C& c)
  235: {
  236:         std::sort(c.begin(), c.end());
  237: }
  238:
  239: template<typename C, typename Pred>
  240: // requires Container<C>() && Binary_Predicate<Value_type<C>>()
  241: void sort(C& c, Pred p)
  242: {
  243:         std::sort(c.begin(), c.end(), p);
  244: }
  245:
  246: template<typename C, typename Val>
  247: // requires Container<C>() && Equality_comparable<C,Val>()
  248: Iterator<C> find(C& c, Val v)
  249: {
  250:         return std::find(c.begin(), c.end(), v);
  251: }
  252:
  253: template<typename C, typename Pred>
  254: // requires Container<C>() && Predicate<Pred,Value_type<C>>()
  255: Iterator<C> find_if(C& c, Pred p)
  256: {
  257:         return std::find_if(c.begin(), c.end(), p);
  258: }
  259:
  260: #endif //H112
```

```
    1: //
    2: //    Gundolf Haase, Oct 18 2024
    3: //
    4: #pragma once
    5: #include <chrono>                   // timing
    6: #include <stack>
    7:
    8: //using Clock = std::chrono::system_clock;   //!< The wall clock timer chosen
    9: using Clock = std::chrono::high_resolution_clock;
   10: using TPoint= std::chrono::time_point<Clock>;
   11:
   12: // [Galowicz, C++17 STL Cookbook, p. 29]
   13: inline
   14: std::stack<TPoint> MyStopWatch; //!< starting time of stopwatch
   15:
   16: /** Starts stopwatch timer.
   17:  *  Use as @code tic(); myfunction(...) ; double tsec = toc();  @endcode
   18:  *
   19:  *  The timining can be nested and the recent time point is stored on top of the sta
ck.
   20:  *
   21:  *  @return recent time point
   22:  *  @see toc
   23:  */
   24: inline auto tic()
   25: {
   26:     MyStopWatch.push(Clock::now());
   27:     return MyStopWatch.top();
   28: }
   29:
   30: /** Returns the elapsed time from stopwatch.
   31:  *
   32:  * The time point from top of the stack is used
   33:  * if time point @p t_b is not passed as input parameter.
   34:  * Use as @code tic(); myfunction(...) ; double tsec = toc();  @endcode
   35:  * or as @code auto t_b = tic(); myfunction(...) ; double tsec = toc(t_b);  @endcode
   36:  * The last option is to be used in the case of
   37:  * non-nested but overlapping time measurements.
   38:  *
   39:  * @param[in]  t_b start time of some stop watch
   40:  * @return elapsed time in seconds.
   41:  *
   42: */
   43: inline double toc(TPoint const &t_b = MyStopWatch.top())
   44: {
   45:     // https://en.cppreference.com/w/cpp/chrono/treat_as_floating_point
   46:     using Unit       = std::chrono::seconds;
   47:     using FpSeconds = std::chrono::duration<double, Unit::period>;
   48:     auto t_e = Clock::now();
   49:     MyStopWatch.pop();
   50:     return FpSeconds(t_e-t_b).count();
   51: }
```