MAGMA WS 18/19

WERNER BLEY

Dieser Kurs richtet sich an Studierende meiner Vorlesung Algebra aus dem WS18/19 und orientiert sich inhaltlich an der Vorlesung. Im allgemeinen Teil dieses Skripts habe ich mich sehr eng an die MAGMA Kurzeinführung von Ulrich Thiel gehalten. Mit seiner freundlichen Genehmigung möchte ich auf sein Skript verweisen, das Sie unter

`http://www.mathematik.uni-stuttgart.de/~thiel/publications/magma_kurzeinfuehrung.pdf`

finden können. Es ist deutlich ausführlicher und geht tiefer auf die Konzepte vom MAGMA ein.

Weiterhin sehr hilfreich ist das MAGMA Handbuch, das Sie unter

`https://magma.maths.usyd.edu.au/magma/handbook`

finden.

1. Grundlagen

1.1. **Die ersten Schritte.** Von der Konsole wird MAGMA mit dem Kommando `magma` gestartet. Man kann MAGMA wie einen Taschenrechner benutzen. Jedes Kommando muss mit einem Semikolon abgeschlossen werden. Probieren Sie

```
2+3;
2*(-3);
(-3)^117;
3/7+2/17;
(1/13)/(-5/19);
```

MAGMA wird mit dem Befehl

```
quit;
```

verlassen.

1.2. **Strukturen von MAGMA.** Jedes Objekt hat in MAGMA einen wohldefinierten *Typ* und gehört einer darüber liegenden Struktur, dem *Parent*, an. Diese Informationen können wir folgt ermittelt werden.

```
> Type(5);
RngIntElt
> Parent(5);
Integer Ring
> Type(3/2);
FldRatElt
> Parent(3/2);
Rational Field
```

Date: 18. Oktober 2018.

Die Ausgaben sind weitgehend selbsterklärend. Die Befehle

```
Z := Integers();
Q := Rationals();
```

erzeugen die ganzen Zahlen bzw. die rationalen Zahlen. Versuchen Sie die folgenden Befehle.

```
a := 9/3;
IsPrime(a);
IsPrime(Z!a);
```

Wie der Name erraten läßt, testet die Funktion `IsPrime`, ob eine ganze Zahl eine Primzahl ist. Die Eingabe für `IsPrime` muss also vom Typ `RngIntElt` sein. Bei der Zuweisung `a := 9/3;` erzeugt aber MAGMA ein Objekt vom Typ `FldRatElt`, daher die Ausgabe *false*, obwohl ja $a = 3$ scheinbar offensichtlich eine Primzahl ist. Will man, daß MAGMA a als Element der ganzen Zahlen, so muss man dies MAGMA mittels dem Typen-Umwandlungsoperator `!` mitteilen. Versuchen Sie

```
Type(a);
Type(Z!a);
Z ! (3/4);
```

1.3. **Intrinsics.** MAGMA hat eine große Anzahl an vordefinierten Funktionen, sogenannten *intrinsics*. Wir haben schon mehrere davon kennen gelernt: `Integers(), Rationals(), IsPrime(n)`. Intrinsics haben eine bestimmte Anzahl von Eingabeobjekten von einem bestimmten Typ. Hier sind weitere Beispiele:

```
Divisors(2020);
PrimeDivisors(2020);
Factorization(2020);
GCD(12, 33);
```

Sowohl die Befehle als auch die Ausgaben sind weitgehend selbsterklärend. Wenn man sich darüber informieren will, wie die verlangte Eingabe ist und wie die Ausgabe zu interpretieren ist, so hilft neben dem MAGMA-Handbuch auch die Eingabe des Intrinsic-Namens abgeschlossen mit Semikolon weiter. Zum Beispiel

```
Divisors;
IsPrime;
```

Sie sehen, daß MAGMA den selben Namen für unterschiedliche Eingabetypen verwendet; MAGMA kann dann anhand der Anzahl und der Typen der Eingabeparameter erkennen, welche Funktion zu verwenden ist.

1.4. **Aussagenlogik.** MAGMA kennt den Typ `BoolElt`, wie wir schon bei der Funktion `IsPrime` gesehen haben. Einfache Beispiele sind

```
a := true;
b := false;
a or b;
a and b;
not a;
```

Sehr oft benötigt man für Vergleichstests `eq` für $=$, `lt` für $<$, `le` für $\leq$, `gt` für $>$ und `ge` für $\geq$.

1.5. **Konditionalausdrücke.** Konditionalausdrücke haben die Gestalt

```
if BoolAusdruck1 then
    Anweisungen 1;
elif BoolAusdsruck 2 then
    Anweisungen 2;
else
    Anweisungen 3;
end if;
```

Dabei kann es beliebig viele `elif`-Teile geben. Betrachten wir das folgende Beispiel. Wir wollen die Funktion $f\colon \mathbb{N}_{>0} \longrightarrow \mathbb{N}_{>0}$ mit

$$f(n) := \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ 3n+1, & \text{sonst.} \end{cases}$$

realisieren. Dazu betrachten wir

```
n := 35;
if IsEven(n) then n := Integers()!(n/2); else n := 3*n+1; end if; n;
```

Man kann einen solchen Ausdruck also auch in eine Zeile schreiben; beim Gebrauch von MAGMA als Taschenrechner hat dies gewisse Vorteile, ist aber eher unleserlich. Wenn wir später eigene Funktionen schreiben, werden wir daher (meist) eine Schreibeweise wie folgt bevorzugen

```
if IsEven(n) then
    n := Integers()!(n/2);
else
    n := 3*n+1;
end if;
```

Da man mit ↑ Befehle zurückholen kann, können Sie die Kurzversion des obigen Konditionalausdrucks nun wieder und wieder durchführen. Was beobachten Sie?

1.6. **Mengen.** Die Verwendung von Mengen in MAGMA ist sehr intuitiv. Wir wollen uns daher hier nur auf das notwendigste beschränken. Versuchen Sie

```
M := {1,2,3,4,5,6,7,8,9};
N := {2..5};
Type(M);
```

Die Kardinalität einer Menge ermittelt man mit `#`, `join` ergibt die Vereinigung und `meet` den Schnitt. Hier einige triviale Beispiele, allesamt selbsterklärend:

```
A := {1,2,3};
B := {3,4,5,6};
#A; #B;
A join B;
A meet B;
A diff B;
4 in A;
4 in B;
4 notin B;
{1,2} subset {1..10};
```

MAGMA erlaubt auch eine Art der Definition von Mengen, die dem Mathematiker intuitiv sofort klar ist. Wir erläutern dieses Konzept an Beispielen:

```
M := {1..20};
X := {a : a in M | IsEven(a)};
Y := {a : a in M | IsOdd(a) and a lt 11};
```

X ist also die Menge der Elemente $a \in M$ mit der Eigenschaft, daß a gerade ist. Y ist die Menge der $a \in M$, die ungerade und kleiner als 11 sind.

Wir wollen die Menge M der Primzahlen p bstimmen, die sich als Summe dreier Primzahlen a, b, c mit $3800 \leq a, b, c \leq 4100$ schreiben lassen. Sodann wollen wir entscheiden, ob es eine Primzahl zwischen $\min(M)$ und $max(M)$ gibt, die nicht in M liegt.

```
M := {a+b+c : a,b,c in {3800..4100} | IsPrime(a) and IsPrime(b) and IsPrime(c)
                                      and IsPrime(a+b+c)};
N := {n : n in {Min(M)..Max(M)} | IsPrime(n)};
N subset M;
M subset N;
N diff M;
```

Aufgabe 1.1. Verstehen Sie diese Zeilen. Warum dauert das relativ lang?

1.7. **Sequenzen.** *Sequenzen* funktionieren ähnlich wie Mengen, allerdings spielt jetzt die Reihenfolge eine Rolle und es sind Wiederholungen möglich. Der Datentyp heißt `SeqEnum`. Hier einige Beispiele:

```
S := [1,2,2,3,3,3,4,4,4,4,5,5,5,5,5];
#S;
S[1]; S[2];
S[1] := 17;
7 in S;
Type(S);
T := [100,101,102];
S cat T;
```

Der Befehl `cat` hängt also zwei Sequenzen aneinander, natürlich unter Beachtung der Reihenfolge.

Die Befehle `SetToSequence` bzw. `SequenceToSet` konvertieren zwischen Mengen und Sequenzen.

```
SetToSequence( {1,2,3,4} );
SequenceToSet( [1,2,2,3,3,3,4,4,4,4] );
```

1.8. **Funktionen und Intrinsics.** In diesem Abschnitt wollen wir lernen, wie man die Funktionalität von MAGMA erweitern kann, indem man eigene Funktionen und Intrinsics schreibt und einbindet.

Die Struktur von Funktionen ist wie folgt:

```
function_name := function(input_1, input_2,....)
    Anweisungen;
    return output_1, output_2, ....;
end function;
```

Hier ein einfaches Beispiel. Wir wollen eine Funktion f definieren, die $x \mapsto x^3$ realisiert.

```
f := function(x)
    return x^3;
end function;
```

```
f(2);
f(1.1);
```

Komplexere Funktionen will man natürlich nicht bei jedem Start von MAGMA neu schreiben. Wir erzeugen daher eine Datei MyFunctions.m (oder Sie wählen sich einen anderen Namen) und schreiben die Funktion in diese Datei. Mit `load ''MyFunctions.m''` kann man dann die Funktionen dieser Datei laden und in MAGMA benutzen.

Wir wollen die Funktion

$$f(n) := \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ 3n+1, & \text{sonst.} \end{cases}$$

als Function realisieren. Editieren Sie dazu die Datei `MyFunctions.m` und speichern Sie sie mit dem Inhalt

```
f := function(n)
    if IsEven(n) then
        n := Integers()!(n/2);
    else
        n := 3*n+1;
    end if;
    return n;
end function;
```

Nach `load ''MyFunctions.m''` können wir nun die Funktion f benutzen. Schleifen werden wir zwar erst kennen lernen, aber intuitiv ist klar, was in den folgenden Zeilen passiert.

```
while n ne 1 do
    n := f(n);
    n;
end while;
```

Für die nächste Aufgabe möchten Sie vielleicht eine `for`-Schleife verwenden. Wir greifen deshalb etwas vor und geben hier die einfachste Form einer solchen Schleife an:

```
for i:=n to m do
    Anweisungen;
end for;
```

Hier sollten n, m ganze Zahlen sein. Zum Beispiel können Sie mit der folgenden Schleife die 20 Primzahlen $> N$ in eine Sequenz `P` anfügen, wobei N eine natürliche Zahl ist.

```
P := [];
N := 2000000000000000000;
for i:=1 to 20 do
    N := NextPrime(N);
    Append(~P, N);
end for;
```

Aufgabe 1.2. Bekanntlich sind die Fibinacci-Zahlen rekursiv definiert durch

$$F_0 := 0, F_1 := 1; F_n := F_{n-1} + F_{n-2}, \text{ falls } n \geq 2.$$

Schreiben Sie eine Funktion `MyFibonacci(n)`, die die n-te Fibonacci-Zahl berechnet. Testen Sie Ihre Implementierung durch Vergleich mit der MAGMA-Funktion `Fibonacci`.

Oftmals ist es nicht sinnvoll, große Objekte, etwa sehr lange Sequenzen, an eine Funktion zu übergeben, da dann eine Kopie des Objekts angelegt wird. Statt dessen kann man auch einen Zeiger auf das Objekt übergeben. In diesem Fall spricht man in MAGMA von `Prozeduren`. Diese liefern keinen Wert zurück, erlauben es aber ein Objekt zu ändern. Prozeduren haben zwei verschieden Arten von Übergabeparametern, nämlich gewöhnliche Übergabeparameter wie in Funktionen sowie Referenzen (Zeiger) auf Objekte. Eine Referenz auf ein Objekt X übergibt man mittel $\sim X$. Hier ein Beispiel.

```
MyCat := procedure(~seq, n)
    seq := seq cat [n];
end procedure;

S := [1..10^5];
#S;
S[#S];
MyCat(~S, -1);
#S;
S[#S];
```

Wir kommen nun zu Intrinsics. Intrinsics sind Funtionen oder Prozeduren, die MAGMA so behandelt, als wären es feste Bestandteile von MAGMA. Daher verlangt die Implementierung auch etwas mehr Sorgfalt; MAGMA will genauere Informationen über die Funktion bzw. die Prozedur. Eine Intrinsic wird wie folgt defieniert

```
intrinsic Intrinsic_Name(Eingabe_1 :: Typ_1, ..., Eingabe_n :: Typ_n) -> AusgabeTyp_1, ..., A
{Beschreibung der Intrinsic.}
    Anweisungen;
    return Ausgabe_1, ..., Ausgabe_m;
end intrinsic;
```

Hier ein Beispiel. Wir realisieren die Fibonacci-Zahlen als Intrinsic.

```
intrinsic MyFibo(n :: RngIntElt) -> RngIntElt
{Meine Fibonaccizahlen.}
   if n eq 0 then return 0; end if;
   if n eq 1 then return 1; end if;
   a := 0; b := 1;
   for i:=2 to n do
       c := b;
       b := a+b;
       a := c;
   end for;
   return b;
end intrinsic;
```

Schreiben Sie diese Intrinsic in eine Datei *MyIntrinsics.m* und geben Sie in MAGMA den Befehl `Attach('MyIntrinsics.m')` ein. Damit wird die Funktion eingebunden. Versuchen Sie nun

```
MyFibo;
```

Eine Intrinsic akzeptiert auch Referenzen als Eingabeparameter, allerdings darf dann in der Intrinsic kein return-Statement enthalten sein. Eine MAGMA-eigene Intrinsic, wo wir dies oft verwenden werden, ist `Append(~ S, x)`. Hier wird an eine Liste S bestehend aus Objekten eines gewissen Typs eine weiteres Objekt x des gleichen Typs T angefügt. Ein Beispiel

```
LangeListe := [1..10^7];
#LangeListe;
Append(~LangeListe, -1);
#LangeListe;
LangeListe[ #LangeListe ];
```

1.9. **Schleifen.** Wir wollen uns hier auf `for`- und `while`-Schleifen beschränken und diese anhand von Beispielen kennenlernen. Zunächst zur `for`-Schleife.

```
for i:=0 to 99 do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

vergleicht Ihre Implementierung mit der MAGMA-eigenen Implementierung anhand der ersten hundert Fibonacci-Zahlen. Wenn Ihre Funktion richtig arbeitet, so bekommen Sie 100 mal den wahrheitswert `true`. Äquivalent dazu sind

```
for i in [0..99] do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

oder

```
for i in {0..99} do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

Auch `for`-Schleifen der folgenden Art sind erlaubt

```
for i,j in {1,2,3} do
    print "i = ", i, " j = ", j;
end for;
```

Hier haben wir eine etwas primitive Art der Ausgabe benutzt, die weitgehend selbsterklärend ist. Wenn nicht, so konsultieren Sie bitte das Handbuch.

Die Form einer `while`-Schleife ist wie folgt

```
while BoolElt do
    Anweisungen;
end while;
```

Wir betrachten wieder einige Beispiele. Wir wollen eine Liste der Primzahlen $\leq N$ erzeugen.

```
N := 200;
p := 2;
P := [];
while p le N do
    Append(~P, p);
    p := NextPrime(p);
end while;
```

Die folgende Schleife sucht die erste Primzahl $\geq N$, die kongruent zu a modulo m ist. Dabei sollte $ggT(a, m) = 1$ gelten.

```
N := 500;
a := 1;
m := 133;
p := NextPrime(N-1);
found := p mod m eq a;
while not found do
    p := NextPrime(p);
    found := p mod m eq a;
end while;
p;
```

Wir wollen aus den obigen Beispielen Intrinsics machen und diese dann mit `Attach('MyIntrinsics.m')` einbinden.

```
intrinsic MyPrimes(N :: RngIntElt) -> SeqEnum
{Sequenz der Primzahlen kleiner oder gleich N.}
    p := 2;
    P := [];
    while p le N do
        Append(~P, p);
        p := NextPrime(p);
    end while;
   return P;
end intrinsic;
```

Aufgabe 1.3. Beweisen Sie, dass die Bedingung $ggT(m, a) = 1$ notwendig ist. Das die Bedingung hinreichend für die Existenz einer Primzahl p mit $p \equiv a \pmod{m}$ ist, ist nicht-trivial. Tatsächlich gibt es unendlich viele solcher Primzahlen. Schreiben Sie ein

`intrinsic MyPrime(N :: RngIntElt, a :: RngIntElt, m :: RngIntElt) -> RngIntElt`,

das die kleinste Primzahl $p \geq N$ mit $p \equiv a \pmod{m}$ liefert.

2. Gruppentheorie, Teil 1

Wir wollen in diesem Abschnitt verschiedene Typen von Gruppen kennen lernen und verstehen, wie sie in MAGMA dargestellt werden und wie man mit ihnen umgehen kann.

Wir beginnen mit den endlichen abelschen Gruppen. Wie in der Vorlesung bezeichne C_n die zyklische Gruppe der Ordnung n. Hierzu beachte man, daß je zwei zyklische Gruppen der Ordnung n isomorph sind, so daß die Notation gerechtfertigt ist. In MAGMA wird die C_n durch den Befehl `CyclicGroup(GrpAb, n)` erzeugt. Auch der Aufruf `CyclicGroup(n)` ist korrekt, liefert aber keine Gruppe vom Typ `GrpAb`, sondern eine Gruppe vom Typ `GrpPerm`. Hierzu kommen wir später; zunächst wollen wir den Typ `GrpAb` genauer betrachten. Geben Sie

```
C := CyclicGroup(GrpAb, 14); C;
```

ein. MAGMA liefert uns eine abelsche Gruppe isomorph zu $\mathbb{Z}/14\mathbb{Z}$ zurück. Der Erzeuger kann mit `C.1` angesprochen werden. Schöner ist es unter Umständen, wenn man

```
C<a> := CyclicGroup(GrpAb, 14); C;
```

benutzt. Dann kann man auf den Erzeuger mittels `a` zugreifen. Versuchen sie die folgenden Eingaben, überlegen Sie sich aber bereits zuvor das Ergebnis.

```
Order(a);
Order(2*a);
Order(7*a);
8*a eq 22*a;
Order(3*a) eq Order(a);
{x : x in C} eq {i*a : i in [5..18]};
```

Wir betrachten nun abelsche Gruppen der Form $C_{n_1} \times \ldots \times C_{n_l}$ mit $l, n_1, \ldots, n_l \in \mathbb{N}$. Dies ist eine abelsche Gruppe der Ordnung $n_1 \cdots n_l$. Der Aufruf

```
AbelianGroup([n_1,..., n_l])
```

erzeugt diese Gruppe. Wie wir in der Vorlesung lernen werden, ist jede endliche abelsche Gruppe bis auf Isomorphie von dieser Form. Mit dem Kommando

```
A := AbelianGroup([2,3,5]);
```

erzeugen wir also eine abelsche Gruppe, die isomorph zu $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/3\mathbb{Z} \times \mathbb{Z}/5\mathbb{Z}$ ist. Die Erzeuger heißen `A.1, A.2` und `A.3`. Versuchen sie die folgenden Eingaben, überlegen Sie sich aber wieder vorab das Ergebnis.

```
[Order(A.1), Order(A.2), Order(A.3)];
A<a,b,c> := AbelianGroup([2,3,5]);
a eq A.1; b eq A.2; c eq A.3;
Order(a+b);
a+b eq a + 4*b;
Order(A);
```

Der Aufruf `AbelianGroup([n])` erzeugt ebensfalls die zyklische Gruppe C_n. Hierzu folgendes Beispiel:

```
A<a> := AbelianGroup([100]);
B<b> := CyclicGroup(GrpAb, 100);
bool, f := IsIsomorphic(A,B);
bool; f;
f(a);
```

Sicherlich haben Sie sich gewundert, warum `CyclicGroup` zwei Argumente hat. Wie bereits weiter oben erwähnt, ist auch `CyclicGroup(n)` korrekt, liefert aber eine Gruppe vom Typ `GrpPerm`. Versuchen Sie folgende Eingaben:

```
C<c> := CyclicGroup(100);
C;
Type(C);
IsIsomorphic(A,C);
```

MAGMA weigert sich also Gruppen von verschiedenem Typ auf Isomorphie zu testen. Mit dem Befehl `PermutationGroup(A)` kann man aus einer Gruppe A vom Typ `GrpAb` eine Gruppe vom Typ `GrpPerm` machen. Hier ein Beispiel

```
C<c> := CyclicGroup(100);
C;
Type(C);
IsIsomorphic(PermutationGroup(A), C);
```

`GrpPerm` steht für endliche Permutationsgruppen. Dies ist der Typ von Gruppen, für die sehr viele Probleme algorithmische Lösungen haben. Natürlich kann man diese Probleme auch für endliche abelsche Gruppen lösen. Hier sind die Probleme jedoch meist trivial oder haben eine einfache Lösung.

Permutationsgruppen sind Untergruppen einer symmetrischen Gruppe S_n. Ohne Einschränkung stellen wir uns die S_n meist als die Gruppe der Permutationen der Menge $\{1, \ldots, n\}$ vor. Wie wir in der in der Vorlesung sehen werden, kann man jede endliche Gruppe G als Permutationsgruppe darstellen (mit einem geeigneten n). Hier ist die Beweisidee, die wir später an Beispielen umsetzen wollen. Sei $\sigma \in G$. Dann defieren wir

$$f_\sigma \colon G \longrightarrow G, \quad \tau \to \sigma\tau.$$

Die Abbildung

$$G \longrightarrow S(G), \quad \sigma \to f_\sigma$$

ist eine Einbettung (d.h. ein injektiver Gruppenhomomorphismus) und wir könne G mit seinem Bild identifizieren. Hierbei bezeichnet $S(G)$ die Gruppe der Permutationen der Menge G. Numerieren wir die Gruppenelemente, so können wir natürlich $S(G)$ und $S_{|G|}$ identifizieren.

Wir müssen also zunächst die symmetrischen Gruppen S_n untersuchen. Sie werden in MAGMA durch `SymmetricGroup(n)` oder kurz durch `Sym(n)` erzeugt. Versuchen Sie

```
S := Sym(3);
Set(S);
```

Wie wir sehen, verwendet MAGMA für die Elemente der symmetrischen Gruppen die übliche Zyklenschreibweise. Im Gegensatz zur Vorlesung wird in MAGMA jedoch $\sigma\tau$ nicht wie "σ nach τ" interpretiert, sondern wie "τ nach σ". Grund hierfür ist die MAGMA-Konvention, daß Abbildungen stets von rechts wirken. Es gilt also zum Beispiel $(1,2)(2,3) = (1,3,2)$. Die entsprechende Eingabe in MAGMA ist

```
s := S ! (1,2)(2,3);
s;
```

Betrachten wir nun das folgende Beispiel.

```
C<a> := CyclicGroup(4); C;
C ! (1,2)(3,4);
C ! (1,3)(2,4);
```

C wird also dargestellt als Untergruppe der S_4 erzeugt von $a = (1,2,3,4)$. Da

$$\langle a \rangle = \{id, (1,2,3,4), (1,3)(2,4), (1,4,3,2)\}$$

erklären sich die obigen Ausgaben von MAGMA.

Weitere wichtige Gruppen, die in MAGMA als Permutationsgruppen dargestellt werden, sind die Diedergruppen. Das intrinsic `DihedralGroup(n)` erzeugt die Diedergruppe D_n mit $|D_n| = 2n$. Mit dem Befehl `AlternatingGroup(n)` erzeugt man die alternierende Guppe A_n.

Aufgabe 2.1. Testen Sie mit MAGMA, ob $D_3 \simeq S_3$ und $C_3 \simeq A_3$. Geben Sie gegebenenfalls einen expliziten Isomorphismus an.

Als letzten Typ von Gruppen wollen wir Matrixgruppen kennen lernen. Wir beschränken uns hier auf die Konstruktion einer $\mathrm{Gl}_n(\mathbb{F}_p)$ und ihrer Untergruppen. Hierbei bezeichnet $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ den Körper mit p Elementen. Ferner werden wir

die Quaternionengruppe der Ordnung 8 zunächst als Matrixgruppe, und darauf aufbauend als Permutationsgruppe realisieren.

Wir betrachten folgende Sequenz.

```
QX<x> := PolynomialRing( Rationals());
F<i> := NumberField(x^2+1);
M := GL(2,F);
E := M![1,0,0,1];
I := M![i,0,0,-i];
J := M![0,-1,1,0];
K := M![0,-i,-i,0];
Q := MatrixGroup<2,F|[I,J,K]>;
```

Zunächst definieren wir den Polynomring $\mathbb{Q}[x]$ über den rationalen Zahlen. Der zweite Befehl erzeugt die Gaußschen Zahlen $\mathbb{Q}(i)$. `M` ist die Gruppe der invertierbaren 2×2-Matrizen mit Einträgen in $F = \mathbb{Q}(i)$. Durch die folgenden Zeilen werden die Matrizen

$$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, I = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, J = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, K = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}$$

definiert, und schließlich in der letzten Zeile die Untergruppe der $\mathrm{Gl}_2(F)$, die durch I, J, K erzeugt wird.

Aufgabe 2.2. Überprüfen Sie mit Magma, ob Q mit der in Aufgabe 2 des 2.Übungsblatts definierten Quaternionengruppe

$$\left\langle \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} \right\rangle$$

übereinstimmt.

Die Befehle

```
IsAbelian(Q);
#Q;
```

zeigen, dass Q tatsächlich nicht-abelsch von der Ordnung 8 ist. Aus den Übungen wissen wir außerdem, daß es genau zwei nicht-abelsche Gruppen der Ordnung 8 (bis auf Isomorphie) gibt. Versuchen Sie

```
D := DihedralGroup(4);
IsIsomorphic(Q, D);
```

Wie zu erwarten war, weigert sich MAGMA, da `Q` vom Typ `GrpMat` ist und nicht wie `D` vom Typ `GrpPerm`. Wir wollen daher die Quaternionengruppe auch noch als Permutationsgruppe realisieren und verwenden dazu die obige Beweisskizze. Betrachten Sie

```
S8 := Sym(8);
Qelts := [g : g in Q];
v := [Index(Qelts, I*g) : g in Qelts]; v;
sigma := S8!v;
w := [Index(Qelts, J*g) : g in Qelts];
tau := S8!w;
H := sub<S8 | [sigma, tau]>;
#H;
D := DihedralGroup(4);
```

```
#D;
IsIsomorphic(D, H);
IsAbelian(H);
IsAbelian(D);
#D eq #H;
```

Aufgabe 2.3. Versuchen Sie die obige Befehlssequenz zu verstehen. Stellen Sie dazu den Zusammenhang zur obigen Beweisskizze her.

Wir wollen nun eine Funktion schreiben, die die D_4 als Permutationsgruppe erzeugt. Dazu stellen wir uns die D_4 als die Gruppe der Isometrien eines Quadrates vor. Wir numerieren die Ecken von 1 bis 4. Jede Isometrie ist durch die zugehörige Permutation der Ecken eindeutig bestimmt. Also können wir die D_4 als Untergruppe der S_4 darstellen. Aus der Vorlesung wissen wir, daß die D_4 durch die Drehung σ um 90 Grad und eine Spiegelung τ erzeugt wird. Dies sollte ausreichen, um die folgende Funktion zu verstehen.

```
MyD4 := function()
    S4 := Sym(4);
    sigma := S4 ! (1,2,3,4);
    tau := S4 ! (1,2)(3,4);
    return sub<S4 | [sigma, tau]>;
end function;
```

Wir testen die Funktion durch

```
G := MyD4();
G;
IsIsomorphic(G, DihedralGroup(4));
```

Aufgabe 2.4. Schreiben Sie Funktionen MyD5() und MyD6(), und allgemeiner eine Funktion MyDn(n).

Aufgabe 2.5. Schreiben Sie eine Funktion MyQ8(), die die Quaternionengruppe der Ordnung 8 als Permutationsgruppe erzeugt.

Aufgabe 2.6. Realisieren Sie MyDn(n) und MyQ8() ebenfalls als intrinsics und speichern Sie die intrinsics in MyIntrinsiscs.m ab. In den folgenden Sitzungen wollen wir diese Datei stets zur Verfügung haben.

LMU München, Theresienstrasse 39, 80333 München, Germany
E-mail address: bley@math.lmu.de