Eine sehr kurze Einführung in GAP

- GAP hilft beim Rechnen kleiner Beispiele, suchen nach Gegenbeispielen. GAP bietet
  - gute Möglichkeiten zum Rechnen mit (Permutations-)Gruppen, sehr viele häufig benötigte Funktionen sind in GAP schon verfügbar
  - große Bibliotheken von Gruppen, Charaktertafeln et cetera
  - Möglichkeiten zur Verwendung dieser in Funktionen
- GAP wird von den meisten Paket-Managern angeboten oder kann per

  `http://www.gap-system.org/`

  bezogen werden. Auf dieser Seite gibt es unter „Documentation“→ „Manuals“auch
  - ein „Tutorial“ (etwa 100 Seiten, kurze Einführung, unter anderem ”A First Session with GAP ”)
  - sowie das „Reference“ Manual (etwa 900 Seiten, sehr ausführlich, enthält fast alle verfügbaren Befehle der Standardinstallation mit Beispielen, gut durchsuchbar)

  sowohl als PDF- als auch als html-Version
- Gestart wird GAP durch die Eingabe von `gap` in der Konsole.
- Beendet wird GAP durch die Eingabe von `quit;`
- Jeder GAP-Befehl muss mit einem Strichpunkt abgeschlossen werden, falls man dies vergisst, so muss dies in einer der darauffolgenden Zeilen nachgeholt werden. Zwei Strichpunkte verhindern die Ausgabe der Antwort von GAP, was insbesondere bei langen Ausgaben hilfreich sein kann
- Nach einer „fehlerhaften Eingabe“ geht GAP in einen „`break loop`“ und zeigt als Prompt nun `brk>` (beziehungsweise, wenn man in tieferen „`break loop`“s ist, also in einem „`break loop`“ wieder ein Fehler aufgetreten ist, `brk_02>`, `brk_03>` und so weiter) an. Verlassen werden kann der innerste „`break loop`“ durch Eingabe von `quit;`
- Mitloggen der Ein- und Ausgabe, etwa in die Datei `Datei.log` im aktuellen Verzeichnis, durch `LogTo('Datei.log');`. Beenden des mitloggens durch `LogTo();`
- Mit der Taste „↑“ kommt man, wie in der Konsole, zu den letzten Eingaben der aktuellen Sitzung.
- Einmaliges Drücken der Tabulator-Taste vervollständigt den bisher eingegeben Befehl, soweit dies eindeutig ist. Zweimaliges Drücken zeigt alle möglichen Vervollständigungen an.
- Setzt man ein Fragezeichen direkt vor einen Befehl, so bekommt man Hilfe zu diesem Befehl angezeigt. Etwa liefert `?Comm;` Hilfe zum Befehl `Comm`, der Kommutatoren berechnet.
- Variablen wird, wie üblich, durch `:=` ein Wert zugewiesen.
- Rechnen mit Permutationen:
  - Permutationen werden als disjunkte Zykel ein- und ausgegeben: `f := (1,4)(2,5,3);` und `g := (2,3);` liefert die Permutationen

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 2 & 1 & 3 \end{pmatrix} \quad \text{und} \quad g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 2 & 4 & 5 \end{pmatrix}$$

- Verknüpft werden Permutationen durch `*`. Dabei bedeutet `f*g;` zuerst f, dann g. Hier liefert dieser Befehl das Resultat `(1,4)(2,5)`
- `()` stellt die Identität dar
- Durch `g^-1;` wird die Inverse von g berechnet. Durch `g^f;` wird die Konjugation $f^{-1}gf$ von g mit f berechnet. Diese Befehle liefern hier `(2,3)` beziehungsweise `(2,5)`
- Permuationsgruppen können zum Beispiel durch `G := SymmetricGroup(5);`, durch Angaben der Erzeuger `H := Group((2,3,4,5,6), (2,3));` oder durch Angabe einer Liste von Erzeugern `K := Group([(1,2,3),(3,4,5)]);` kreiert werden. Bekantermaßen erzeugen die bei H angegeben Permutationen eine zu S_5 isomorphe Gruppe, die allerdings auf der Ziffernmenge $\{2,3,4,5,6\}$ operiert. Da in `GAP` die Gruppe `SymmetricGroup(5)` auf den ersten fünf natürlichen Zahlen operiert, liefert die Abfrage `G = H;` hier das Resultat `false`
- Der Befehl `IsomorphismGroups(G, H);` prüft die Gruppen auf Isomorphie und liefert im Erfolgsfall einen Isomorphismus, sonst `fail`. Hier erhält man bei diesem Aufruf, da die Gruppen G und H wie oben definiert isomorph sind, folglich einen Isomorphismus
- `IsomorphicSubgroups(G, K);` prüft, ob G eine zu K isomorphe Untergruppe besitzt und liefert im Erfolgsfall eine Liste von Isomorphismen von K auf Untergruppen von G, sonst eine leere Liste. In diesem Fall ist das Resultat `[ [ (3,4,5), (1,2,3) ] -> [ (3,4,5), (1,2,3) ] ]` - gerade die natürliche Einbettung von $K = A_5$ in $G = S_5$

- Einige Befehle zum bestimmen der Untergruppenstruktur: Das Beispiel A_5
  - Der Befehl `CSG := ConjugacyClassesSubgroups(AlternatingGroup(5));` weist `CSG` eine Liste mit den Konjugiertenklassen der Untergruppen von A_5 zu (insbesondere alle Isomorphietypen von Untergruppen) und liefert hier als Ergebnis

    ```
    [ Group( () )^G, Group( [ (2,3)(4,5) ] )^G, Group( [ (3,4,5) ] )^G,   Group(
    [ (2,3)(4,5), (2,4)(3,5) ] )^G, Group( [ (1,2,3,4,5) ] )^G,   Group( [ (3,4,5),
    (1,2)(4,5) ] )^G, Group( [ (1,2,3,4,5), (2,5)(3,4) ] )^G,   Group( [ (2,3)(4,5),
    (2,4)(3,5), (3,4,5) ] )^G,   AlternatingGroup( [ 1 .. 5 ] )^G ]
    ```

  - Auf den i-ten Listeneinträge kann man durch `CSG[i];` zugreifen.
  - Offenbar liefert `CSG[1];` gerade die Konjugiertenklasse der trivialen Gruppe. Um sich, etwa durch die Ordnung der Gruppe, nochmal davon zu berzeugen, kann man die Ordnung durch den Befehl `Order` (dieser gibt die Ordnung der Gruppe zurück) beziehungsweise durch den Befehl `Size` (hier wird die Gruppe als Liste ihrer Elemente aufgefasst und die Länge dieser Liste zurückgegeben) ausgeben lassen. Man erhält die Ordnung also durch `Order(CSG[1][1]);` - dabei wird die Konjugiertenklasse `CSG[1]` als Liste (hier mit einem Element, da in dieser Konjugiertenklasse nur eine Untergruppe liegt) aufgefasst und durch die zusätzliche `[1]` auf das erste Element der Liste zugegriffen.
  - Einen Hinweis auf den Isomorphietyp der Untergruppe kann die Ordnung geben. Hier will man also eine Funktion, die einer Konjugiertenklasse von Untergruppen die Ordnung einer dieser Untergruppen zuweist. Kurzformen von Funktionen mit nur einem Parameter können in `GAP` etwa folgendermaßen erstellt werden: `CCS -> Order(CCS[1]);`. Dabei wird die Konjugiertenklasse `CCS` als Liste aufgefasst, das erste Element ausgewählt und dann dessen Ordnung bestimmt. Mit Hilfe von `List(CSG, CCS -> Order(CCS[1]));` wird nun die Liste, welche als erstes Argument übergeben wird durchlaufen, und auf jedes Element dieser Liste die Funktion, welche als zweites Argument übergeben wird angewendet. Hier erhält man nun `[ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]`

- Eine kurze Funktion: Gibt es Quadratwurzeln?
  - Will man mit Hilfe von `GAP` untersuchen, ob man in einer konkreten endlichen Gruppe G Quadratwurzeln ziehen kann, ob also alle Elemente Quadrate sind, so kann man eine Funktion `SquareIsOnto(G)` schreiben, die die Gruppe G als Argument übergeben bekommt und `true` zurückliefert, falls die Abbildung $g \mapsto g^2$ surjektiv ist, `false` andernfalls
  - Solche Funktionen kann man in Dateien speichern und diese dann einlesen. Will man solch eine Datei aus `GAP` heraus anlegen, so kann während einer `GAP`-Sitzung durch den Befehl `EDITOR := ''emacs'';` einen Editor auswählen (dies kann auch `GAP` bei jedem Start automatisch machen, dafür muss dieser Befehl in der `.gaprc` stehen, siehe "Reference" Manual). Durch den Befehl `Edit(''SquareIsOnto.g'');` wird die Datei `SquareIsOnto.g` im aktuellen Verzeichnis geöffnet, falls diese existiert, sonst wird eine leere Datei angelegt und geöffnet. Schließt man den Editor, so werden automatisch die Befehle in dieser Datei eingelesen. In diese Datei kann man nun Beispielsweise folgende Funktion schreiben:

```
SquareIsOnto := function(G)
local squares, x;   # Deklaration der lokalen Variablen
squares := [];      # Liste, in die alle Quadrate eingetragen werden
for x in G do       # fuer jedes Element aus G das Quadrat in squares
                    # eintragen
    Add(squares, x^2);
od;
squares := DuplicateFreeList(squares);
                    # mehrfache auftretende Elemente loeschen
if Size(squares) = Order(G) then
    return true;
else
    return false;
fi;
end;
```

    Kürzer geht es etwa durch

```
SquareIsOnto := function(G)
return Size(DuplicateFreeList(List(G, g -> g^2))) = Order(G);
end;
```

  - Ist etwa die Datei `SquareIsOnto.g` schon fertig erstellt, so kann diese einfach durch den Befehl `Read(''SquareIsOnto.g'');` eingelesen werden.

- Weiter nützliche `GAP`-Funktionen, die in manchen Aufgaben des Praktikums nützlich sein können:
  - Spezielle Gruppen:
    - `CyclicGroup(n)` liefert eine zyklische Gruppe der Ordnung n
    - `AlternatingGroup(n)` liefert eine A_n
    - `DihedralGroup(2n)` liefert die Diedergruppe D_n (die 2 ist kein Tippfehler!)
    - `GL(n,q)`
    - `PSL(n,q)`

- Spezielle Eigenschaften
  - `AllSubgroups(G)` liefert eine Liste aller Untergruppen von G
  - `NormalSubgroups(G)` liefert eine Liste aller Normalteiler von G
  - `SylowSubgroup(G,p)` liefert eine p-Sylowgruppe von G
  - `Centralizer(G,U)` liefert den Zentralisator von U in G
  - `Normalizer(G,U)` liefert den Normalisator von U in G
  - `Centre(G)` liefert das Zentrum von G
  - `DerivedSubgroup(G)` liefert die Kommutatorgruppe G'
- Test von Gruppeneigenschaften
  - `IsSolvable(G)` prüft G auf Auflösbarkeit
  - `IsPGroup(G)` prüft, ob G eine p-Gruppe ist
  - `IsCyclic(G)` prüft, ob G zyklisch ist
  - `IsAbelian(G)` prüft, ob G abelsch ist
  - `IsNormal(G,N)` prüft, ob N ein Normalteiler von G ist
- Rechnen mit ganzen Zahlen
  - `IsPrimeInt(n)` testet eine ganze Zahl n auf Primalität
  - `FactorsInt(n)` gibt alle Primfaktoren einer Zahl n mitsamt Multiplizitäten aus
  - `PartialFactorization(n, a)` faktorisiert eine Zahl n je nach Einstellung a nicht komplett
  - `a = b mod c` testet, ob a kongruent b modulo c ist
  - `PowerModInt(a,m,b)` gibt einen Represäntanten einer Restklasse von $a^m \mod b$ an
  - `ZmodnZ(n)` liefert den Restklassenring $\mathbb{Z}/n\mathbb{Z}$
- Weiteres:
  - `StructureDescription(G)` liefert eine Beschreibung von G, falls verfügbar
  - `Order(G), Order(g)` liefert die Ordnung einer Gruppe bzw. eines Elements
  - `AllGroups(n)` liefert eine Liste aller Gruppen von Ordnung n bis auf Isomorphie
  - Ein endlicher Körper der Ordnung q wird mit `GF(q)` geschrieben und ist multiplikativ von dem Element `Z(q)` erzeugt.