Ludwig-Maximilians-Universität München
Mathematisches Institut
Gregor Kleen

WiSe 2023/24
22. Februar 2024

Programmieren II für Studierende der Mathematik

Klausur — Lösungsvorschlag

Aufgabe	**1**	**2**	**3**	**4**	**5**	Σ
Punkte	26	26	12	24	8	96

	3 ECTS	**6 ECTS**
Bearbeitungszeit	60 Minuten	90 Minuten
Korrigierte Aufgaben	**1**, **2**, **3**	**1**, **2**, **3**, **4**, **5**

1. (26 Punkte) Wir betrachten geschlossene *Polygonzüge* in $\mathbb{C}$. Ein Polygonzug besteht aus einer Folge $(z_i)_{i=0,\dots,n-1}$ mit $n \in \mathbb{N}$ und $z_i \in \mathbb{C}$. Wir verstehen aufeinander Folgende Punkte z_i und z_{i+1}, bzw. z_{n-1} und z_0 als jeweils mit einer Strecke verbunden.

(a) Implementieren Sie eine Klasse `PolChain` deren Objekte jeweils einen Polygonzug über $\mathbb{C}$ speichern können. Für die Darstellung einer komplexen Zahl soll hierbei der entsprechende Datentyp aus der STL dienen. Als Approximation reeller Zahlen soll der eingebaute Datentyp `double` dienen. Verwenden Sie einen geeigneten Container-Datentyp aus der STL.

(b) Implementieren Sie in Ihrer Klasse `PolChain` die Konstruktoren für die folgenden Fälle, jeweils mit geeigneten Parametern:

i. Der leere Polygonzug (enthält 0 Punkte)

ii. Einen gegebenen Wert der gleichen container-Datenstruktur, wie Sie sie auch in `PolChain` verwenden, von Punkten

PolChain

```
class PolChain {
  public:
    vector<complex<double>> z;

    PolChain() {}
    PolChain(const vector<complex<double>>& z_): z(z_) {}
    PolChain Konstruktoren

    PolChain size
    PolChain length

    PolChain add

    winding
};
```

(c) Implementieren Sie eine konstante Methode `size` in Ihrer Klasse `PolChain`, die die Anzahl von Punkten n des Polygonzugs als Rückgabewert liefert.

PolChain size

```
int size() const { return z.size(); }
```

(d) Implementieren Sie eine konstante Methode `length` in Ihrer Klasse `PolChain`, die die Länge des Polygonzugs als Rückgabewert liefert.

PolChain length

```
int length() const {
  double l = 0;
  int n = z.size();
  if (n > 1)
    for (int i = 0; i < n; i++)
      l += abs(z[(i + 1) % n] - z[i]);
  return l;
```

```
}
```

(e) Überladen Sie den Zuweisungs-Additionsoperator für Objekte Ihrer Klasse `PolChain` semantisch sinnvoll.

PolChain add

```
PolChain& operator+=(const PolChain& p) {
  int n = size();
  z.resize(n + p.size());
  for (int i = 0; i < p.size(); i++)
    z[n + i] = p.z[i];
  return *this;
}
```

(f) Implementieren Sie in Ihrer Klasse `PolChain` zusätzlich Konstruktoren für die folgenden Fälle, jeweils mit geeigneten Parametern:

i. Die Eckpunkte eines regelmäßigen n-Ecks, wobei die Eckpunkte jeweils auf dem Einheitskreis um den Nullpunkt liegen

ii. Die Eckpunkte eines regelmäßigen n-Ecks, wobei die Eckpunkte jeweils auf einem Kreis mit Radius r um den Nullpunkt liegen

iii. Die Punkte $f(1), f(f(1)), \ldots, f^n(1)$ die entstehen durch $1, 2, \ldots, n$-facher Anwendung einer gegebenen Funktion $f\colon \mathbb{C} \to \mathbb{C}$ auf 1.

PolChain Konstruktoren

```
PolChain(int n, double r = 1): z(n) {
  for (int i = 0; i < n; i++)
    z[i] = polar(r, i * 2 * M_PI / n);
}
PolChain(int n, function<complex<double>(complex<double>)> f): z(n) {
  z[0] = 1;
  for (int i = 1; i < n; i++)
    z[i] = f(z[i - 1]);
}
```

(g) Erstellen Sie eine konstante Methode `winding` in Ihrer Klasse `PolChain` mit einem Parameter $q \in \mathbb{C}$ und Rückgabetyp `int`. Es soll zurückgegeben werden:

$$\frac{1}{2\pi i} \sum_{k=0}^{n-1} \ln\left(\frac{z_{k+1} - a}{z_k - a}\right) \qquad \text{wobei } z_n := z_0$$

Werfen Sie eine exception falls q auf dem Polygonzug liegt. Hierfür können Sie prüfen ob die relative Abweichung zwischen der Summe der Abstände von q jeweils zu z_i und z_{i+1} und dem Abstand zwischen z_i und z_{i+1} selbst unterhalb von $\varepsilon = 1 \cdot 10^{-6}$ liegt. Werfen Sie auch eine exception falls der Rückgabewert betragsmäßig um mehr als εL von der nächsten[1] reellen Zahl abweicht, mit $\varepsilon = 1 \cdot 10^{-6}$ und L der Länge des Polygonzugs (Sie können die Methode `length` verwenden auch wenn Sie sie nicht implementiert haben).

[1] Sie können an dieser Stelle einfach auf die reelle Achse projezieren

winding

```
int winding(complex<double> q) const {
  int n = size(), w;
  complex<double> s = 0;
  for (int k = 0; k < n; k++) {
    double d1 = abs(q - z[k]),
      d2 = abs(z[(k + 1) % n] - q),
      d3 = abs(z[(k + 1) % n] - z[k]);
    if (d1 + d2 - d3 < 1e-6 * d3)
      throw invalid_argument("Given point lies on polygonal chain");

    s += log((z[(k + 1) % n] - q) / (z[k] - q));
  }
  s /= 2 * M_PI * complex<double>{0,1};
  w = static_cast<int>(s.real() + 0.5);

  if (abs(complex<double>{static_cast<double>(w), 0} - s) > 1e-6 * length())
    throw runtime_error("Winding number not close enough to whole number");

  return w;
}
```

2. (26 Punkte) Nach Cayley und Dickson können Quartenionen $q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ dargestellt werden als Paar von komplexen Zahlen $q = (a + b\mathbf{i}) + (c + d\mathbf{i})\mathbf{j} = z + w\mathbf{j}$.

(a) Implementieren Sie eine Klasse `Quaternion` deren Objekte jeweils eine Quaternion in obiger Darstellung speichern können. Für die Darstellung einer komplexen Zahl sollen Sie hierbei den entsprechenden Datentyp aus der STL verwenden. Als Approximation reeller Zahlen soll der eingebaute Datentyp `double` dienen.

Quaternion

```
class Quaternion {
  public:
    complex<double> z, w;

    Quaternion Konstruktoren

    Quaternion Einausgabe

    Quaternion Multiplikation
    Quaternion Addition

    Quaternion Konjugation
};
```

(b) Implementieren Sie in Ihrer Klasse `Quaternion` die folgenden zwei Konstruktoren:

i. Einen Konstruktor der sich sowohl mit keinem, wie auch mit einem Argument vom Typ `double` aufrufen lässt.
Für ein Argument mit Wert x soll die Quaternion x (reelle Zahl eingebettet in die Quartenionen) erzeugt werden, ohne Argument die Quaternion 0.

ii. Einen Konstruktor mit vier Argumenten vom Typ `double` die a, b, c und d entsprechen, wie oben.

Quaternion Konstruktoren

```
Quaternion(double x = 0): z(x, 0), w(0, 0) {}
Quaternion(double a, double b, double c, double d): z(a, b), w(c, d) {}
```

(c) Überladen Sie die Ein- und Ausgabeoperatoren für Ströme geeignet, sodass Objekte Ihrer Klasse `Quaternion` in der Darstellung $+a+b\mathrm{i}+c\mathrm{j}+d\mathrm{k}$ eingelesen und ausgegeben werden können. Achten Sie darauf etwaige Fehler beim einlesen, durch eine Änderung am internen Zustand des Eingabestroms, geeignet zu kommunizieren.

Hinweis. Als Abtrennung zwischen den Komponenten kann direkt das jeweilige Vorzeichen der Komponenten a, b, c bzw. d fungieren.

Quaternion Einausgabe

```
friend ostream& operator<<(ostream& stream, const Quaternion& q) {
  return
    stream << showpos
           << q.z.real()
           << q.z.imag() << "i"
           << q.w.real() << "j"
           << q.w.imag() << "k";
}
friend istream& operator>>(istream& stream, Quaternion& q) {
  double a, b, c, d;
  char u1, u2, u3;
  stream >> a
         >> b >> u1
         >> c >> u2
         >> d >> u3;
  if (u1 != 'i' || u2 != 'j' || u3 != 'k')
    stream.setstate(ios::failbit);
  else
    q = Quaternion(a, b, c, d);
  return stream;
}
```

(d) Überladen Sie den Multiplikationsoperator für Objekte Ihrer Klasse `Quaternion` als mit der Klasse befreundete Funktion. Für $q_1 = z_1 + w_1\mathbf{j}$ und $q_2 = z_2 + w_2\mathbf{j}$ gilt:

$$q_1 \cdot q_2 = (z_1 w_1 - z_2\overline{w_2}) + (z_1 w_2 + z_2\overline{w_1})\mathbf{j}$$

Quaternion Multiplikation

```
friend Quaternion operator*(const Quaternion& q1, const Quaternion q2) {
  complex<double>
    z3 = q1.z * q1.w - q2.z * ::conj(q2.w),
    w3 = q1.z * q2.w + q2.z * ::conj(q1.w);
  return Quaternion(z3.real(), z3.imag(), w3.real(), w3.imag());
}
```

(e) Überladen Sie den Additions-Zuweisungsoperator für Objekte Ihrer Klasse `Quaternion` als Methode der Klasse. Die Addition erfolgt Komponentenweise.

Quaternion Addition

```
Quaternion& operator+=(const Quaternion& q) {
  z += q.z; w += q.w;
  return *this;
}
```

(f) Implementieren Sie eine konstante Methode `conj` für Ihre Klasse `Quaternion`, die keine (expliziten) Parameter akzeptiert und die zu q, der Quaternion auf der die Methode aufgerufen wird, $q^* = -\frac{1}{2}(q + \mathbf{i}q\mathbf{i} + \mathbf{j}q\mathbf{j} + \mathbf{k}q\mathbf{k})$ die *konjugierte* Quaternion als Rückgabewert liefert. Verwenden Sie hierfür den Multiplikations- und Additions-Zuweisungsoperator für `Quaternion`, auch dann, wenn Sie ihn nicht implementiert haben.

Quaternion Konjugation

```
Quaternion conj() const {
  Quaternion
    i = Quaternion(0, 1, 0, 0),
    j = Quaternion(0, 0, 1, 0),
    k = Quaternion(0, 0, 0, 1);
  Quaternion q = *this;
  q += i * *this * i;
  q += j * *this * j;
  q += k * *this * k;
  return -0.5 * q;
}
```

3. (12 Punkte) Welche Ausgabe produziert das folgende Programm?

Hinweis. Bei `uint8_t` handelt es sich um einen vorzeichenlosen Ganzzahltyp mit genau 8 bit.

exm_eval.cpp

```
#include <cstdint>
#include <iostream>

using namespace std;

int f1(int& i, int j) { j += --i; return i++; }
uint8_t f2(int& i, int j) { i -= j; return i; }
int& f3(int& i, int& j) { j -= 3*i; return i; }
int f4(int i, int j) { double x = i / j; return x; }

void print(int i, int j, int k) {
  cout << i << " " << j << " " << k << endl;
}

int main() {
  int i, j, k;
  i = 1; j = 2; k = f1(i, j); print(i, j, k);
  i = 1; j = 2; k = f2(i, j); print(i, j, k);
  i = 1; j = 2; k = f3(i, j); print(i, j, k);
  i = 1; j = 2; k = f4(i, j); print(i, j, k);
}
```

```
1 2 0
-1 2 255
1 -1 1
1 2 0
```

4. (24 Punkte) Wir betrachten die Cayley-Dickson-Konstruktion für 2^n dimensionale Algebren über $\mathbb{R}$. Für $a_0, b_0 \in \mathbb{R}$ bilden wir als Element der 2^1 dimensionalen Cayley-Dickson-Algebra das Tupel (a_0, b_0). Mit a_n, b_n Elemente der 2^n dimensionalen Cayley-Dickson-Algebra bilden wir als Element der 2^{n+1} dimensionalen Cayley-Dickson-Algebra das Tupel (a_n, b_n). Im Folgenden identifizieren wir Elemente (x, y) der 2^1 dimensionalen Cayley-Dickson-Algebra mit komplexen Zehlen $x + yi$.

(a) Vereinbaren Sie ein Klassen-template `CayleyDickson` mit einem Templateparameter n vom Typ `unsigned int`. Objekte der Klasse `CayleyDickson<n>` sollen zwei Attribute vom Typ `CayleyDickson<n - 1>` enthalten.

Spezialisieren Sie ihr template für den Fall $n = 1$ indem Sie ihre Klasse in diesem Fall von `complex<double>` ableiten.

Hinweis. Bei abgeleiteten Klassen wurden Sie angehalten den Destruktor speziell zu behandeln. Tun Sie dies sowohl für das template selbst, wie auch für seine Spezialisierung.

CayleyDickson

```
template<unsigned int dim>
  class CayleyDickson {
    private:
      using P = CayleyDickson<dim - 1>;
      using S = CayleyDickson<dim>;

    public:
      P a, b;

      virtual ~CayleyDickson() = default;

      CayleyDickson Gen Konstruktor
      CayleyDickson Gen Konvertierkonstruktor

      CayleyDickson Gen real

      CayleyDickson Ausgabe

      CayleyDickson Addition

      CayleyDickson conj

      CayleyDickson Multiplikation

      CayleyDickson norm
};
```

```
template<> class CayleyDickson<1>: public complex<double> {
  public:
    virtual ~CayleyDickson() = default;

    CayleyDickson Basis Konstruktor
    CayleyDickson Basis Konvertierkonstruktor

    CayleyDickson Basis norm
};
```

(b) Implementieren Sie einen Konstruktor für ihr template `CayleyDickson<n>`, der null, einen oder zwei Parameter vom Typ `CayleyDickson<n - 1>` akzeptiert. Wird ein Argument nicht angegeben, soll der Standardkonstruktor verwendet werden um einen Wert zu erzeugen.

CayleyDickson Gen Konstruktor

```
CayleyDickson(P a_ = P(), P b_ = P()): a(a_), b(b_) {}
```

(c) Implementieren Sie einen Konstruktor für Ihre Spezialisierung `CayleyDickson<1>` mit einem Parameter vom Typ `complex<double>` und einen weiteren Konstruktor mit null, einem oder zwei Parametern vom Typ `double`.

CayleyDickson Basis Konvertierkonstruktor

```
CayleyDickson(const complex<double>& x): complex<double>(x) {}
```

CayleyDickson Basis Konstruktor

```
CayleyDickson(double re_ = 0, double im_ = 0): complex<double>(re_, im_) {}
```

(d) Implementieren Sie einen weiteren Konstruktor für `CayleyDickson<n>` mit einem Parameter vom Typ `double`.

CayleyDickson Gen Konvertierkonstruktor

```
CayleyDickson(double re_): a(re_) {}
```

(e) Implementieren Sie eine konstante Methode `real` für `CayleyDickson<n>`. Es gilt $\mathrm{Re}\,((a_n, b_n)) = \mathrm{Re}(a_n)$.

CayleyDickson Gen real

```
double real() const { return a.real(); }
```

(f) Überladen Sie den Ausgabeoperator für Werte vom Typ `CayleyDickson<n>` geeignet. Es genügt rekursiv die interne Struktur des Objektes auszugeben.

CayleyDickson Ausgabe

```
friend ostream& operator<<(ostream& stream, const S& z) {
  return stream << "(" << z.a << "," << z.b << ")";
}
```

(g) Überladen Sie die arithmetischen Operatoren für unäre Negation, Subtraktion und Addition für `CayleyDickson<n>` komponentenweise.

CayleyDickson Addition

```
S operator-() const {
  return S{-a, -b};
}
friend S operator-(const S& z, const S& w) {
  return S{z.a - w.a, z.b - w.b};
}
friend S operator+(const S& z, const S& w) {
  return S{z.a + w.a, z.b + w.b};
}
```

(h) Wir bezeichnen den folgenden Ausdruck als *Konjugation* von (a_n, b_n):

$$(a_n, b_n)^* := (a_n^*, -b_n)$$

Implementieren Sie eine mit `CayleyDickson<n>` befreundete Funkton `conj` die die Konjugation des gegebenen Parameters als Rückgabewert liefert.

CayleyDickson conj

```
friend S conj(const S& z) {
  return S{conj(z.a), -z.b};
}
```

(i) Es gilt für die Multiplikation:

$$(a, b) \cdot (c, d) := (ac - d^* b, da + bc^*)$$

Überladen Sie den Multiplikationsoperator für `CayleyDickson<n>` entsprechend.

CayleyDickson Multiplikation

```
friend S operator*(const S& z, const S& w) {
  return S{
    z.a * w.a - conj(w.b) * z.b,
    w.b * z.a + z.b * conj(w.a)
  };
}
```

(j) Wir definieren:

$$\|(a, b)\| := (a, b) \cdot (a, b)^*$$

Implementieren Sie eine konstante Methode `norm` sowohl für `CayleyDickson<n>`, wie auch für `CayleyDickson<1>`, die die Norm des Objekts auf dem sie angewandt wurde als Rückgabewert liefert.

CayleyDickson norm

```
double norm() const {
  return (*this * conj(*this)).real();
}
```

CayleyDickson Basis norm

```
double norm() const { return abs(*this); }
```

5. (8 Punkte) Implementieren Sie ein Hauptprogramm, das alle in einer Datei enthaltenen Zeichenketten von *druckbaren Nichtzwischenraum*-Zeichen der Länge vier oder länger auf separaten Zeilen ausgibt. Es soll jeweils nur die Gesamtzeichenkette ausgegeben werden, nicht einzelne Teilzeichenketten (insb. wird jedes Zeichen maximal so oft ausgegeben, wie es in der Eingabedatei vorkommt).
Der Name der einzulesenden Datei soll dem Programm über den ersten Kommandozeilenparameter übergeben werden.

Achten Sie auf zumindest rudimentäre Fehlerbehandlung.

Hinweis. Für die Bestimmung ob ein Zeichen vom Typ `unsigned char` druckbar ist, können Sie die Funktion `bool isprint(unsigned char)` als gegeben betrachten. Analog für die Zwischenraum-Eigenschaft `bool isspace(unsigned char)`.

strings

```
int main(int argc, char* argv[]) {
  if (argc != 2) {
    cerr << "usage: " << argv[0] << " <FILE>" << endl;
    exit(2);
  }

  ifstream ein(argv[1]);
  if (!ein) { cerr << "Could not open: " << argv[1] << endl; exit(1); }

  ein >> noskipws;
  char c; string s;
  while (ein >> c) {
    if (isprint(c) && !isspace(c))
      s += c;
    else {
      if (s.size() >= 4) cout << s << endl;
      s = "";
    }
  }

  if (s.size() >= 4) cout << s << endl;
}
```