Getting started in GAP

GAPDays Summer 2025

Meike Weiss and Lukas Schnelle

August 2025

Minicourse 1

1. Getting started
   a) Basic Basics
   b) Basic Programming
2. Programming in GAP: Working on problems from scratch
3. GAP Packages and Libraries: Using existing GAP infrastructure

Lectures and Exercises:

What is GAP?

- GAP stands for Groups, Algorithms and Programming
- System for computational discrete algebra

What is GAP?

- GAP stands for Groups, Algorithms and Programming
- System for computational discrete algebra

Why use GAP?

What is GAP?

- GAP stands for Groups, Algorithms and Programming
- System for computational discrete algebra

Why use GAP?

Programming Language:

- Open-source
- Interactive use and scripting
- Garbage collecter
- Break loop (for debugging)

What is GAP?

- GAP stands for Groups, Algorithms and Programming
- System for computational discrete algebra

Why use GAP?

Programming Language:

- Open-source
- Interactive use and scripting
- Garbage collecter
- Break loop (for debugging)

Mathematical capabilities:

- Large library of implementations of various algebraic algorithms
- Databases of groups, character tables and much more
- Separate packages of additional functions

Installing GAP

- Available for Linux, macOs and Windows
- Installation guide on the GAP website `https://www.gap-system.org/install/`
- Ask for help!

GAP Session

Start GAP Session:

```
meike@DESKTOP-AR8T0J7:~$ gap
  ┌─────┐   GAP 4.14.0 of 2024-12-05
  │ GAP │   https://www.gap-system.org
  └─────┘   Architecture: x86_64-pc-linux-gnu-default64-kv9
 Configuration:  gmp 6.2.0, GASMAN, readline
 Loading the library and packages ...
 Packages:   AtlasRep 2.1.9, AttributeScheduler 0.1, AutoDoc 2023.06.19, Browse 1.8.21,
             CTblLib 1.3.9, datastructures 0.3.1, Digraphs 1.10.0, FactInt 1.6.3, FGA 1.5.0,
             Forms 1.2.12, GAPDoc 1.6.7, genss 1.6.9, GRAPE 4.9.2, IO 4.9.1,
             NautyTracesInterface 0.3, orb 4.9.1, PrimGrp 3.4.4, recog 1.4.3,
             SimplicialSurfaces 0.7, SmallGrp 1.5.4, SpinSym 1.5.2, StandardFF 1.0,
             TomLib 1.2.11, TransGrp 3.6.5, utils 0.85
 Try '??help' for help. See also '?copyright', '?cite' and '?authors'
gap>
```

End GAP Session:

```
gap> quit;
```

Documentation

- Tutorial includes a first introduction
- Reference Manual includes complete descriptions and examples of all functions
- Use `?` to access documentation related to specific commands in a GAP session

```
gap> ?CyclicGroup;
gap> ?SymmetricGroup;
```

Syntax

- Every command should end with a semicolon `;` (or `;;` if the output should not be printed)
- Use `:=` for assignments of variables

```
gap> 1 + 1;
2
gap> x := 1 + 1;;
gap> x;
2
```

Standard Arithmetic: `+`, `-`, `*`, `/`, `^`, `mod`

Comparison Operators: `=`, `<>`, `<`, `>`, `≤`, `≥`

Booleans: `true`, `false`

Lists

- Collection of elements (numbers, list or other objects)
- List can contain different types of elements (but not a good practice)
- Index starts from 1
- More functionalities in the documentation

Lists

- Collection of elements (numbers, list or other objects)
- List can contain different types of elements (but not a good practice)
- Index starts from 1
- More functionalities in the documentation

```
gap> L := [4, 1, 3];;
gap> L[1];
4
```

Lists

- Collection of elements (numbers, list or other objects)
- List can contain different types of elements (but not a good practice)
- Index starts from 1
- More functionalities in the documentation

```
gap> L := [4, 1, 3];;
gap> L[1];
4
```

→ Sets are lists without repetitions

Lists

```
gap> M := [1..3];;
```

Lists

```
gap> M := [1..3];;
gap> M[2];
2
```

Lists

```
gap> M := [1..3];;
gap> M[2];
2
gap> M[1] := 4;;
gap> M;
[ 4, 2, 3 ]
```

Lists

```
gap> M := [1..3];;
gap> M[2];
2
gap> M[1] := 4;;
gap> M;
[ 4, 2, 3 ]
gap> Add(M,8);
gap> M;
[ 4, 2, 3, 8 ]
```

Lists

```
gap> M := [1..3];;
gap> M[2];
2
gap> M[1] := 4;;
gap> M;
[ 4, 2, 3 ]
gap> Add(M,8);
gap> M;
[ 4, 2, 3, 8 ]
gap> Remove(M,1);
4
gap> M;
[ 2, 3, 8 ]
```

Lists

```
gap> M := [1..3];;
gap> M[2];
2
gap> M[1] := 4;;
gap> M;
[ 4, 2, 3 ]
gap> Add(M,8);
gap> M;
[ 4, 2, 3, 8 ]
gap> Remove(M,1);
4
gap> M;
[ 2, 3, 8 ]
gap> Maximum(M);
8
```

Copy of List

Use `ShallowCopy` to make a copy that can be modified without changing the original

```
gap> L := [4, 1, 3];;
```

Copy of List

Use `ShallowCopy` to make a copy that can be modified without changing the original

```
gap> L := [4, 1, 3];;
```

```
gap> M := L;;
gap> Add(M,6);;
gap> L;
[ 4, 1, 3, 6 ]
```

Copy of List

Use `ShallowCopy` to make a copy that can be modified without changing the original

```
gap> L := [4, 1, 3];;
```

```
gap> M := L;;
gap> Add(M,6);;
gap> L;
[ 4, 1, 3, 6 ]
```

```
gap> N := ShallowCopy(L);;
gap> Add(N,6);;
gap> L;
[ 4, 1, 3 ]
```

Loops

Different options: for, while or repeat loop

Loops

Different options: for, while or repeat loop

```
gap> for i in [1..2] do
> Print(i^2,"\n");
> od;
1
4
```

Loops

Different options: for, while or repeat loop

```
gap> for i in [1..2] do
> Print(i^2,"\n");
> od;
1
4
```

```
gap> i := 1;;
gap> while i <= 2 do
> Print(i^2,"\n");
> i := i + 1;
> od;
1
4
```

Loops

Different options: for, while or repeat loop

```
gap> for i in [1..2] do
> Print(i^2,"\n");
> od;
1
4
```

```
gap> i := 1;;
gap> while i <= 2 do
> Print(i^2,"\n");
> i := i + 1;
> od;
1
4
```

→ break, continue and return to exit a loop earlier

Conditional Statements

```
gap> n := 7;;
gap> if n mod 2 = 0 then
> Print("Even");
> else
> Print("Odd");
Odd
> fi;
```

Conditional Statements

```
gap> n := 7;;
gap> if n mod 2 = 0 then
> Print("Even");
> else
> Print("Odd");
Odd
> fi;
```

→ Use `elif` for else if statement

Read Files

- Often helpful to write commands or function in files
- .g is common file type for GAP code
- Instead of writing in the terminal, read these files

Read Files

- Often helpful to write commands or function in files
- .g is common file type for GAP code
- Instead of writing in the terminal, read these files

```
gap> Read("FirstSquares.g");
```

a) Consider one of the provided files (`https://github.com/MeikeWeiss/GAP-Days2025-Intro/tree/master/Exercise%201/Exercise1a`), read the code and find the (syntax) errors by loading it in your GAP session.
   - FirstSquares
   - Faculty
   - Signum
   - SortList

b) Lists:
   - Compute the sum of the first 100 numbers using a for (and while) loop.
   - Define a list of integers and compute the list consisting of their squares. Try do to this just by using one command.
   - Define a list of integers and compute the sublist consisting of those that are even. Try do to this just by using one command.

c) Groups:
   - Let G be the group generated by $(1,2,3,4),(5,6,7,8),(1,5)(2,6)(3,7)(4,8)$. Compute the order of G and show that G is not abelian. Additionally, compute the center of G and show that it is a cyclic group of order four and that it has index 8.
   - Given a set S of elements in a given group, compute a smaller subset consisting of S-conjugate representatives (within S). (Intermediate)
   - More exercises can be found here https://www.ilariacolazzo.info/gap/tutorials/sheet2/.

d) Matrices:
   - Create a square matrix M and a vector v and compute $M * v$ and $v * M$.
   - Determine the determinant, the eigenvalues and the eigenvectors.

Functions

Named parts of code, that can be easily reused. They can have inputs for use in the function.

Functions

Named parts of code, that can be easily reused. They can have inputs for use in the function.

```
gap> timesThree:=function(x)
> return 3*x;
> end;;
```

Functions

Named parts of code, that can be easily reused. They can have inputs for use in the function.

```
gap> timesThree:=function(x)
> return 3*x;
> end;;
gap> timesThree(5);
15
```

Often helpful when loading functions from files.

`percentage.g`:

```
plusTenPercent := function(x)
return x*(1.1);
end;;
```

Often helpful when loading functions from files.

`percentage.g`:

```
plusTenPercent := function(x)
return x*(1.1);
end;;
plusPercent := function(x, y)
local decimal;
decimal := (1+(y/100));
return x*decimal;
end;;
```

Often helpful when loading functions from files.

`percentage.g`:

```
plusTenPercent := function(x)
return x*(1.1);
end;;
plusPercent := function(x, y)
local decimal;
decimal := (1+(y/100));
return x*decimal;
end;;
```

Then in GAP:

```
gap> Read("percentage.g");
gap> plusTenPercent(15);
16.5
gap> plusPercent(15, 5);
63/4
```

Fibonacci Sequence

Let us write a function, that computes a given element in the Fibonacci sequence. `fib.g`:

```
fibonacciNumber:=function(n)
if n = 0 then
return 0;
elif n = 1 then
return 1;
else
return fibonacciNumber(n-1) + fibonacciNumber(n-2);
fi;
end;;
```

Now let us run this code:

```
gap> Read("fib.g");
gap> fibonacciNumber(5);
5
gap> fibonacciNumber(12);
144
```

What happens for other numbers?

```
gap> fibonacciNumber(-5);
Error, recursion depth trap (5000) in
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
... at *stdin*:9
you may 'return;'
brk>
```

Break loop

Opens when the code encouters an error.

Sometimes can be continued (by typing `return;`) and sometimes not.

Allows interaction with the variables at the current state.

Can be called manually with `Error("text to show");`

Let us consider the example from before:

```
gap> fibonacciNumber(-5);
Error, recursion depth trap (5000) in
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
fibonacciNumber( n - 1 ) at fib.g:7 called from
... at *stdin*:9
you may 'return;'
brk> n;
-5003
brk>
```

Let us call `Error(...)` directly if the input is not valid:
`fibv2.g`:

```
fibonacciNumber:=function(n)
if n < 0 then
Error("this function does not work for negative numbers");
fi;
if n = 0 then
return 0;
elif n = 1 then
return 1;
else
return fibonacciNumber(n-1) + fibonacciNumber(n-2);
fi;
end;;
```

```
gap> Read("fibv2.g");
gap> fibonacciNumber(-5);
Error, this function does not work for negative numbers at
fibv2.g:3 called from
<function "fibonacciNumber">( <arguments> )
called from read-eval loop at *stdin*:110
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk>
```

Write functions, that accomplish the following. Also test them for a sensible number of inputs, so that the correctness is somewhat ensured.

Easy

- The Wythoff function, i.e. a generalisation of the Fibonacci function where the starting integers can be freely chosen
-* Compute the greatest common divisor by using the Euclidean algorithm
- A **FizzBuzz** function, i.e. takes an integer n as input and returns a list with n entries, where entry i is
  (i) `FizzBuzz` if i is divisible by 3 and 5
  (ii) `Fizz` if i is divisible by 3
  (iii) `Buzz` if i is divisible by 5
  (iv) i if none of the above are true
- A palindrome checker, i.e. for an input string if the reverse of that string is the same.

Intermediate

- A function that solves the word problem in $\mathbb{Z}/n\mathbb{Z}$ for a given integer n and list of generators. E.g. find a word $(a_i)_{1\leq i\leq k} \in \{3, 5\}$ such that $\left(\left(\sum_{i=1}^{k} a_i\right) \bmod n\right) = t$ for a provided target t.
-* A function which computes the sign of a given permutation, which is of type permutation.

* These functions do have built in equivalents, which can be used to check whether your function works as expected.