# 4.1

**Aufgabe 1 Vektor-Funktionen** (5 Punkte)

Felder vom Typ `vector<double>` mit beliebig vielen Werten lassen sich natürlich auch als Vektoren im Sinne der Mathematik betrachten.

Schreiben Sie folgende Vektor-Funktionen:

`double max(const vector<double> &v)`
Die Funktion gibt den größten im Vektor auftretenden Wert zurück.

`bool allPositive(const vector<double> &v)`
Die Funktion ermittelt, ob alle Werte des Vektors positiv sind.

`double product(const vector<double> &v1, const vector<double> &v2)`
Die Funktion gibt das Skalarprodukt zweier Vektoren zurück.

`vector<double> product(const vector<double> &v, double f)`
Die Funktion gibt das Produkt des Vektors mit der Zahl zurück.

`double norm(const vector<double> &v)`
Die Funktion gibt die Norm eines Vektors zurück ($||x||_2 = \sqrt{\sum_i x_i^2}$).

`void normalize(vector<double> &v)`
Die Funktion normalisiert einen übergebenen Vektor.

Testen Sie Ihre Funktionen in einem passenden Hauptprogramm. Lassen Sie dazu den Nutzer beliebig große Vektoren eingeben.

```
#include <iostream>
#include <vector>

using namespace std;

double max(const vector<double>& v) {}
bool allPositive(const vector<double>& v) {}
double product(const vector<double>& v1, const vector<double>& v2) {}
vector<double> product(const vector<double>& v, double f) {}
double norm(const vector<double>& v) {}
void normalize(vector<double>& v) {}

int main(){}
```

```cpp
double max(const vector<double>& v) {

    if (v.empty()) {
        throw invalid_argument("Der Vektor ist leer.");
    }

    double maxVal = -numeric_limits<double>::infinity();
    for (double val : v) {
        if (val > maxVal) {
            maxVal = val;
        }
    }
    return maxVal;
}
```

#include <limits>

-Infinity

```cpp
bool allPositive(const vector<double>& v) {
    for (double val : v) {
        if (val <= 0) {
            return false;
        }
    }
    return true;
```

```cpp
// Skalarprodukt
double product(const vector<double>& v1, const vector<double>& v2) {
    if (v1.size() != v2.size()) {
        throw invalid_argument("Die Vektoren haben unterschiedliche Größen.");
    }

    double result = 0.0;
    for (size_t i = 0; i < v1.size(); ++i) {
        result += v1[i] * v2[i];
    }
    return result;
}
```

v1.size = v2.size

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

- $a$ = 1st vector
- $b$ = 2nd vector
- $n$ = dimension of the vector space
- $a_i$ = component of vector a
- $b_i$ = component of vector b

Typically used for array indexing, loop counters, and when working with the sizes of containers such as std::vector, std::string, and other standard containers.
Use ++i when you don't need the old value of i, which is almost always the case in loop counters.
Use i++ if you specifically need the value of i before it's incremented.
Here they behave the same (if we ignore the extra memory)

```cpp
// Das Produkt eines Vektors mit einer Zahl berechnen
vector<double> product(const vector<double>& v, double f) {
    vector<double> result(v.size());
    for (size_t i = 0; i < v.size(); ++i) {
        result[i] = v[i] * f;
    }
    return result;
}
```

cin >> f

```cpp
// Die Norm berechnet
double norm(const vector<double>& v) {
    double sum = 0.0;
    for (double val : v) {
        sum += val * val;
    }
    return sqrt(sum);
}
```

$$(||x||_2 = \sqrt{\sum_i x_i^2})$$

```cpp
// Vektor normalisieren (in-place)
void normalize(vector<double>& v) {
    double vectorNorm = norm(v);
    if (vectorNorm == 0) {
        throw invalid_argument("Die Norm des Vektors ist 0. Der Vektor kann nicht normalisiert werden.");
    }

    for (double& val : v) {
        val /= vectorNorm;
    }
}
```

-> u/0

$$\hat{u} = \frac{\vec{u}}{||\vec{u}||}$$