Einführung in die Programmierung

Tutorium 11

Wintersemester 2025/2026

Arbeitsgruppe Systemsoftware
Angewandte Informatik 12

technische universität dortmund

Arbeitsgruppe Systemsoftware

Definition *n-dimensionaler Zeilenvektor*

Ein n-dimensionaler Zeilenvektor $\vec{v}$ ist ein Element der Menge M^n.

Dabei gelten die folgenden Voraussetzungen:

- M ist die Menge, die den Wertebereich darstellt.
- M^n bezeichnet das n-fache kartesische Produkt der Menge M mit sich selbst, also $M \times \cdots \times M$.

Die Menge M^n umfasst somit alle Zeilenvektoren, die sich aus diesem Wertebereich bilden lassen.

Aufgabe 1.1: Zeilenvektor

Die einzelnen Komponenten sollen mit einem Getter get(k) mit k = 0, 1, .. n − 1 abgerufen werden können.
Gilt k = n oder sogar k > n, liefert dieser 0 als Rückgabewert.
Vervollständigen Sie den Getter.

```
…
51      // Getter
52      T get(std::size_t n) const {
53          T result;
54          if (n >= N) {
55              result = 0;
56          } else {
57              //TODO 1):
58              result = components[n];
59              //TODO.
60          }
61          return result;
62      }
```

```
Vect<int, 3> v1{10, 20, 30};
```

$\overrightarrow{v1} = (10, 20, 30)$

```
int wert_index_1 = v1.get(1);
```

$wert\_index = 20$

```
int wert_index_1 = v1.get(5);
```

$wert\_index = 0$

Aufgabe 1.2: Zeilenvektor

Mit einem Setter set(k, value) mit k = 0, 1, ..n−1 kann die k-te Komponente des Vektors auf den Wert von value gesetzt werden.
Gilt k = n oder sogar k > n, soll nichts geschehen.
Vervollständigen Sie den Setter.

```
…
66      // Setter
67      void set(std::size_t n, T value) {
68          //TODO 2):
69          if (n < N) {
70              components[n] = value;
71          }
72          //TODO.
73      }
```

```
Vect<int, 3> v1{10, 20, 30};
```

$\overrightarrow{v1} = (10, 20, 30)$

```
v1.set(0, 99);
```

$\overrightarrow{v1} = (99, 20, 30)$

```
v1.set(5, 555);
```

$\overrightarrow{v1} = (99, 20, 30)$

Aufgabe 1.3: Zeilenvektor

Die Vektoren soll man lexikographisch ordnen können. Überladen Sie dazu geeignet den Operator <.
Alle anderen (in der Vorlage bereits vorbereiteten) Vergleichsoperatoren bauen auf diesen bzw. dem Vergleichsoperator für das Feld components auf und müssen nicht mehr implementiert werden.
Schauen Sie sich dazu die Präsenzaufgabe zu den Vergleichsoperatoren für das Fraction-Objekt noch einmal an.

```
…
75      // Comparison
76      bool operator<(const Vect& other) const {
77          //TODO 3):
78          return this->components < other.components;
79          //TODO.
80      }
```

```
Vect<int, 3> v1{1, 100, 100};
```

$\overrightarrow{v1} = (1, 100, 100)$

```
Vect<int, 3> v2{2, 0, 0};
```

$\overrightarrow{v2} = (2, 0, 0)$

```
v1 < v2
```

$true$

```
Vect<int, 3> v1{0, 0, 100};
```

$\overrightarrow{v1} = (0, 0, 100)$

```
Vect<int, 3> v2{0, 100, 0};
```

$\overrightarrow{v2} = (0, 100, 0)$

```
v1 < v2
```

$true$

```
Vect<int, 3> v1{0, 0, 100};
```

$\overrightarrow{v1} = (0, 0, 100)$

```
Vect<int, 3> v2{0, 0, 100};
```

$\overrightarrow{v2} = (0, 0, 100)$

```
v1 < v2
```

$false$

Aufgabe 1.4: Zeilenvektor

Vektoren sollen miteinander addiert und subtrahiert werden können.
Hierbei werden die einzelnen Komponenten paarweise addiert bzw. subtrahiert.
Vervollständigen Sie hierzu die Operatoren + und -.
Hat ein Vektor mehr Komponenten als der andere, so ist der Vektor mit weniger Komponenten durch Nullen ’aufzufüllen’.
Die Referenz auf die Summe bzw. Differenz wird als Rückgabewert zurückgegeben.

```
…
88      // Vector-Vector Arithmetic
89      template <typename U, std::size_t M> Vect<typename std::common_type<T, U>::type,
        MAX(M, N)> operator+(const Vect<U, M>& other) const {
90          constexpr std::size_t max_dim = MAX(M, N);
91          using ResultType = typename std::common_type<T, U>::type;
92          //TODO 4a):
93          for (std::size_t i = 0; i < max_dim; ++i) {
            result.set(i, static_cast<ResultType>(this->get(i)) +
            static_cast<ResultType>(other.get(i))); }
94          //TODO.
95          return result;
96      }
…
```

$$\vec{a} + \vec{b} = (a_1, a_2 \cdots a_n) + (b_1, b_2 \cdots b_n) = (a_1 + b_1, a_2 + b_2 \cdots a_n + b_n)$$

```
Vect<int, 2> v1{10, 20};
```

$\overrightarrow{v1} = (10, 20)$

```
Vect<float, 3> v2{1.5, 2.5, 3.5};
```

$\overrightarrow{v2} = (1.5, 2.5, 3.5)$

```
auto summe = v1+ v2;
```

$\overrightarrow{summe} = (11.5, 22.5, 3.5)$

```
…
99      Vect operator-(const Vect& other) const {
100         Vect result;
101         //TODO 4b):
102         for (std::size_t i = 0; i < N; ++i)
            { result.components[i] = components[i] - other.components[i]; }
103         //TODO.
104         return result;
105     }
```

$$\vec{a} - \vec{b} = (a_1, a_2 \cdots a_n) - (b_1, b_2 \cdots b_n) = (a_1 - b_1, a_2 - b_2 \cdots a_n - b_n)$$

Aufgabe 1.5: Zeilenvektor

Vektoren sollen um einen Wert skaliert werden können. Dazu wird jede Komponente des Vektors mit dem entsprechenden Skalar multipliziert.
Vervollständigen Sie hierzu den entsprechenden Operator *.
Eine Referenz auf den skalierten Vektor wird als Rückgabewert zurückgegeben.

```
…
107     // Vector-Scalar Arithmetic
108     template <typename Scalar> Vect<typename std::common_type<T, Scalar>::type, N>
        operator*(Scalar scalar) const {
109         using ResultType = typename std::common_type<T, Scalar>::type;
110         Vect<ResultType, N> result;
111         //TODO 5):
112         for (std::size_t i = 0; i < N; ++i) { result.set(i, components[i] * scalar); }
113         //TODO.
114         return result;
115     }
```

$$s * \vec{v} = s * (a_1, a_2 \cdots a_n) = (s * a_1, s * a_2 \cdots s * a_n)$$

Aufgabe 1.6: Zeilenvektor

Implementieren Sie die Punktspiegelung eines Vektors am Koordinatenurspung, indem Sie die Implementierung des überladenen unären Operators - vervollständigen. Hierbei seien im Grunde genommen einfach die Vorzeichen aller Komponenten umzukehren.

```
…
134     // Point reflection
135     Vect operator-() const {
136         Vect result;
137         //TODO 6):
138         for (std::size_t i = 0; i < N; ++i) { result.set(i, -components[i]); }
139         //TODO.
140         return result;
141     }
```

$$-\vec{v} = (-1) * (a_1, a_2 \cdots a_n) = (-a_1, -a_2 \cdots -a_n)$$

Aufgabe 1.7: Zeilenvektor

Es soll außerdem die Berechnung des Standardskalarproduktes zweier Vektoren ermöglicht werden.
Vervollständigen Sie hierzu den entsprechenden Operator *.
Das Skalarprodukt wird hier als Rückgabewert zurückgegeben.

```
…
143     // Inner product
144     template <typename U, std::size_t M> typename std::common_type<T, U>::type
        operator*(const Vect<U, M>& other) const {
145         using ResultType = typename std::common_type<T, U>::type;
146         constexpr std::size_t max_dim = MAX(N, M);
147         ResultType result = 0;
148         //TODO 7):
149         for (std::size_t i = 0; i < max_dim; ++i)
            { result += static_cast<ResultType>(this->get(i)) *
            static_cast<ResultType>(other.get(i)); }
150         //TODO.
151         return result;
152     }
```

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i \cdot b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Aufgabe 1.8: Zeilenvektor

Es sei die Methode norm(*p*) berechnet die *p*-Norm des Vektors:

(a) norm(0) liefert die Summe aller Komponenten des Vektors.

(b) (b) norm(*p*) (p != 0) berechnet die *p*-Norm des Vektors.

Die *p*-Norm ist wie folgt definiert:

$$\|\vec{v}\|_p = \sqrt[p]{\sum_{k=1}^{n} [|v_i|^p]}$$

```
…
154     // Norm
155     double norm(int p = 2) const {
156         if (p < 0) {
157             T max_val = components[0];
158             for (const auto& value : components) {
159                 if (ABS(value) > ABS(max_val)) {
160                     max_val = value;
161                 }
162             }
163             return static_cast<double>(ABS(max_val));
…
```

```
…
164         } else {
165             double sum = 0;
166             //TODO 8):
167             if (p == 0) {
168                 for (const auto& val : components) {
169                     sum += static_cast<double>(val);
170                 }
171             } else {
172                 for (const auto& val : components) {
173                     sum += std::pow(std::abs(static_cast<double>(val)), p);
174                 }
175             //TODO.
176             return sum;
177         }
178     }
```

Aufgabe 1.9: Zeilenvektor

Implementieren Sie mithilfe von Norm, Skalarprodukt und den arithmetischen Operatoren die (euklidische) Distanzfunktion

$$dist(\vec{w}) \coloneqq \|\vec{w} - \vec{v}\|_2$$

sowie die Winkelfunktion

$$angle(\vec{w})$$

die den Winkel zwischen $\vec{w}$ und $\vec{v}$ berechnet.

```
…
182     // Distance
183     template <typename U, std::size_t M> double dist(const Vect<U, M>& other) const {
184         double result = 0;
185         //TODO 9b):
186         result = (*this - other).norm(2);
187         //TODO.
188         return result;
189     }
```

$$d(\vec{a},\vec{b}) = \|\vec{a}-\vec{b}\|_2 = \sqrt{\sum_{i=1}^{n}(a_i-b_i)^2}$$

```
…
191     // Angle
192     template <typename U, std::size_t M> double angle(const Vect<U, M>& other) const {
193         double result = 0;
194         //TODO 9c):
195         double dot = *this * other;
196         double magA = this->norm(2);
197         double magB = other.norm(2);
198         if (magA > 0 && magB > 0) { result = std::acos(dot / (magA * magB)); }
199         //TODO.
200         return result;
201     }
```

$$\alpha = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}|_2 \cdot |\vec{b}|_2}\right)$$

Definition *Komplexe Zahlen*

Die Menge der komplexen Zahlen $\mathbb{C}$ ist eine Erweiterung der reellen Zahlen.

Eine komplexe Zahl z wird definiert als $\mathrm{z} = \mathrm{a} + \mathrm{b} \cdot \mathrm{i}$.

- $\mathrm{a} \in \mathbb{R}$: Realteil von z
- $b \in \mathbb{R}$: Imaginärteil von z
- i : Imaginäre Einheit, $i^2 = -1$

Geometrische Interpretation *Komplexe Zahlen*

$$z = \begin{pmatrix} a \\ b \end{pmatrix} \text{ bzw. } \vec{z} = (a, b)$$

Eine komplexe Zahl z wird definiert als $\mathrm{z} = \mathrm{a} + \mathrm{b} \cdot \mathrm{i}$.

- $\mathrm{a} \in \mathbb{R}$: Realteil von z
- $b \in \mathbb{R}$: Imaginärteil von z
- i : Imaginäre Einheit, $i^2 = -1$

Aufgabe 2.1: Komplexe Zahlen

Vervollständigen Sie die Getter der Klasse Complex:
real() für den Realteil und imag() für den Imaginärteil.

```
…
256     // Getter
257     //TODO 1):
258     T real() const { return get(0); }
259     T imag() const { return get(1); }
260     //TODO.
```

$$\vec{z} = (a, b)$$

Aufgabe 2.2: Komplexe Zahlen

Die Methode conj() soll die komplexe Zahl konjugieren.

```
...
262         //TODO 2):
263         Complex conj() const { return Complex(real(), -imag()); }
264         //TODO.
```

$$\overline{z} = a - b \cdot i$$

Aufgabe 2.3: Komplexe Zahlen

Vervollständigen Sie ebenfalls den Setter polar(r, θ).

Dieser setzt den Real- und Imaginärteil der Zahl auf die kartesische Darstellung der polaren Parameter r, θ.

```
…
270     // Setter
271     //TODO 3):
272     void polar(double r, double theta) {
273         set(0, static_cast<T>(r * std::cos(theta)));
274         set(1, static_cast<T>(r * std::sin(theta)));
275     }
276     //TODO.
```

$$a = r \cdot \cos(\theta)$$
$$b = r \cdot \sin(\theta)$$

Aufgabe 2.4: Komplexe Zahlen

Implementieren Sie die Multiplikation zweier komplexer Zahlen, indem Sie die Implementierung des überladenen Operators * vervollständigen.

Die Multiplikation zweier komplexer Zahlen überschreibt in der Vererbungsstruktur das Skalarprodukt.

```
…
278     // Arithmetic
279     //TODO 4):
280     template <typename U> Complex<typename std::common_type<T, U>::type>
        operator*(const Complex<U>& other) const {
281         using ResultType = typename std::common_type<T, U>::type;
282         return Complex<ResultType>(real() * other.real() - imag() * other.imag(),
            real() * other.imag() + imag() * other.real());
283     }
284     //TODO.
```

$$z_1 \cdot z_2 = (ac - bd) + (ad + bc)i$$

Aufgabe 2.5: Komplexe Zahlen

Vervollständigen Sie analog die Implementierung der Division /.

```
…
286     //TODO 5):
287     template <typename U> Complex<typename std::common_type<T, U>::type>
        operator/(const Complex<U>& other) const {
288         using ResultType = typename std::common_type<T, U>::type;
289         ResultType denom = other.real() * other.real() + other.imag() * other.imag();
290         return Complex<ResultType>((real() * other.real() + imag() * other.imag()) /
            denom, (imag() * other.real() - real() * other.imag()) / denom);
291     }
292     //TODO.
```

$$\frac{z_1}{z_2} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} i$$