# Group theory in SAGE

David Joyner and David Kohel

2-18-2008

**Abstract**

SAGE is an open source computer algebra system implemented using an object-oriented categorical framework, with methods for objects, methods for their elements, and methods for their morphisms. Currently, SAGE has the ability to deal with abelian groups, permutation groups, and matrix groups over a finite field. This paper will present an overview of the implementations of the group-theoretical algorithms in SAGE, with some examples. We conclude with some possible future directions.

SAGE [**?**] is a general purpose computer algebra system started in 2005, built on top of existing open source packages, including GAP for group theory, Maxima for symbolic computation, Pari for number theory, and Singular for multivariate polynomial computations and commutative algebra. In design, SAGE uses the best of the ideas in Magma [**?**], Mathematica [**?**] and other systems, but uses the popular mainstream language Python as its interpreter. This paper will restrict itself to presenting an overview of the implementations of the group-theoretical algorithms in SAGE.

According to the GAP website, each year between 50 and 100 papers are published which use GAP in an essential way. SAGE is far too young to have such an impressive research record. Still, it has been already used by several people for published research in coding theory, number theory, and modular forms, as well as being used in teaching both graduate and undergraduate math classes.

Currently, SAGE has the ability to deal with abelian groups (finitely generated multiplicative abelian groups, groups of Dirichlet characters, and dual groups of finite abelian groups), permutation groups, matrix groups

over a finite field, and congruence subgroups. As a recreational aside, some algorithms enabling one to model the Rubik's cube using group theory are also included. For example, three fast optimized solvers are included with SAGE. The sections below will deal with these classes of groups separately.

In rough design, group theory in SAGE is implemented in an object-oriented categorical framework, with methods for group objects $G$ and $H$, methods for their elements $g$ and $h$, methods for their set of morphisms $\mathrm{Hom}(G, H)$ and methods for the element homomorphisms $\phi : G \to H$. For instance, a permutation group $G$ would be an object which has, for example, a method called `order`, which returns $|G|$. This class with its methods, are collected into a permutation group Python "module". Likewise the class of permutations $g \in G$ also has a method called `order`, which computes the smallest $n > 0$ for which $g^n = 1$. The class of permutations with associated methods, are collected into a permutation group element "module". Similarly, there exist a class for the sets $\mathrm{Hom}(G, H)$ of group homomorphisms between groups $G$ and $H$, with associated methods such as `domain` and `codomain`. When both $G$ and $H$ are both permutation groups, the elements of $\mathrm{Hom}(G, H)$ belong to a class of permutation group homomorphisms, and its associated member functions such as `kernel` are collected into a permutation group morphism "module". The SAGE source code contains comprehensive documentation on these modules.

We conclude with a section on possible future directions of SAGE and group theory.

# 1 The GAP interface

For the most part, SAGE's high-level group-theoretic capabilities are derived from the extensive functionality of the computer algebra system GAP. Fast low-level arithmetic is achieved by native code in C or Python. SAGE communicates with GAP and the other components using pseudotty's and a Python package called `pexpect` [**?**]. The SAGE/Python functions which call GAP using `pexpect` are called "wrappers".

- `Pexpect:` *makes Python a better tool for controlling other applications.* (`pexpect.sourceforge.net`)

  Pexpect is a pure Python module for spawning child applications; controlling them; and responding to expected patterns in their output.

`gap.eval('gapcommand')` sends `'gapcommand'` to GAP

For example[1],

SAGE

```
sage: gap.eval('2+3')
'5'
```

evaluates $2+3$ in GAP and returns the answer as a string. Some such GAP "string" output can be used in SAGE using the `eval` command. For example, integers and lists can (`eval('5')+1` returns `6`), but also polynomials in some cases. Except for these simple data structures, in most cases, GAP output cannot be used directly in SAGE and conversely.

- `Pseudotty:` A device which appears to an application program as an ordinary terminal but which is *in fact* connected to a different process. Pseudo-ttys have a slave half and a control half.

  `gap_console()` brings up a GAP prompt in SAGE

  For example,

  SAGE

  ```
  sage: gap_console()
  GAP4, Version: 4.4.10 of 02-Oct-2007, x86_64-unknown-linux-gnu-gcc
  gap> 2+3;
  5
  gap>
  ```

  brings up GAP inside SAGE. There is no preparsing.

A functional understanding of pseudotty's and pexpect is not needed to use SAGE, however, it helps to understand the functioning of SAGE commands.

For objects and morphisms, but not elements, SAGE uses Python wrappers of GAP functions, which it communicates to using `pexpect`. For many high level algorithms, for example the computation of derived series, there is

[1]Note that usually GAP requires a semi-colon at the end, but single semi-colons are not needed inside a `gap.eval` command. If you want to suppress the output then you do need to use a double semi-colon though.

no noticable overhead. For some low level operations, such as computations with permutation group elements, SAGE has a native optimized compiled implementation (still not as fast as GAP's implementation, written in C code). However, for almost all matrix group and abelian group operations, SAGE currently passes the computation to GAP and returns the result via `pexpect`.

# 2 Abelian groups

Finitely generated abelian groups are supported, both finite and infinite, with a multiplicative notation for elements. For the finite abelian groups, we simply wrap the appropriate GAP functions. However, for the infinite abelian groups, the corresponding GAP functions are located in a GAP package which had ambiguous licensing, so the code could not be used. Some SAGE functions for infinite abelian groups are 100% pure Python.

To be concrete, we present some background in order to introduce notation. A finitely generated abelian group is a group $A$ for which there exists a finite presentation defined by an exact sequence

$$\mathbb{Z}^k \to \mathbb{Z}^\ell \to A \to 1,$$

for positive integers $k, \ell$ with $k \leq \ell$.

For example, a finite abelian group has a decomposition

$$A = \langle a_1 \rangle \times \langle a_2 \rangle \times \cdots \times \langle a_\ell \rangle,$$

where $ord(a_i) = p_i^{c_i}$, for some primes $p_i$ and some positive integers $c_i$, $i = 1, 2, \ldots, \ell$. GAP calls the list (ordered by size) of the $p_i^{c_i}$ the *abelian invariants*. In SAGE they will be called *invariants*. In this situation, $k = \ell$ and $\phi : \mathbb{Z}^\ell \to A$ is the map $\phi(x_1, \ldots, x_\ell) = a_1^{x_1} \ldots a_\ell^{x_\ell}$, for $x_i \in \mathbb{Z}$. The matrix of relations $M : \mathbb{Z}^k \to \mathbb{Z}^\ell$ is the matrix whose rows generate the kernel of $\phi$ as a $\mathbb{Z}$-module. In other words, $M = (M_{ij})$ is an $k \times \ell$ diagonal matrix with $M_{ii} = p_i^{c_i}$. Consider now the subgroup $B \subset A$ generated by $b_1 = a_1^{f_{1,1}} \ldots a_\ell^{f_{\ell,1}}$, $\ldots$ , $b_m = a_1^{f_{1,m}} \ldots a_\ell^{f_{\ell,m}}$. The kernel of the map $\phi_B : \mathbb{Z}^m \to B$ defined by $\phi_B(x_1, \ldots, x_m) = b_1^{x_1} \ldots b_m^{x_m}$, for $x_i \in \mathbb{Z}$, is the kernel of the matrix $F = (f_{i,j})$. regarded as a map $\mathbb{Z}^m \to (\mathbb{Z}/p_1^{c_1}\mathbb{Z}) \times \cdots \times (\mathbb{Z}/p_\ell^{c_\ell}\mathbb{Z})$. In particular, $B \cong \mathbb{Z}^m/\ker(F)$. If $B = A$ then the Smith normal form (SNF) of a

generator matrix of $\ker(F)$ and the SNF of $M$ are the same. The diagonal entries $s_i$ of the SNF $S = \mathrm{diag}(s_1, s_2, \ldots, s_r, 0, \ldots, 0)$, are called *determinantal divisors* of $F$, where $r$ is the rank. The *invariant factors* of A are:

$$s_1, s_2/s_1, \ldots, s_r/s_{r-1}.$$

The elementary divisors use the highest (non-trivial) prime powers occuring in the factorizations of the numbers $s_1, s_2, \ldots, s_r$. The definition of elementary divisors of an abelian group used by SAGE is that of Rotman [**?**][2].

SAGE supports multiplicative abelian groups on any prescribed finite number of generators.

Here's a simple example:

SAGE

```
sage: F.<a,b,c,d,e> = AbelianGroup(5, [5,5,7,8,9])
sage: F(1)
1
sage: prod([ a, b, a, c, b, d, c, d ])
a^2*b^2*c^2*d^2
sage: d * b**2 * c**3
b^2*c^3*d
sage: G = AbelianGroup(3,[2,2,2]); G
Multiplicative Abelian Group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names="xy"); H
Multiplicative Abelian Group isomorphic to C2 x C3
sage: AbelianGroup(5)
Multiplicative Abelian Group isomorphic to
  Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

It is self-explanatory, but what is created above is, first, a group $F$ with 5 generators $a, b, c, d, e$, of orders $5, 5, 7, 8, 9$, respectively. The first line above can also be created using the two lines

[2]Since different texts have different definitions of this, this usage may be "non-standard", depending on your background.

SAGE

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()
```

Secondly, a group $G$ with 3 (unnamed) generators of orders $2, 2, 2$. Next, a group $H$ is created having 2 generators $x, y$, of orders $2, 3$. Finally, a free abelian group of rank 5 is created.

**Example 1** Abelian groups arise naturally throughout mathematics. Inheritance in Python means that an object can inherit from the class of abelian groups to achieve functionality as an abelian group, or in this example, construct an isomorphism with a group object.

SAGE

```
sage: k = FiniteField(101)
sage: E = EllipticCurve([k(1),k(3)])
sage: G, gens = E.abelian_group()
sage: G
Multiplicative Abelian Group isomorphic to C87
sage: gens
((32 : 68 : 1),)
```

We note that the second return value is a sequence of group generators which implicitly determines an isomorphism. In order to return instead an isomorphism, it remains to develop a generic framework for morphisms between arbitrary groups. In this instance the abelian group homomorphism must handle the translation between multiplicative and additive notation.

The elements themselves of an abelian group $A$ have some convenient methods implemented. For example, you can determine the permutation which a group element is associated to when you regard $A$ as a permutation group. Also, you can determine the order of an element and read off the powers of the generators occurring in the elements' expression.

Here's a simple example:

SAGE

```
sage: A = AbelianGroup(3,[2,3,4],names="abc"); A
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: a,b,c=A.gens()
sage: (c^3*b).list()
[0, 1, 3]
sage: G = A.permutation_group(); G
Permutation Group with generators \
[(1,13)(2,14)(3,15)(4,16)(5,17)(6,18)(7,19)\
(8,20)(9,21)(10,22)(11,23)(12,24),\
 (1,5,9)(2,6,10)(3,7,11)(4,8,12)(13,17,21)\
(14,18,22)(15,19,23)(16,20,24),\
 (1,3,2,4)(5,7,6,8)(9,11,10,12)(13,15,14,16)\
(17,19,18,20)(21,23,22,24)]
sage: g = a.as_permutation(); g
(1,13)(2,14)(3,15)(4,16)(5,17)(6,18)(7,19)(8,20)\
(9,21)(10,22)(11,23)(12,24)
sage: g in G
True
```

Also implemented are homomorphisms. With them, you can compute images and kernels. Here's an example:

SAGE

```
sage: H = AbelianGroup(3,[2,3,4],names="abc"); H
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: a,b,c = H.gens()
sage: G = AbelianGroup(2,[2,3],names="xy"); G
Multiplicative Abelian Group isomorphic to C2 x C3
sage: x,y = G.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])
sage: phi(y*x)
a*b
sage: phi(y^2)
b^2
sage: phi.parent() == Hom(G,H)
True
sage: phi.parent()
Set of Morphisms from Multiplicative Abelian Group
isomorphic to C2 x C3 to Multiplicative Abelian Group
isomorphic to C2 x C3 x C4 in Category of groups
```

The dual group (the group of complex characters) of a finite abelian group is also implemented. GAP does not have such dual groups functionality, so for this implementation it is not simply a matter of wrapping GAP functions.

Here's an example:

SAGE

```
sage: F = AbelianGroup(5, [2,3,5,7,8], names="abcde")
sage: a,b,c,d,e = F.gens()
sage: Fd = DualAbelianGroup(F, names="ABCDE")
sage: A,B,C,D,E = Fd.gens()
sage: A*B^2*D^7
A*B^2
sage: A(a)    ## random last few digits
-1.0000000000000000 + 0.00000000000000013834419720915037*I
sage: B(b)    ## random last few digits
-0.49999999999999983 + 0.86602540378443871*I
sage: A(a*b)    ## random last few digits
-1.0000000000000000 + 0.00000000000000013834419720915037*I
```

Note that since the field of complex numbers is represented using floating point numbers, inacurracies my enter into the least significant digits. If desired, a higher precision or even a different base ring may be specified (though characteristic 0 is currently assumed).

Similarly a `DirichletCharacter` is the extension of a homomorphism

$$(\mathbb{Z}/N\mathbb{Z})^* \to R^*,$$

for some ring $R$, to the map $\mathbb{Z}/N\mathbb{Z} \to R$ obtained by sending nonunits to 0:

SAGE

```
sage: G = DirichletGroup(35)
sage: G.gens()
([zeta12^3, 1], [1, zeta12^2])
sage: g0,g1 = G.gens()
sage: g = g0*g1; g
[zeta12^3, zeta12^2]
```

```
sage: g.order()
12
```

Dirichlet group elements in SAGE support, among others, special methods for computing Galois orbits, Gauss sums, and generalized Bernoulli numbers

# 3 Permutation groups

A *permutation group* is a finite group G whose elements are permutations of a given finite set X (i.e., bijections $X \to X$) and whose group operation is the composition of permutations. The number of elements of $X$ is called the *degree* of G.

## 3.1 Some permutation group methods

A permutation is inputted into SAGE as either a string that defines a permutation using disjoint cycle notation, or a list of tuples which represent disjoint cycles:

$$\begin{array}{ccc} (a,\ldots,b)(c,\ldots,d)\ldots(e,\ldots,f) & \leftrightarrow & [(a,\ldots,b),(c,\ldots,d),\ldots,(e,\ldots,f)] \\ () = \text{identity} & \leftrightarrow & [()]. \end{array} \tag{1}$$

Warning: There is a SAGE command `Permutation` which does *not* currently make a permutation group element. This command forms part of the combinatorics package, is not to be confused with permutation group constructors.

You can construct the following standard groups as permutation groups. The finite matrix groups over GF($q$) in the list use standard permutation representations to construct them as permuation groups.

- `SymmetricGroup`, $S_n$ of order $n!$, and `AlternatingGroup`, $A_n$ of order $n!/2$ ($n$ can also be a list $X$ of distinct positive integers)
- `DihedralGroup`, $D_n$ of order $2n$, and `CyclicPermutationGroup`, $C_n$ of order $n$
- `TransitiveGroup`, $i^{th}$ transitive group of degree $n$ from the GAP tables of transitive groups (requires the "optional" package `database_gap`)

- `MathieuGroup`, of degrees 9, 10, 11, 12, 21, 22, 23, or 24
- `KleinFourGroup`, subgroup of $S_4$ of order 4 isomorphic to $C_2 \times C_2$
- `PGL(n,q)`, `PSL(n,q)`, `PSp(2n,q)`, projective (general, special, symplectic, resp.) linear group of $n \times n$ matrices over the finite field GF($q$)
- `PSU(n,q)`, projective special unitary group of $n \times n$ matrices having coefficients in the finite field GF($q^2$) that respect a fixed nondegenerate sesquilinear form, of determinant 1.
- `PGU(n,q)`, projective general unitary group of $n \times n$ matrices having coefficients in the finite field GF($q^2$) that respect a fixed nondegenerate sesquilinear form, modulo the centre.
- `Suzuki(q)`, Suzuki group over GF($q$), ${}^2B_2(2^{2k+1}) = Sz(2^{2k+1})$.
- `direct_product_permgroups`, which takes a list of permutation groups and returns their direct product.

Permutation groups include methods for computing composition series, lower and upper central series, multiplication table, character table, quotient group by a normal subgroup, Sylow subgroups, the number of groups of a given order, and many others. The are made available via wrappers for corresponding GAP functions. There are a number of functions which interface with GAP's SmallGroups database, so it is a good idea to have that installed before trying out the examples below[3].

Here are some examples which illustrate various ways to construct permutations in $S_4$:

SAGE

```
sage: G = SymmetricGroup(4)
sage: G((1,2,3,4))
(1,2,3,4)
sage: G([(1,2),(3,4)])
(1,2)(3,4)
sage: G('(1,2)(3,4)')
```

[3]The SmallGroups database is not GPL'd and so must be installed into SAGE separately. See the optional packages page `http://www.sagemath.org/packages.html` for instructions.

```
(1,2)(3,4)
sage: G([1,2,4,3])
(3,4)
sage: G([2,3,4,1])
(1,2,3,4)
sage: G(G((1,2,3,4)))
(1,2,3,4)
sage: G(1)
()
```

The constructor `PermutationGroupElement` creates an element of some the symmetric group $S_n$ for $n$ minimal. Automatic coercion (using $S_n \subset S_{n+m}$ allows the multiplication of elements in different groups, with the result in a minimal group containing both elements. Moreover, such coercion permits equality testing between elements in different permutation groups.

SAGE

```
sage: g1 = PermutationGroupElement([(1,2),(3,4,5)])
sage: g1.parent()
Symmetric group of order 5! as a permutation group
sage: g2 = PermutationGroupElement([(1,2)])
sage: g2.parent()
Symmetric group of order 2! as a permutation group
sage: g1*g2
(3,4,5)
sage: g2*g1
(3,4,5)
sage: g1 == g2
False
```

A permutation group can be constructed in SAGE either by standard constructors or by a given list of generators. SAGE recognises mathematically identical groups as equal even if they are constructed with different sets of generators.

SAGE

```
sage: S5 = SymmetricGroup(5)
```

```
sage: A5 = AlternatingGroup(5)
sage: A5
Alternating group of order 5!/2 as a permutation group
sage: A5.gens()
((1,2,3,4,5), (3,4,5))
sage: A5.is_subgroup(S5)
True
sage: G = PermutationGroup([(1,2,3,4,5),(1,2,3)])
sage: G
Permutation Group with generators [(1,2,3,4,5), (1,2,3)]
sage: G == A5
True
sage: G.group_id()   # requires database_gap* package
[60, 5]
```

Similarly, Galois groups computed with PARI can be imported into SAGE using the GAP small groups database:

SAGE

```
sage: H = pari('x^4 - 2*x^3 - 2*x + 1').polgalois()
sage: G = PariGroup(H, 4); G
PARI group [8, -1, 3, "D(4)"] of degree 4
sage: H = PermutationGroup(G); H  # requires database_gap
Transitive group number 3 of degree 4
sage: H.gens()                    # requires database_gap
((1,2,3,4), (1,3))
```

There is an underlying GAP object that implements each permutation group. One can apply GAP commands (such as `DerivedSeries`) to this GAP object.

SAGE

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: H = gap(G); H
Group([ (1,2,3,4) ])
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: H.DerivedSeries()   # output somewhat random
```

```
[ Group([ (1,2,3)(4,5), (3,4) ]),
  Group([ (1,5)(3,4), (1,5)(2,4), (1,4,5) ]) ]
sage: G.derived_series()
[Permutation Group with generators
 [(1,2,3)(4,5), (3,4)],
 Permutation Group with generators
 [(1,5)(3,4), (1,5)(2,4), (1,3,5)]]
```

Here $G$ is a "SAGE group", to which SAGE's methods (such as `order`) apply. Hit `G.[tab]` to see a list of all the SAGE commands available. On the other hand, $H$ is a "GAP group", to which (any and all of) GAP's group-theoretical methods (such as `Size`) apply. Hit `H.[tab]` to see a list of all the GAP commands available. The command `G.derived_series()` returns the groups as SAGE objects, whereas the output of `H.DerivedSeries()` are GAP objects.

A *composition series* of a group $G$ is a normal series

$$1 = H_0 \subset H_1 \subset \cdots \subset H_n = G,$$

with strict inclusions, such that each $H_i$ is a maximal normal subgroup of $H_{i+1}$. The SAGE method `composition_series` wraps the corresponding GAP function `CompositionSeries`:

SAGE

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.composition_series()  # random output order
[Permutation Group with generators [(1,2,3)(4,5), (3,4)],
 Permutation Group with generators
 [(1,5)(3,4), (1,5)(2,4), (1,3,5)],
 Permutation Group with generators [()]]
```

SAGE has analogous commands `lower_central_series` and `upper_central_series`.

You can also save and reload objects created in a SAGE session:

SAGE

```
sage: G = DihedralGroup(6)
sage: Z = G.center()
sage: save(G, 'G')
sage: save(Z, 'Z')
sage:
Exiting SAGE (CPU time 0m2.08s, Wall time 615m27.05s).
Exiting spawned Gap process.
wdj@wooster:~/wdj/sagefiles/sage-2.9.alpha5$ ./sage
----------------------------------------------------------------------
| SAGE Version 2.9.alpha5, Release Date: 2007-12-10                  |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

sage: G = load('G')
sage: Z = load('Z')
sage: Z.is_subgroup(G)
True
```

SAGE does not remember $Z$ as the center of $G$, but rather that $Z$ and $G$ are given by certain generators as a permutation group. From that, it can determine that $Z$ is a subgroup of $G$. If you use `save_session` and `load_session` instead, the behaviour is similar, except that it automatically saves the variables for you:

SAGE

```
----------------------------------------------------------------------
| SAGE Version 2.9.alpha5, Release Date: 2007-12-10                  |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

sage: G = DihedralGroup(6)
sage: Z = G.center()
sage: Z.is_subgroup(G)
True
sage: save_session('dihedral6')
sage:
Exiting SAGE (CPU time 0m0.23s, Wall time 1m3.16s).
Exiting spawned Gap process.
wdj@wooster:~/wdj/sagefiles/sage-2.9.alpha5$ ./sage
----------------------------------------------------------------------
| SAGE Version 2.9.alpha5, Release Date: 2007-12-10                  |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

sage: load_session('dihedral6')
sage: G; Z
Dihedral group of order 12 as a permutation group
Permutation Group with generators [(1,4)(2,5)(3,6)]
sage: Z.is_subgroup(G)
```

```
True
```

SAGE's `direct_product` wraps GAP's `DirectProduct`, `Embedding`, and `Projection` functions. The direct product of permutation groups will be a permutation group again. `Input`: two permutation groups, $G_1$, $G_2$. `Output`: a 5-tuple $(D, \iota_1, \iota_2, \mathrm{pr}_1, \mathrm{pr}_2)$ - a permutation group and four morphisms.

- $D$ – a direct product of $G_1$, $G_2$, returned as a permutation group;
- $\iota_1$ – an embedding of $G_1$ into $D$;
- $\iota_2$ – an embedding of $G_2$ into $D$;
- $\mathrm{pr}_1$ – the projection of $D$ onto $G_1$ (satisfying $\iota_1 \circ \mathrm{pr}_1 = 1$);
- $\mathrm{pr}_2$ – the projection of $D$ onto $G_2$ (satisfying $\iota_2 \circ \mathrm{pr}_2 = 1$).

Some examples:

SAGE

```
sage: G = CyclicPermutationGroup(4)
sage: D = G.direct_product(G,False)
sage: D
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
sage: D,iota1,iota2,pr1,pr2 = G.direct_product(G)
sage: D; iota1; iota2; pr1; pr2
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
Homomorphism : Cyclic group of order 4 as a permutation group
  --> Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
Homomorphism : Cyclic group of order 4 as a permutation group
  --> Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
Homomorphism : Permutation Group with generators
               [(1,2,3,4), (5,6,7,8)]
  --> Cyclic group of order 4 as a permutation group
Homomorphism : Permutation Group with generators
               [(1,2,3,4), (5,6,7,8)]
  --> Cyclic group of order 4 as a permutation group
sage: g=D([(1,3),(2,4)]); g
(1,3)(2,4)
sage: d=D([(1,4,3,2),(5,7),(6,8)]); d
(1,4,3,2)(5,7)(6,8)
sage: iota1(g); iota2(g); pr1(d); pr2(d)
(1,3)(2,4)
```

```
(5,7)(6,8)
(1,4,3,2)
(1,3)(2,4)
```

SAGE can also display the matrix of values of the irreducible characters of a permutation group $G$ at the conjugacy classes of $G$. The columns represent the conjugacy classes of $G$ and the rows represent the different irreducible characters in the ordering given by GAP.

Some examples[4]:

SAGE

```
sage: # Alternating group A_4 of order 12:
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: G.order()
12
sage: G.character_table()
[   1    1      1           1   ]
[   1    1  -zeta3 - 1    zeta3 ]
[   1    1  zeta3 -zeta3   - 1  ]
[   3   -1      0            0  ]
sage: # Dihedral group D_4 of order 8:
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.order()
8
sage: G.character_table()
[ 1  1  1  1  1]
[ 1 -1 -1  1  1]
[ 1 -1  1 -1  1]
[ 1  1 -1 -1  1]
[ 2  0  0  0 -2]
```

SAGE can compute the Sylow $p$-subgroups of $G$, where $p$ is a prime (this is a $p$-subgroup of $G$ whose index in $G$ is relatively prime to $p$). Here's an example:

[4]Here, `zeta3` refers to a primitive 3-rd root of unity.

SAGE

```
sage: G = AlternatingGroup(5)
sage: G.order()
60
sage: G2 = G.sylow_subgroup(2)
sage: G2.order()
4
sage: G.sylow_subgroup(3)
Permutation Group with generators [(1,2,3)]
sage: G.sylow_subgroup(5)
Permutation Group with generators [(1,2,3,4,5)]
sage: G.sylow_subgroup(7)
Permutation Group with generators [()]
```

> Nobody has ever paid a license fee for the proof that Sylow subgroups exist in every finite group, Nobody should ever pay a license fee for computing Sylow subgroups in a given finite group.
>
> *Joachim Neubüser* [**?**]

SAGE also computes the quotient group $G/N$, where $N$ is a normal subgroup of a permutation group. The method `quotient_group` wraps the GAP operator `/`.

SAGE

```
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: N = PermutationGroup([(1,2,3)])
sage: G.quotient_group(N)
Permutation Group with generators [(1,2)]
```

HAP ("Homology, Algebra, Programming") is a GAP package for group homology and cohomology written by Graham Ellis [**?**]. Thanks to SAGE's wrappers of several HAP functions, homology and cohomology of permutation groups can be computed in SAGE. To compute the homology groups $H^5(S_3, \mathbb{Z})$ and $H^5(S_3, \mathrm{GF}(2))$, use the following commands.

SAGE

```
sage: G = SymmetricGroup(3)
sage: G.cohomology(5)      # needs optional gap_packages
  Trivial Abelian Group
sage: G.cohomology(5,2)    # needs optional gap_packages
  Multiplicative Abelian Group isomorphic to C2
```

The GAP package HAP is not distributed with SAGE, so an additional SAGE package must be loaded for these commands to work. For further details, we refer the interested reader to the expository paper [**?**].

## 3.2 Element methods

Currently, elements of a Permutation group can act on some simple objects, such as strings, lists, and multivariate polynomials. For example,

SAGE

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: g('abcd')
'bcda'
sage: g([0,6,-2,11])
[6, -2, 11, 0]
sage: h = G((1,2))
sage: g*h == h*g
False
sage: (g*h)(1); g(h(1)); h(g(1))
1
3
1
```

Unfortunately, the current notation suggests `g(i)` is a left action, it is not compatible with the current definition of multiplication `(g*h)(i)` is `h(g(i))` rather than `g(h(i))`. Moreover, methods implementing more sophisticated actions are currently lacking (see §**??** below for further discussion).

The Rubik's cube[5] is a puzzle with $9 \times 6 = 56$ facets, but only 48 of them move. (You can think of the 6 center facets as being fixed, since we assume you have fixed an orientation in space of your cube once and for all.) Labeling the facets 1, 2, ..., 48 in any fixed way you like, each move of the Rubik's cube amounts to permuting the symbols in $\{1, 2, \ldots, 48\}$.

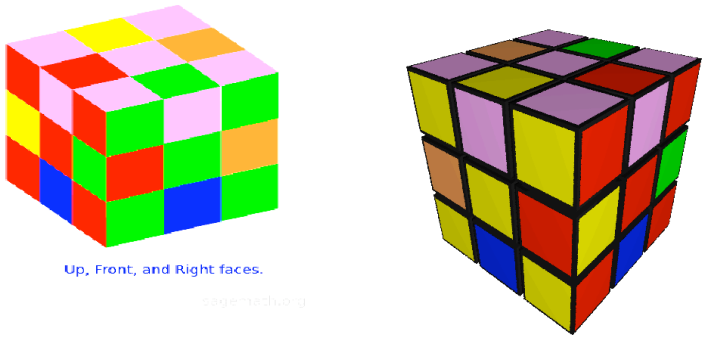


Figure 1: Different SAGE plots of the "superflip" in 3d.

A "basic move" consists of a quarter turn in the clockwise direction of one of the 6 faces. These moves are usually denoted $R$(ight), $L$(eft), $U$(p), $D$(own), $F$(ront), $B$(ack). Each quarter face turn of a Rubik's cube has order 4. Let us compute the order of the Rubik's cube group and the order of a face turn:

SAGE

```
sage: f= [(17,19,24,22),(18,21,23,20),(6,25,43,16),\
(7,28,42,13),(8,30,41,11)]
sage: b=[(33,35,40,38),(34,37,39,36),( 3, 9,46,32),\
( 2,12,47,29),( 1,14,48,27)]
sage: l=[( 9,11,16,14),(10,13,15,12),( 1,17,41,40),\
( 4,20,44,37),( 6,22,46,35)]
sage: r=[(25,27,32,30),(26,29,31,28),( 3,38,43,19),\
( 5,36,45,21),( 8,33,48,24)]
sage: u=[( 1, 3, 8, 6),( 2, 5, 7, 4),( 9,33,25,17),\
(10,34,26,18),(11,35,27,19)]
sage: d=[(41,43,48,46),(42,45,47,44),(14,22,30,38),\
(15,23,31,39),(16,24,32,40)]
```

[5]The reader wishing more background may consult [?] for details.

```
sage: cube = PermutationGroup([f,b,l,r,u,d])
sage: F,B,L,R,U,D = cube.gens()
sage: cube.order()
43252003274489856000
sage: F.order()
4
```

The face turns applied to the cube with the following labeling

Rubik's cube labeling

```
                +--------------+
                |  1    2    3 |
                |  4  top    5 |
                |  6    7    8 |
 +--------------+--------------+--------------+--------------+
 |  9   10   11 | 17   18   19 | 25   26   27 | 33   34   35 |
 | 12  left  13 | 20 front  21 | 28 right  29 | 36  rear  37 |
 | 14   15   16 | 22   23   24 | 30   31   32 | 38   39   40 |
 +--------------+--------------+--------------+--------------+
                | 41   42   43 |
                | 44 bottom 45 |
                | 46   47   48 |
                +--------------+
```

gives rise to an embedding $G \hookrightarrow S_{48}$. It turns out that SAGE already has the Rubik's cube group $G$ "pre-programmed":

SAGE

```
sage: rubik = CubeGroup()
sage: rubik
The PermutationGroup of all legal moves of the Rubik's cube.
```

Next, we shall construct the "superflip" (every edge is flipped, but otherwise the cube is in the solved state) using a known shortest manuever in the face-turn metric.

SAGE

```
sage: rubik = CubeGroup()
```

```
sage: superflip = "R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*\
                   F^3*B^3*D^3*F^2*D^3*R^2*U^3*F^2*D^3"
sage: P = rubik.plot3d_cube(superflip)
sage: show(P)
sage: G = rubik.group()
sage: rubik.move(superflip)[0]
(2,34)(4,10)(5,26)(7,18)(12,37)(13,20)(15,44)(21,28)(23,42)(29,36)(31,45)(39,47)
```

This last line tells us what the superflip move is as an element of our permutation group representation of the Rubik's cube group. Now we shall construct this move group theoretically, using the fact that it is the unique non-trivial element in the center of the Rubik's cube group.

SAGE

```
sage: Z = G.center()
sage: s = Z.gens()[0]
sage: s == rubik.move(superflip)[0]
True
sage: S = RubiksCube(s)
sage: S.solve()
"L2 B2 D2 F L2 R2 B U2 B D2 R2 F' L' R D' B' F R2 D2 R' U' D'"
```

This uses a program written by Dik T. Winter implementing Kociemba's algorithm. Other options are `S.solve("dietz")`, which uses Eric Dietz's cubex program, `S.solve("optimal")`, which uses Michael Reid's optimal program, or `S.solve("gap")`, which uses GAP to solve the "word problem". In fact, Kociemba's algorithm is not bad since it returns a move which is 22 moves in the face-turn metric. (The move given above is only 20 moves.)

## 3.3 Permutation group homomorphisms

SAGE can also compute kernels and images. We shall give a simple example:

SAGE

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: phi = PermutationGroupMorphism_im_gens( G, H, gens, gens)
```

```
sage: phi.image(G)
'Group([ (1,2,3,4) ])'
sage: phi.kernel()
Group(())
sage: phi.image(g)
'(1,2,3,4)'
sage: phi(g)
'(1,2,3,4)'
sage: phi.range()
Dihedral group of order 8 as a permutation group
sage: phi.codomain()
Dihedral group of order 8 as a permutation group
sage: phi.domain()
Cyclic group of order 4 as a permutation group
```

# 4 Matrix groups

Although the projective groups $\mathrm{PSp}_{2n}(K)$ and $\mathrm{PSL}_n(K)$are mathematically the quotients of the matrix groups $\mathrm{Sp}_{2n}(K)$ and $\mathrm{SL}_n(K)$. However, over a finite field, GAP (and, at least currently, SAGE) implements these completely differently. The groups $\mathrm{PSL}(n, \mathrm{GF}(q))$, $\mathrm{PSp}(2n, \mathrm{GF}(q))$, and the other projective classical groups, are realized as *permutation* groups and all the methods in the above section apply[6].

On the other hand, the `MatrixGroup` class is designed for computing in the standard matrix groups $\mathrm{GL}_n$, $\mathrm{SL}_n$, $\mathrm{GO}_n$, $\mathrm{SO}_n$, $\mathrm{GU}_n$ and $\mathrm{SU}_n$, and $\mathrm{Sp}_{2n}$, and their subgroups defined by a (relatively small) finite set of generators.

SAGE

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0, -1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_class_representatives()
[
[1 0]
[0 1],
[0 1]
[2 1],
```

[6]In fact, for some $\mathrm{PSL}(n, \mathrm{GF}(q))$ ($n > 5$ must be a prime), extra methods have been implemented to help with the computation in [**?**], but details would take us too far afield.

```
[0 1]
[2 2],
[0 2]
[1 1],
[0 2]
[1 2],
[0 1]
[2 0],
[2 0]
[0 2]
]
```

## 4.1 General and special linear groups

In SAGE, general linear groups can be constructed over rings other than finite fields. However, at the present time, not many methods are implemented unless the base ring is finite.

SAGE

```
sage: GL(4,QQ)
General Linear Group of degree 4 over Rational Field
sage: GL(1,ZZ)
General Linear Group of degree 1 over Integer Ring
sage: GL(100,RR)
General Linear Group of degree 100 over Real Field
 with 53 bits of precision
sage: GL(3,GF(49,'a'))
General Linear Group of degree 3 over Finite Field
 in a of size 7^2
```

For all the above groups but the last one, SAGE has very few methods implemented so far. However, when the ground field is finite, various methods exist for computing with them.

SAGE

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[1,0]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.order()
```

```
48
sage: H = GL(2,F)
sage: H.order()
48
sage: H == G
True
sage: H.as_matrix_group() == G
True
```

Similarly, the special linear groups have special methods implemented over finite fields:

SAGE

```
sage: FF = GF(3)
sage: for n in range(1,9):
....:     SL(n,FF).order()
....:
1
24
5616
12130560
237783237120
42064805779476480
67034222101339041669120
961721214905722855895197286400
sage: G = SL(2,GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3
sage: G.conjugacy_class_representatives()
[
 [1 0]
 [0 1],
 [0 2]
 [1 1],
 [0 1]
 [2 1],
 [2 0]
 [0 2],
 [0 2]
 [1 2],
 [0 1]
 [2 2],
 [0 2]
 [1 0]
```

```
]
```

In addition, SAGE has implementations of methods for the special linear group over the integers, `SL(2,ZZ)`, and its congruence subgroups $\Gamma_0(N)$, $\Gamma_1(N)$, and $\Gamma(N)$ motivated by their associated spaces of modular forms.

SAGE

```
sage: G = Gamma0(5)
sage: G
Congruence Subgroup Gamma0(5)
```

For the number theoretic methods for congruence subgroups we refer the interested reader to the SAGE reference manual for details.

## 4.2 Orthogonal groups

The general orthogonal group $\mathrm{GO}(e, d, q)$ consists of those $d \times d$ matrices over the field $\mathrm{GF}(q)$ that respect a non-singular quadratic form specified by $e \in \{-1, 0, 1\}$. The value of $e$ must be 0 for odd $d$ (and can optionally be omitted in this case), respectively one of 1 or $-1$ for even $d$.

`SpecialOrthogonalGroup` returns a group isomorphic to the special orthogonal group $\mathrm{SO}(e, d, q)$, which is the subgroup of all those matrices in the general orthogonal group that have determinant one. (The index of $\mathrm{SO}(e, d, q)$ in $\mathrm{GO}(e, d, q)$ is 2 if $q$ is odd, but $\mathrm{SO}(e, d, q) = \mathrm{GO}(e, d, q)$ if $q$ is even.)

*Warning*: GAP's notation for the finite orthogonal groups differs from that of SAGE. (This is forced by the Python constraint that optional arguments must follow all required arguments.) Whereas GAP uses the notation `GO([e,]d,q)` and `SO([e,]d,q)` where [...] denotes an optional value, SAGE uses the notation `GO(d,GF(q),e=0)` and `SO(d,GF(q),e=0)`.

SAGE

```
sage: G = SO(3,GF(5))
sage: G.gens()
[
[2 0 0]
```

```
[0 3 0]
[0 0 1],
[3 2 3]
[0 2 0]
[0 3 1],
[1 4 4]
[4 0 0]
[2 0 4]
]
sage: G = SO(3,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 3 generators:
[[[2, 0, 0]
sage: GO( 3, GF(7), 0)
General Orthogonal Group of degree 3, form parameter 0,
 over the Finite Field of size 7
sage: GO( 3, GF(7), 0).order()
672
```

Special classes also exist for the general and special unitary groups $\mathrm{GU}_n$ and $\mathrm{SU}_n$ and the symplectic groups $\mathrm{Sp}_{2n}$. Similarly for the orthogonal groups, there exist special methods for these groups over finite fields.

As with homomorphisms between permutation groups, SAGE can compute with homomorphisms between matrix groups over finite fields. The `hom` code wraps GAP's `GroupHomomorphismByImages` function but only for matrix groups.

SAGE

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS([1,1,0,1])])
sage: H = MatrixGroup([MS([1,0,1,1])])
sage: phi = G.hom(H.gens())
sage: phi
Homomorphism : Matrix group over Finite Field
 of size 5 with 1 generators:
 [[[1, 1], [0, 1]]] --> Matrix group over Finite Field
 of size 5 with 1 generators:
 [[[1, 0], [1, 1]]]
sage: phi(MS([1,1,0,1]))
[1 0]
[1 1]
sage: F = GF(7); MS = MatrixSpace(F,2,2)
```

```
xsage: F.multiplicative_generator()
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
```

# 5 Future developments

The main advantages of SAGE over a stand-alone package like GAP (from which SAGE draws much of its group-theoretic functionality) is the combination of

1. a standard, well-supported, and modern programming language Python with user-friendly interfaces;

2. a comprehensive libary of the best open source algorithms covering most all areas of mathematics.

3. a well thought out categorical design of classes of objects and their elements together with the sets of morphisms $\mathrm{Hom}(X, Y)$ and their element homomorphisms $\phi : X \to Y$;

## 5.1 Categories and morphisms

The categorical framework lets one create objects $X$ and homsets $\mathrm{Hom}(X, Y)$ in SAGE and manipulate them. This framework needs to be extended to include more functionality on the homsets and their subsets of $\mathrm{Iso}(X, Y)$ or $\mathrm{Aut}(X)$. Additional work is needed on efficient integrity checking to ensure, for given groups $G$ and $H$ that a map created in $\mathrm{Hom}(G, H)$ specifies a valid homomorphism. Constructors for homomorphisms between the different classes of abelian groups, permutation groups, and matrix groups will aid in translating between classes on which different methods may apply or be more efficient.

## 5.2 Efficiency of algorithms

In order to optimize performance, more of the basic arithmetic and needs to be coded in Cython/Pyrex [**?**], so that calls to GAP are necessary only for

sophisticated high-level algorithms. We envision that more SAGE/Python code will emerge as native applications are developed to make use of the diverse range of SAGE code for rings, algebras, and combinatorics, or to develop applications to to number theory and algebraic geometry.

## 5.3 Abelian groups

Currently, abelian groups are implemented with a multiplicative notation providing consistency of group operations * and ^ (as opposed to + and *) within the category of groups. An additive interface to abelian groups is envisioned to provide an alternative representation, and a class from which many naturally occurring additive groups could inherit (like the group of points on an elliptic curve of Example **??** or the additive group of a finite ring). A metastructure for groups would handle translation between the additive and multiplicative notation.

Moreover, it would be desirable to support more (potentially) infinite abelian groups like unit groups of rings like $\mathbb{Q}^\times$, $\mathbb{Z}_{(p)}^\times$, $\mathbb{Z}_p^\times$, $\mathbb{Q}_p^\times$, $\mathbb{R}^\times$ and $\mathbb{C}^\times$ (unit groups of the rationals, the valuation ring $\mathbb{Z}_{(p)} = \{n/m \mid (m,p) = 1\}$, the $p$-adics, reals and complexes) and additive groups of $GF(q)$, $\mathbb{Q}$, and $\mathbb{Z}$, and permitting the construction quotients $\mathbb{Q}/\mathbb{Z}$ and of dual groups $\mathrm{Hom}(A, \mathbb{Q}^\times)$, $\mathrm{Hom}(A, \mathbb{Z}_p^\times)$, $\mathrm{Hom}(A, \mathbb{C}^\times)$, and $\mathrm{Hom}(A, \mathbb{Q}/zzz)$. There are an infinity of possible structures, but these groups arise naturally in mathematics, and the framework for category theory gives a clean interface in which to define them.

## 5.4 Permutation groups

Permutation groups arise naturally in the representation of group actions on finite sets. We envision enhancing and extending the permutation group class to allow left or right actions on arbitrary finite sets, with additional Python classes in SAGE for the $G$-sets on which they act. This would constitute a new category in SAGE, including $G$-set morphisms, consisting of a homomorphism $G \to H$ with compatible map of sets.

Implementing permutations groups which act naturally (on the right or left) on more general sets than $\{1, 2, \ldots, n\}$ would simplify actions on the index sets $\{0, 1, \ldots, n-1\}$. In terms of efficiency, one would like to do something like:

SAGE?

```
sage: G = SymmetricGroup(n,0,n-1)
sage: for i in range(n):
....:     g = G.random_element()
....:     x[g(i)]
```

Obviously, for large $n$, one does not want to create the entire list `g([ i for i in range(n) ])` in order to extract one element.

The example of the Rubik's cube is less trivial example of a $G$-set. The Rubik's code group acts naturally on the set $S$ of states of the Rubik's cube. Moreover, this group action $G \times S \to S$ is capatible with the map $\pi_1 : S \to C$ to the corner states $C$, and $\phi_2 : S \to E$ to the set of edge states $E$, capatible in the sense that the natural actions

$$G \times C \to C \text{ and } G \times E \to E,$$

satisfy $\phi_i(g(x)) = g(\phi_i(x)))$. A category of $G$-sets would give a natural framework for investigating the kernel of the actions of $G$ on $C$ and $E$ (or on equivalence classes of corners and edges), giving a mathematical structure to the classes of reduction moves.

In the mathematical "real world", a category of $G$-sets would provide a natural framework for investigating Galois actions on sets (of class groups, torsion points on abelian varieties, or a set of special points on curves or algebraic varieties). The infrastructure for $G$-sets would give a mechanism for representing the action directly on a finite set, and handle the induced maps under morphisms applied to the underlying sets.

## 5.5 Matrix groups

Matrix groups are the primary objects of study of representation theory, and arise naturally in geometric group actions (e.g. on a vector space equipped with the structure of a bilinear form). Permutation group representations give rise a powerful algorithms for studying finite groups but linear representations provide other tools. Standard packages for group theory – GAP and Magma – given preference to the permutation group representation for $PSL(n, GF(q))$, ignoring the geometric structure encoded in the matrix representation.

A future implementation in SAGE could differentiate the permutation group and matrix group representations as follows:

SAGE?

```
sage: PSL(2,7)
Permutation Group with generators [(3,7,5)(4,8,6),
 (1,2,6)(3,4,8)]
sage: G = PSL(2,GF(7))
sage: G
Projective special linear group of degree 2 over
Finite field of size 7
sage: G([1,1,0,1])
[1 1]
[0 1]
```

Over a finite field, `PSL(n,R)` would have augmented features, like the ability to determine the order and generators (based on the permutation representation), while explicit homomorphisms could translate between the permutation representation when $R$ is a finite field.

SAGE?

```
sage: G = PSL(2, GF(7))
sage: H = G.permutation_group()
sage: H
Permutation Group with generators [(3,7,5)(4,8,6),
 (1,2,6)(3,4,8)]
sage: f = G.permutation_representation()
sage: f
Homomorphism from Permutation Group ... to Projective
special linear group of degree 2 over Finite field of size 7
```

These matrix repsentations also provide a tool for analysis of infinite matrix groups. For instance, a coset of $\Gamma(p)$ in $\mathrm{SL}_2(\mathbb{Z})$, has a representative in $\mathrm{SL}_2(\mathrm{GF}(p))$ using the natual exact sequence:

$$1 \to \Gamma(p) \to \mathrm{SL}_(\mathbb{Z}) \to \mathrm{SL}_2(\mathrm{GF}(p)) \to 1$$

One could then pass to an element of $\mathrm{PSL}_2(\mathrm{GF}(p))$, via the exact sequence of matrix groups:

$$1 \to \mathrm{GF}(p)^* \to \mathrm{SL}_2(\mathrm{GF}(p)) \to \mathrm{PSL}_2(\mathrm{GF}(p)) \to 1$$

and use homomorphisms to access the efficient permutation group algorithms for PSL$(2, p)$. The role of a computer algebra system like SAGE is to facilitate such translation of a mathematical structures.

## 5.6 Other group families

Besides increased functionality in matrix groups and permutation groups, a wider class of groups, and their methods, remain to be implemented. These include Lie groups and Lie algebras, Coxeter groups, $p$-groups, crystallographic groups, polycyclic groups, solvable groups, free groups and braid groups. At the time of this writing, SAGE's functionality with these groups has room for a great deal of improvement (to put it euphemistically). However, there exist a wide range of open source programs (particularly in GAP) for dealing with such groups.

To illustrate how straightforward adding this functionality to SAGE really is (anyone with time and a basic understanding of group theory and Python/SAGE can do this), we give an example of a simple wrapper. Wrapping a GAP function in SAGE is a matter of writing a program in Python which uses the pexpect interface to pipe various commands to GAP and read back the input into SAGE.

For example[7], suppose we want to make a wrapper for computation of the Cartan matrix of a simple Lie algebra. The Cartan matrix of $G_2$ is available in GAP using the commands

GAP

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> R:= RootSystem( L );
<root system of rank 2>
gap> CartanMatrix( R );
[ [ 2, -1 ], [ -3, 2 ] ]
```

(Incidentally, most of the GAP Lie algebra implementation was written by Thomas Breuer, Willem de Graaf and Craig Struble.) In SAGE, one can

[7]We emphasize that this is just an example to illustrating wrappers. In fact, the module `sage/combinat/cartan_matrix.py`, written by Mike Hansen, implements the method `cartan_matrix` directly in Python, without calling GAP, so this particular wrapper is not needed.

simply type

GAP

```
sage: L = gap.SimpleLieAlgebra('"G"', 2, 'Rationals'); L
<Lie algebra of dimension 14 over Rationals>
sage: R = L.RootSystem(); R
<root system of rank 2>
sage: R.CartanMatrix()
[ [ 2, -1 ], [ -3, 2 ] ]
```

Note the `'"G"'` which is evaluated in Gap as the string `"G"`.

Using this example, we show how one might write a Pyhon/SAGE program who's input is, say, `('G',2)` and who's output is the matrix above (but as a SAGE matrix).

First, the input must be converted into strings consisting of legal GAP commands. Then the GAP output, which is also a string, must be parsed and converted if possible to a corresponding SAGE/Python class object.

Python

```
def cartan_matrix(type, rank):
    """
    Return the Cartain matrix of given Chevalley type and rank.

    INPUT:
        type -- a Chevalley letter name, as a string, for
                a family type of simple Lie algebras
        rank -- an integer (legal for that type).

    EXAMPLES:
        sage: cartan_matrix("A",5)
        [ 2 -1  0  0  0]
        [-1  2 -1  0  0]
        [ 0 -1  2 -1  0]
        [ 0  0 -1  2 -1]
        [ 0  0  0 -1  2]
        sage: cartan_matrix("G",2)
        [ 2 -1]
        [-3  2]
    """

    L = gap.SimpleLieAlgebra('"%s"'%type, rank, 'Rationals')
    R = L.RootSystem()
```

```
    sM  = R.CartanMatrix()
    ans = eval(str(sM))
    MS  = MatrixSpace(ZZ, rank)
    return MS(ans)
```

The output `ans` is a Python list. The last two lines convert that list to a SAGE class object Matrix instance.

Anyone interesting in contributing code to SAGE, please subscribe to the `sage-devel` list at `http://www.sagemath.org/lists.html` and post your idea. All contributions, are welcome!

## 5.7 Open source groups database

On the internet, you can find various online databases of mathematical objects:

- Sloane's online database of integer sequences, `http://www.research.att.com/~njas/sequences/`,
- Linear error-correcting codes, `http://www.codetables.de` and `http://www.win.tue.nl/%7Eaeb/voorlincod.html`,
- Atlas character tables for certain group families, `http://brauer.maths.qmul.ac.uk/Atlas/v3/`,

and so on. However, to our knowledge, no such an online database exists for finite groups. We envision a peer-review system, much in the same way that Sloane has set up for the OEIS, which allows anyone to contribute valid group-theoretic data to the database. It would also be available for download in a suitable format, and licensed in such a way that the downloaded data could be redistributed. Furthermore, we envision a certificate system that SAGE can issue which can help with the following scenario. Suppose 10 different people want to contribute methematical data to this peer-reviewed open source database. Suppose each of these contributions were obtained from an hour of computer calculation using SAGE. It would be useful for the referees if SAGE could implement a "trusted certificate of computation" which verified that a specific computation was actually performed (on a specific computer which a specific version of SAGE at a specific time).

*Acknowledgements*: We thank Mike Hansen, Arturo Magidin, and the referee for their careful reading and many helpful suggestions.

# References

[C1] H. Cohen, **Advanced topics in computational number theory**, Springer, 2000.

[C2] H. Cohen, **A course in computational algebraic number theory**, Springer, 1996.

[Cy] Cython, `http://www.cython.org/`.

[H] G. Ellis, The `GAP` package HAP 1.8.4, available from `http://www.gap-system.org/Packages/hap.html`.

[J1] D. Joyner, *A primer on computational group homology and cohomology*, to appear in **Aspects of Infinite Groups**, (ed. Ben Fine), World Scientific. Available at `http://arxiv.org/abs/0706.0549`

[J2] D. Joyner, **Adventures with group theory: Rubik's cube, Merlin's machine, and other mathematical toys**, 2nd edition, The Johns Hopkins Univ. Press, 2008.

[JK] D. Joyner and A. Ksir, *Modular representations on some Riemann-Roch spaces of modular curves X(N)*, in **Computational Aspects of Algebraic Curves**, (Editor: T. Shaska) Lecture Notes in Computing, WorldScientific, 2005. Available at `http://front.math.ucdavis.edu/math.AG/0502586`

[M] Magma, `http://magma.maths.usyd.edu.au/magma/`.

[Ma] Mathematica, `http://www.wolfram.com/`.

[N] J. Neubüser, "An invitation to Computational Group Theory," in **Groups St Andrews, Galway, 1993**, (ed. C. M. Campbell), Cambridge University Press, 1995. Available at: `http://www.gap-system.org/Doc/Talks/talks.html`.

[P] Pexpect, `http://pexpect.sourceforge.net/`.

[R] J. Rotman, **An introduction to the theory of groups**, 4th ed, Springer, 1995.

[S] W. Stein, SAGE *Mathematics Software (Version 2.10)*, The SAGE Group, 2008, `http://www.sagemath.org`.

[St] W. Stein, **Modular Forms: A Computational Approach**, with an appendix by Paul Gunnells, AMS Graduate Studies in Mathematics, Vol. 79, 2007.

David Joyner
Mathematics Department
U. S. Naval Academy
Annapolis, MD 21402, USA
wdj@usna.edu

David Kohel
Institut de Mathématiques de Luminy
163, avenue de Luminy, case 907
13288 Marseille cedex 9 FRANCE
kohel@iml.univ-mrs.fr