

Group theory in **SAGE**

David Joyner and David Kohel

1-24-2008

Abstract

In rough design, **SAGE** is implemented using a 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 [S] is a general purpose computer algebra system started in 2005 which includes (among many other packages) 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 [M], Mathematica [Ma] 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 a categorical framework, with methods for objects, methods for their elements, and methods for their morphisms. For instance, a permutation group G would be an object. It has (for example) a method called `order`, which computes $|G|$. This particular method, and many others, are collected into a permutation group “(Python-)module”. A permutation $g \in G$ has (for example) a method called `order`, which computes the smallest $n > 0$ for which $g^n = 1$. This method, and many others, are collected into a permutation group element “module”. Finally, constructions of homomorphisms between permutation groups, and any methods which apply to them (for example, `kernel`), are collected in a permutation group morphism “module”. You can find these well documented in the **SAGE** source code.

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

1 The GAP interface

For the most part, **SAGE**'s group-theoretic ability is derived from the extensive functionality of the computer algebra system GAP. For the most part, **SAGE** communicates with GAP and the other components using pseudotty's and a Python package called `pexpect` [P]. 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 example¹,

¹Note 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.

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 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.

Of course, a functional understanding of `pseudotty`'s and `pexpect` is not needed to use **SAGE**. However, it might help to understand why **SAGE** commands work as they do.

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 no noticeable 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** passes the computation to GAP and returns the result via `pexpect`.

2 Abelian groups

Both finite and infinite (but finite rank) abelian groups are supported, but currently only with a multiplicative notation. 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, here is some background, which also serves to introduce notation. An *abelian group* (of finite rank) is a group A for which there exists an exact sequence $\mathbb{Z}^k \rightarrow \mathbb{Z}^\ell \rightarrow A \rightarrow 1$, for some positive integers k, ℓ 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 $\text{ord}(a_i) = p_i^{c_i}$, for some primes p_i and some positive integers c_i , $i = 1, 2, \dots, \ell$. 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 $\phi : \mathbb{Z}^\ell \rightarrow A$ is the map $\phi(x_1, \dots, x_\ell) = a_1^{x_1} \dots a_\ell^{x_\ell}$, for $x_i \in \mathbb{Z}$. The matrix of relations $M : \mathbb{Z}^k \rightarrow \mathbb{Z}^\ell$ is the matrix whose rows generate the kernel of ϕ as a \mathbb{Z} -module. In other words, $M = (M_{ij})$ is an $\ell \times \ell$ diagonal matrix with $M_{ii} = p_i^{c_i}$. Consider now the subgroup $B \subset A$ generated by $b_1 = a_1^{f_{1,1}} \dots a_\ell^{f_{\ell,1}}$, \dots , $b_m = a_1^{f_{1,m}} \dots a_\ell^{f_{\ell,m}}$. The kernel of the map $\phi_B : \mathbb{Z}^m \rightarrow B$ defined by $\phi_B(x_1, \dots, x_m) = b_1^{x_1} \dots b_m^{x_m}$, for $x_i \in \mathbb{Z}$, is the kernel of the matrix $F = (f_{i,j})$ regarded as a map $\mathbb{Z}^m \rightarrow (\mathbb{Z}/p_1^{c_1}\mathbb{Z}) \times \cdots \times (\mathbb{Z}/p_\ell^{c_\ell}\mathbb{Z})$. In particular, $B \cong \mathbb{Z}^m / \ker(F)$. If $B = A$ then the Smith normal form (SNF) of a generator matrix of and the SNF of are the same. The diagonal entries s_i of the SNF $S = \text{diag}(s_1, s_2, \dots, s_r, 0, \dots, 0)$, are called *determinantal divisors* of F . where r is the rank. The *invariant factors* of A are:

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

The elementary divisors use the highest (non-trivial) prime powers occurring in the factorizations of the numbers s_1, s_2, \dots, s_r . The definition of elementary divisors of an abelian group which **SAGE** uses is that of Rotman [R].

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: mul([ a, b, a, c, b, d, c, d ], F(1))
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, resp.. The first line above can also be created using the two lines

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 in the theory of modular abelian varieties [St]. **SAGE** has good support for this. Here is an example of ways in which abelian groups enter:

SAGE

```

sage: J = J0(50)
sage: T = J.torsion_subgroup(); T
Torsion subgroup of Jacobian of the modular curve
associated to the congruence subgroup Gamma0(50)
sage: T.multiple_of_order()
15
sage: T.divisor_of_order()
15
sage: T.gens()
[[1/15, 3/5, -3/5, -1/15]]
sage: T.invariants()
[15]
sage: d[0].torsion_subgroup().order()
3
sage: d[1].torsion_subgroup().order()
5

```

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: G = AbelianGroup(3,[2,3,4],names="abc"); G
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: a,b,c=G.gens()
sage: (c^3*b).list()
[0, 1, 3]
sage: Gp = G.permutation_group(); Gp
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: ap = a.as_permutation(); ap
(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: ap in Gp
True
```

Similarly, if you have an element in A and you want to write as a product of other elements (i.e., you want to solve a “word problem”) then you can use **SAGE** for that as well. This only works for finite groups A and essentially, this function is a wrapper for the GAP functions `EpimorphismFromFreeGroup` and `PreImagesRepresentative`. Here is an example:

SAGE

```
sage: A = AbelianGroup(5,[3, 5, 5, 7, 8], names="abcde")
sage: a,b,c,d,e = A.gens()
sage: b1 = a^3*b*c*d^2*e^5
sage: b2 = a^2*b*c^2*d^3*e^3
sage: b3 = a^7*b^3*c^5*d^4*e^4
sage: b4 = a^3*b^2*c^2*d^3*e^5
sage: b5 = a^2*b^4*c^2*d^4*e^5
sage: e.word_problem([b1,b2,b3,b4,b5],display=False)
[[b^2*c^2*d^3*e^5, 245]]
sage: (b^2*c^2*d^3*e^5)^245
e
```

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.kernel()
'Group([  ])'

```

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
sage: G = AbelianGroup(5,[3, 5, 5, 7, 8],names="abcde")
sage: Gd = DualAbelianGroup(G,names="abcde")
sage: a,b,c,d,e = Gd.gens()
sage: u = a^3*b*c*d^2*e^5
sage: v = a^2*b*c^2*d^3*e^3
sage: w = a^7*b^3*c^5*d^4*e^4
sage: x = a^3*b^2*c^2*d^3*e^5
sage: y = a^2*b^4*c^2*d^4*e^5
sage: e.word_problem([u,v,w,x,y],display=False)
[[b^2*c^2*d^3*e^5, 245]]

```

Note that since the field of complex numbers is represented using floating point numbers, the output of the least significant digits is somewhat random.

You can specify more precision (or even a different base ring, though the current implementation assumes the characteristic is 0), if desired.

A `DirichletCharacter` is the extension of a homomorphism

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

for some ring R , to the map $\mathbb{Z}/N\mathbb{Z} \rightarrow R$ obtained by sending those $x \in \mathbb{Z}/N\mathbb{Z}$ with $\gcd(N, x) > 1$ to 0. Here's an example:

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
```

For example, you can compute Gauss sums and Galois orbits of such characters in **SAGE**.

3 Permutation groups

A *permutation group* is a finite group G whose elements are permutations of a given finite set X (i.e., bijections $X \rightarrow X$) 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

Also included are methods such as 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. These are, for the most part, 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 below².

Here are some examples.

We illustrate each way to make a permutation 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)')
(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)
()
```

Implicit coercion:

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)])
```

²The 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.

```

sage: g2.parent()
Symmetric group of order 2! as a permutation group
sage: g1*g2
(3,4,5)
sage: g2*g2
()
sage: g2*g1
(3,4,5)

```

The usual **SAGE** construction of a permutation group (which happens in this case to be the simple group A_5):

```

----- SAGE -----
sage: G = PermutationGroup([[ (1,2,3,4,5) ], [ (3,4,5) ]])
sage: G
Permutation Group with generators [ (1,2,3,4,5), (3,4,5) ]
sage: G.group_id() # requires database_gap* package
[60, 5]
sage: G.isomorphism_type_info_simple_group()

rec(
  series := "A",
  parameter := 5,
  name :=
    "A(5) ~ A(1,4) = L(2,4) ~ B(1,4) = O(3,4) ~ C(1,4)\
    = S(2,4) ~ 2A(1,4) = U(2,4) ~ A(1,5) = L(2,5) ~ B(1,5)\
    = O(3,5) ~ C(1,5) = S(2,5) ~ 2A(1,5) = U(2,5)"
)

```

You can also make permutation groups from PARI groups:

```

----- 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. 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 **GAP**’s `CompositionSeries`:

SAGE

```
sage: G = PermutationGroup([[ (1,2,3), (4,5) ], [ (3,4) ]])
```

```
sage: G.composition_series() # somewhat random output
[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 $(D, \iota_1, \iota_2, \text{pr}_1, \text{pr}_2)$ - a permutation group and four morphisms.

- D – a direct product of G_1, G_2 , returned as a permutation group;
- ι_1 – an embedding of G_1 into D ;
- ι_2 – an embedding of G_2 into D ;
- pr_1 – the projection of D onto G_1 (satisfying $\iota_1 \circ \text{pr}_1 = 1$);
- pr_2 – the projection of D onto G_2 (satisfying $\iota_2 \circ \text{pr}_2 = 1$).

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 examples³:

```

----- SAGE -----
sage: G = PermutationGroup([(1,2),(3,4)], [(1,2,3)]) ## A_4
sage: G.order()
12
sage: G.character_table()
[ 1  1  1  1  1 ]
[ 1  1 -zeta3 - 1  zeta3 ]
[ 1  1  zeta3 -zeta3  - 1 ]
[ 3 -1  0  0 ]
sage: G = PermutationGroup([(1,2),(3,4)], [(1,2,3,4)]) ## D_8
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 ]

```

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

```
[ 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:

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 [N]

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 [H]. Thanks to **SAGE**’s wrappers of several HAP functions, homology and cohomology of permutation groups can be computed in **SAGE**. To compute $H^5(S_3, \mathbb{Z})$ and $H^5(S_3, GF(2))$, 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 [J1].

3.2 Element methods

The Rubik’s cube⁴ 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.) Labeling the facets 1, 2, \dots , 48 in any fixed way you like, each move of the Rubik’s cube amounts to permuting the symbols in $\{1, 2, \dots, 48\}$.

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)]
```

⁴The reader wishing more background may consult [J2] for details.

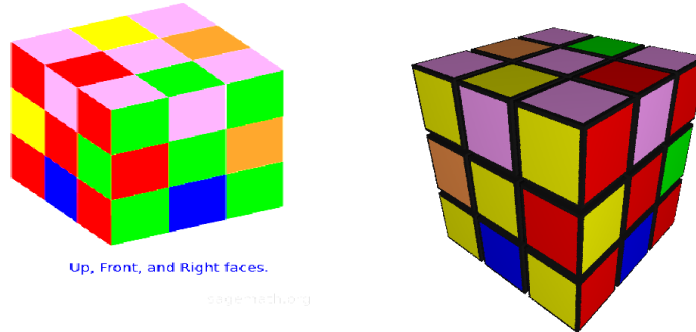


Figure 1: SAGE plots of the “superflip” in 3d.

```
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)]
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 maneuver 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

You might think of projective symplectic groups $PSp(2n)$ as belonging to the same family of matrix groups as the symplectic groups $Sp(2n)$. However, over a finite field, GAP (and, at least currently, **SAGE**) implements these completely differently. The groups $PSL(n, GF(q))$, $PSp(2n, GF(q))$, and the other projective classical groups, are realized as *permutation* groups and all the methods in the above section apply⁵.

The `MatrixGroup` class is designed for computing with matrix groups defined by a relatively (small) finite set of generating matrices.

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],
[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.

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

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, for the special linear groups:

SAGE

```
sage: G = SL(2,GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3
sage: G.is_finite()
True
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]
]

```

The special linear group over the integers, $SL(2, \mathbb{Z})$, brings to mind the congruence subgroups, such as $\Gamma_0(N)$, which are also implemented in **SAGE**.

SAGE

```

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

```

Since most of the methods implemented for congruence subgroups are number-theoretical, and a bit off-topic, we refer the interested reader to the **SAGE** reference manual for details.

4.2 Orthogonal groups

The general orthogonal group $GO(e, d, q)$ consists of those $d \times d$ matrices over the field $GF(q)$ 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 $SO(e, d, q)$, which is the subgroup of all those matrices in the general orthogonal group that have determinant one. (The index of $SO(e, d, q)$ in $GO(e, d, q)$ is 2 if q is odd, but $SO(e, d, q) = GO(e, d, q)$ if q is even.)

Warning: GAP's notation for the finite orthogonal groups differs from **SAGE**'s notation. (This is forced on us, because of the way Python wants optional arguments ordered after the required arguments.) Whereas GAP uses the notation $\text{GO}([e,] d, q)$, $\text{SO}([e,] d, q)$ ([...] denotes an optional value), **SAGE** uses the notation $\text{GO}(d, \text{GF}(q), e=0)$, $\text{SO}(d, \text{GF}(q), e=0)$. Hopefully this will cause no confusion.

```

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

```

Also implemented are symplectic groups and unitary groups over a finite field. These are similar and omitted.

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)
sage: 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

In general, the plans are to rewrite more of the basic operations in terms of Cython/Pyrex [Cy].

The only class of abelian groups which have been implemented at the time of this writing are those with a multiplicative structure. There are plans to implement abelian groups with additive structure as well. Though it was not a group from the `AbelianGroup` or `PermutationGroup` or `MatrixGroup` class, we saw in Example 1 a group given by the torsion points of a Jacobian. It should be possible to “coerce” this into an instance of `AbelianGroup`. Quotients of abelian groups are not yet implemented, Moreover, if R is a commutative ring with unit and $a \in R$ is an element such that R/aR is finite, then, at the time of this writing, **SAGE** cannot “coerce” this quotient into a finite abelian group with generators as described above. These are needed future developments.

Here we explain the relevance of (1) permutation groups and (2) matrix groups as arising from (1) group actions on [finite] sets (G -sets) and (2) actions on vector spaces (G -modules), quadratic spaces (vectors spaces with given inner product), or projective quotients. To do this properly, we envision creating additional Python classes in **SAGE**. In particular there should

be functionality for homomorphisms between G -sets or G -modules, which doesn't exist at the time of this writing.

5.1 Matrix groups

In the domain of matrix groups, it would be useful to have constructors for PSL (and other projective groups) both as permutation groups and as a quotient of matrix groups.

In Magma one has:

```
----- MAGMA -----
> PSL(2,7);
Permutation group acting on a set of cardinality 8
Order = 168 = 2^3 * 3 * 7
  (3, 6, 7)(4, 5, 8)
  (1, 6, 2)(3, 8, 7)
> PSL(2,GF(7));
Permutation group acting on a set of cardinality 8
Order = 168 = 2^3 * 3 * 7
  (3, 6, 7)(4, 5, 8)
  (1, 6, 2)(3, 8, 7)
```

In SAGE:

```
----- SAGE -----
sage: PSL(2,7)
Permutation Group with generators [(3,7,5)(4,8,6),
(1,2,6)(3,4,8)]
```

A future implementation could differentiate this permutation group from:

```
----- SAGE? -----
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]
```

Or maybe the printing would just be something compact like $\text{PSL}(2, \text{GF}(7))$.

Over a finite field R , $\text{PSL}(n, R)$ would have augmented features, like the ability to determine the order and generators (based on the permutation representation). In fact, we should try to implement the following behaviour:

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
```

There should also be capabilities of creating the group $\text{PSL}(2, \mathbb{Z})$ and then to compute and work with the mod p maps $\text{PSL}(2, \mathbb{Z}) \rightarrow \text{PSL}(2, \text{GF}(p))$ for various primes.

5.2 Permutation groups

The example of the Rubik's cube is a good example of a G -set. Let G be the Rubik's code group and S the set of states of Rubik's cube. Then one has a group action $G \times S \rightarrow S$. So G is a group which acts on the G -set S . If one takes the sets C of corner states and E of edge states, then C and E are G -sets with G -set ("forgetful") morphisms $\phi_1 : S \rightarrow C$ and $\phi_2 : S \rightarrow E$ (such that $\phi_i(g(x)) = g(\phi_i(x))$). These states of the Rubik's cube, along with the edge states and corner states, should be implemented in some future version fo **SAGE** as instances of G -set data structures.

For a "real world" example: Consider a projective curve X with Weierstrass points $W(X)$. Since the automorphism group G of X must act on $W(X)$ as a permutation group, this too is an example of a G -set. Indeed, one can try to compute G by looking at the possible induced permutations on $W(X)$.

5.3 Other group families

Besides increased functionality in matrix groups and permutation groups, a wider class of groups, and their methods, must 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).

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, 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 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 Python/**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.

Alternatively, one could code the body of the above function more “pythonically” as follows:

```

Python

```

```
L = gap.new(SimpleLieAlgebra("%s",%s, Rationals);'%(type, rank))
R = L.RootSystem()
sM = R.CartanMatrix()
return sM._matrix_(QQ)
```

If you are 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, especially those with a GPL-compatible license, are welcome!

5.4 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 mathematical 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] —, **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] —, **Adventures with group theory: Rubik’s cube, Merlin’s machine, and other mathematical toys**, 2nd edition, The Johns Hopkins Univ. Press, 2008.
- [JK] — 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] William Stein, **SAGE Mathematics Software (Version 2.10)**, The SAGE Group, 2008, <http://www.sagemath.org>.
- [St] William 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
Université de la Méditerranée
AIX-Marseille II FRANCE
kohel@iml.univ-mrs.fr