Google Summer of Code 2023 - Parametrizing Invariant Spaces

Bruno Edwards

Summer 2023

Contents

Part I
Introduction

This project is titled *Parametrizing Invariant Spaces*, and generally concerns mathematical results from the field of invariant theory.

Invariant theory, at its core, usually addresses questions about how different mathematical transformations are able to leave certain objects (typically polynomials) the same (aka **invariant**).

In this project, we delve into and refine the SageMath implementations of numerous results in invariant theory. Specifically, we investigate infinite sets of polynomials that can be parametrized and are invariant under the action of finite groups when these groups are represented as matrices and act via multiplication, and various related results.

Part II
Matrix Group Action on Polynomial Rings

1 Mathematical Description

A natural question for the reader to ask is "what does it mean for a matrix to act on a polynomial?". There is a standard definition in the literature as follows.

Let R be a commutative ring. Consider the polynomial ring $R[x_1, x_2, ..., x_n]$, where $x_1, x_2, ..., x_n$ are indeterminates.

Consider also the matrix group $G \subset M_{n\times n}(R)$, where $M_{n\times n}(R)$ denotes the set of all $n \times n$ matrices with entries from the ring R. An element A of this matrix group can be represented as:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Given a polynomial $p(x_1, x_2, ..., x_n)$ from $R[x_1, x_2, ..., x_n]$ and a matrix $A \in G$, the action of A on p is defined as the substitution of the transformed vector of indeterminates, resulting from the matrix-vector multiplication, into p. Mathematically, this can be described as:

$$A \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n \end{bmatrix}$$

Following the matrix-vector multiplication, the action of A on p replaces each indeterminate x_i in p with the respective expression from the resulting vector. Therefore, the polynomial p is transformed to:

$$p(a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n, a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n, ..., a_{n1}x_1+a_{n2}x_2+\cdots+a_{nn}x_n)$$

1.1 A simple example

For instance, consider the matrix

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$$

The action of A on the polynomial $p(x, y)$ will transform the polynomial to $p(y, 2x + y)$.

In essence, the implemented code allows for the natural action of matrix group elements on polynomials by altering the indeterminates based on the matrix-vector multiplication of the given matrix with the vector of indeterminates.

2 Implementation

2.1 Interaction with existing coercion framework

The implementation for this was somewhat involved, due to the intricate nature of SageMath's coercion framework. Although writing a function to perform the matrix multiplication and invoking the `MPolynomialRing.subs`[titute] method is simple, defining a suitable infix operator that works with the sage coercion framework is more difficult.

Note that this part of the code is part of a Cython interface.

The `MPolynomialRing_base` class must implement a `_get_action_` method, which (as standard within the coercion model) needs to make a recursive `_get_action_` call up the class hierarchy.

```
cpdef _get_action_(self, G, op, bint self_on_left):
    [...]
    # import inside function to avoid circular import
    from sage.groups.matrix_gps.matrix_group import MatrixGroup_generic
```

```
    if isinstance(G, MatrixGroup_generic) and
         self.base_ring().has_coerce_map_from(G.base_ring()) and
         op==matmul and not self_on_left:
        return MatrixPolynomialAction(G, self)
    return super(MPolynomialRing_base, self)._get_action_(G, op,
         self_on_left)
```

We also need to define a subclass of `Action` (to be returned by `_get_action_`) that takes the operands' parents classes at instantiation time, and the operands themselves upon the invocation of the `_act_` method, and performs the actual computation.

```
class MatrixPolynomialAction(Action):
    def __init__(self, MS, PR):
        self._poly_vars = PR.gens()
        self._vars_vector =
            MatrixConstructor(self._poly_vars).transpose()
        super().__init__(MS, PR, op=matmul)

    def _act_(self, mat, polynomial):
        assert mat.base_ring()==polynomial.base_ring()
        vars_to_sub_module_context=mat*self._vars_vector
        vars_to_sub_ring_context=map(PolynomialRing(mat.base_ring(),
            self._poly_vars), vars_to_sub_module_context)
        substitution_dict={v:s for v,s in zip(self._poly_vars,
            vars_to_sub_ring_context)}
        return polynomial.subs(substitution_dict)
```

Note that the matrix multiplication in the third line of `_act_` returns expressions which need to be manually coerced into ring elements. Furthermore, after discussion on the forum, we agreed that the `@` symbol, which is generally used for matrix multiplication within the sage coercion framework and is referred to as `operator.matmul`, was most appropriate to use for this action. Examples of its use are available within the doctests section.

Part III
Linearly Independent Invariant Generators

3 Invariant Polynomials Under Group Actions

Given a matrix group $G \subset M_{n\times n}(R)$, which acts on the polynomial ring $R[x_1, x_2, ..., x_n]$ as described in the previous section, we aim to show that the set of invariant polynomials under this action forms a ring.

Let $\mathcal{I}$ be the set of all invariant polynomials in $R[x_1, x_2, ..., x_n]$ under the action of G. A polynomial p is said to be invariant if, for every matrix A in G, $A \cdot p = p$.

3.1 Action Distributes Over Addition

For any two polynomials p_1 and p_2 and any matrix $A \in G$:

$$\begin{aligned} A \cdot (p_1 + p_2) &= p_1(A \cdot \mathbf{x}) + p_2(A \cdot \mathbf{x}) && \text{(Expanding based on the matrix action)} \\ &= A \cdot p_1 + A \cdot p_2 && \text{(Grouping the terms)} \end{aligned}$$

Where $\mathbf{x}$ is the vector of indeterminates $[x_1, x_2, ..., x_n]^T$.

3.2 Proof that the sum of two invariant polynomials is invariant

Let p_1 and p_2 be two invariant polynomials in $\mathcal{I}$. For any matrix A in G:

Using the above established property:

$$A \cdot (p_1 + p_2) = A \cdot p_1 + A \cdot p_2$$

Now, since both p_1 and p_2 are invariant:

$$A \cdot p_1 = p_1$$
$$A \cdot p_2 = p_2$$

Substituting the above in, we get:

$$A \cdot (p_1 + p_2) = p_1 + p_2$$

This shows that the sum $p_1 + p_2$ is also invariant under the action of G.

3.3 Action Distributes Over Multiplication

For any two polynomials p_1 and p_2 and any matrix $A \in G$:

$$\begin{aligned} A \cdot (p_1 \cdot p_2) &= p_1(A \cdot \mathbf{x}) \cdot p_2(A \cdot \mathbf{x}) \quad \text{(Using the definition of matrix action)} \\ &= (A \cdot p_1) \cdot (A \cdot p_2) \qquad\qquad\qquad\qquad \text{(Grouping the terms)} \end{aligned}$$

Again, where $\mathbf{x}$ is the vector of indeterminates $[x_1, x_2, ..., x_n]^T$.

3.4 Proof that the product of two invariant polynomials is invariant

Let p_1 and p_2 be two invariant polynomials in $\mathcal{I}$. For any matrix A in G:

Using the above established property:

$$A \cdot (p_1 \cdot p_2) = (A \cdot p_1) \cdot (A \cdot p_2)$$

Again, since both p_1 and p_2 are invariant:

$$A \cdot p_1 = p_1$$
$$A \cdot p_2 = p_2$$

Substituting the above in, we obtain:

$$A \cdot (p_1 \cdot p_2) = p_1 \cdot p_2$$

This demonstrates that the product $p_1 \cdot p_2$ is also invariant under the action of G.

In conclusion, from the above results, it's evident that the set of invariants $\mathcal{I}$ under the action of G is closed under both addition and multiplication, and thus forms a ring.

4 Finding a suitable parametrisation of the set of invariants

This simplifies the process of representing the full space of invariants, since now we only need to describe the ring. In essence, this boils down to finding a complete set of generators to precisely parametrise the polynomials.

Upon starting this project, there was a partial implementation of a general method for finding this for any finite subgroup $G \subset M_{n\times n}(R)$ of the general linear group, however there was a key issue with the implementation.

The implementation followed a broad strategy of, for each necessary degree level, computing the dimensionality $d = Dim(R[\mathbf{x}]^G)$ of the space, and then attempting to find d unique invariants of that degree to act as a basis of the

invariant space. However the approach failed to take into account the fact that invariants that were linearly dependent wouldn't act as a suitable basis for the space, and was hence flawed and often only described a subset of all invariants.

4.1 High level description of the existing implementation

Understanding the existing implementation requires knowledge of the Reynolds operator - a cheap way of calculating invariant polynomials.

Reynolds Operator: we define an operator

$$* : R[\mathbf{x}] \mapsto R[\mathbf{x}]^G$$

defined as

$$f^* := \frac{1}{|G|} \sum_{\pi \in G} f \circ \pi$$

where $\circ$ denotes the group action.

The Reynolds operator has the extremely useful property that it always outputs invariant polynomials!

The existing implementation in sage follows a structure that is common in various invariant generator algorithms in the literature. The general idea is to systemically iterate through all possible monomials, and apply the Reynolds operator to each of them. This will, in a finite amount of time, eventually generate a linearly independent basis of the invariant space. We may come across many duplicate or linearly dependent invariants before having finally spanned the full space, but the computations are quick enough that this is not problematic, and this final loop usually takes well under a second.

```
# the existing, flawed implementation
def invariants_of_degree(self, deg, chi=None, R=None):
   [...]
    for e in IntegerVectors(deg, D):
        F = self.reynolds_operator(R.monomial(*e), chi=chi)
        if not F.is_zero():
            F = F / F.lc()
            inv.add(F)
            if len(inv) == ms[deg]:
                break
    return list(inv)

# the fixed implementation
def invariants_of_degree(self, deg, chi=None, R=None):
    [...]
    for e in IntegerVectors(deg, D):
        F = self.reynolds_operator(R.monomial(*e), chi=chi)
        if not F.is_zero() and
             _new_invariant_is_linearly_independent((F:=F/F.lc()), inv):
            inv.add(F)
            if len(inv) == ms[deg]:
```

```
            break
    return list(inv)

def _new_invariant_is_linearly_independent(F, invariants):
    if len(invariants)==0:
        return True
    return PolynomialSequence(invariants).coefficient_matrix()[0].rank()
         !=
         PolynomialSequence(list(invariants)+[F]).coefficient_matrix()[0].rank()
```

Checking for linear independence can be done efficiently using the existing linear algebra backend already present in sage.

The `PolynomialSequence` class has a `coefficient_matrix` method, so verifying independence boils down to computing the rank with and without the new potential invariant.

There is room for optimisation here, e.g. we could keep track of a row-echelon form of the existing basis coefficient matrix instead, however at present the code is already fast enough.

Part IV
Thorough Invariant Generator Tests - Two Approaches

The existing (flawed) implementation of `invariant_generators` had gone undetected in the SageMath codebase for over 6 years.

Part of the reason for this is that at a glance, the results are not intuitive enough to easily spot mistakes - spotting linear dependences within lists of polynomials is by no means obvious at a glance.

This motivated us to put some more thorough checks in to verify the sets of invariants returned were exactly correct. Broadly speaking, there are two approaches we take to checking this - direct verification of invariants, and comparison of the group Molien series with the invariant ring's Hilbert–Poincaré series.

Although somewhat distinct in nature, these two tests complement each other, in the sense that the first test ensures that we don't return too many invariants, and the second ensures we don't return too few.

The code in this section is not directly exposed in any sagemath interface, but is instead used inside doctests.

5 Direct verification of invariants

The first approach we can take to verify that the invariants are correct is to manually check that they are indeed invariant under the group action.

The invariant rings are always infinite, so we cannot exhaustively check them, however the proofs from earlier show that the invariants are closed under the ring operations, meaning that (assuming the ring operations have been implemented correctly) it suffices to ensure the generators are indeed invariants. That being said, for completeness, we can also randomly sample non-generator ring elements.

The groups are finite so we *could* check every element against the invariants, however since they can be large in size, this can be too computationally demanding, thus we randomly sample elements instead.

```
# checking each generator
sage: def test_generators(group):
....:     invs=group.invariant_generators()
....:     for inv in invs:
....:         assert group.random_element()@inv==inv

# random group and ring elements
sage: def test_invariants(group):
....:     invs=group.invariant_generators()
....:     invariant_ring=invs[0].parent().subring_generated_by(invs)
```

```
....:     for i in range(N_TESTS):
....:         inv=invariant_ring.random_element().element()
....:         assert group.random_element()@inv==inv
```

6 Hilbert–Poincaré series of invariant ring comparison with group Molien series

The above tests ensure that the invariants returned are indeed invariants, however as discussed earlier, there was a bug in the existing implementation which often only led to a subset of the invariants being returned. Bugs like this would go undetected in the above tests.

It follows that we need to adopt another approach to ensure that the set of invariants returned is not incomplete.

We'll introduce two concepts which will aid in computing these tests.

6.1 Molien's Formula

is a key tool in mathematical representation theory that provides a method to calculate the number of invariant polynomials of a given degree for a specific group action.

Given a finite-dimensional complex representation V of G, and $R_n = \mathbb{C}[V]_n$ the space of homogeneous polynomial functions on V of degree n, if G is a finite group, the series can be computed as

$$\sum_{n=0}^{\infty} \dim(R_n^G) t^n = |G|^{-1} \sum_{g \in G} \det(1 - tg|V^*)^{-1}$$

Where R_n^G is the subspace of R_n that is invariant under the group action of G. This means that if we compute the coefficients of the series, we can calculate the dimension of the space of invariant polynomials of each degree.

6.2 Hilbert–Poincaré series

The Hilbert series is a tool used in algebra to understand the growth of the dimension of the homogeneous components of a graded commutative algebra, which in this case, is the ring of invariant polynomials, $R[x_1, ..., x_n]$. Note that R is a field. It is defined as a formal series $HS_S(t) = \sum_{n=0}^{\infty} HF_S(n)t^n$, where $HF_S : n \longmapsto \dim_R S_n$ is the Hilbert function that maps an integer n to the dimension of the R-vector space S_n. If the algebra S is generated by h homogeneous elements of positive degrees $d_1, \ldots, d_h$, then the Hilbert series $HS_S(t)$ can be expressed as a rational fraction of a polynomial $Q(t)$ and the product of terms $1 - t^{d_i}$, $i = 1, ..., h$. **This series encapsulates the growth rate of the dimensions of the graded components of the algebra.**

6.3 The check

When we compute the Hilbert series of the ring of invariants, if it matches the Molien series of G, then we have strong evidence that we have found the complete set of invariants. This is because both series encapsulate the growth rate of the dimensions of the graded components of the algebra of invariants. A mismatch would indicate that the space of invariants found does not have the expected dimension in some degrees, and thus is not the complete space of invariants. Conversely, if the two series match, this means that the dimensions of the spaces of invariants coincide with the expected dimensions for all degrees, which is a strong indication that the complete set of invariants has been identified.

Verifying that the returned set of invariants is complete then boils down to checking that the group's Molien formula is equal to the invariant ring's Hilbert–Poincaré formula, as shown below.

At the start of this project, there was also a bug in the hilbert series implementation, namely the homogeneity check was incorrect. As a result, we cannot treat the calculation as being 100% reliable. Therefore, as an extra check, we also ensure that both existing implementations ("sage" and "singular") return the same result.

```
sage: def test_hilbert(group):
....:     invs=group.invariant_generators()
....:     R=invs[0].parent()
....:     subring=R.subring_generated_by(invs)
....:     h1=subring.hilbert_series(algorithm="sage")
....:     h2=subring.hilbert_series(algorithm="singular")
....:     m=group.molien_series(return_series=False)
....:     assert h1==h2 and h2==m
```

Part V
Graded Homogeneity Checking

7 Bug in the existing Hilbert–Poincaré series implementation

As mentioned above, at the start of this project, there was a bug in the existing implementations of the Hilbert–Poincaré series.

The existing function was located in the `MPolynomialIdeal` class. The Hilbert–Poincaré series is only well defined for homogeneous ideals, and there was a test at the start of the implementation to verify that the ideal was indeed homogeneous. The function also accepts a grading argument, which was necessary for our use case. However, the homogeneity check didn't take the grading into account, meaning that when the grading argument was supplied, the function would proceed when it shouldn't, and wouldn't proceed when it should. To fix this, it was necessary to replace the `is_homogeneous` check with a more comprehensive one. Furthermore, the existing checks interfaced to LibGAP's `p_IsHomogeneous` method on the backend, which didn't implement a way to pass grading parameters.

As such, it was necessary to entirely rewrite the method in a more general way that accepted grading as an argument, and call it appropriately from within the `hilbert_series` methods.

```
class MPolynomialIdeal(...):
   [...]
   def is_homogeneous(self, grading=None):
      [...]
       if not grading:
           for f in self.gens():
               if not f.is_homogeneous():
                   return False
       elif not isinstance(grading, (list, tuple)) or any(not
            isinstance(x, sage.rings.integer.Integer) for x in grading):
           raise TypeError("grading must be a list or a tuple of
                integers")
       else:
           grading_dict={var:grade for var, grade in
                zip(self.ring().gens(), grading)}
           for f in self.gens():
               if not f.is_homogeneous_with_grading(grading_dict):
                   return False

cdef class MPolynomialRing_libsingular(MPolynomialRing_base):
   [...]
   def is_homogeneous_with_grading(self, grading):
      [...]
```

```
if self==0:
  return True

variables=self.parent().gens()

# degree of a monomial wrt grading
monomial_degree = lambda exponents :
    sum(grading[variable]*exponent for variable, exponent in
    zip(variables, exponents))

# calculate degree of first term
terms=iter(self.exponents())
first_term=next(terms)
first_term_degree=monomial_degree(first_term)

# make sure all other degrees are the same
for term in terms:
  if monomial_degree(term)!=first_term_degree:
      return False
return True
```

Part VI
Multivariate Polynomial Subrings

8 Motivation

In this project, we are heavily dealing with multivariate polynomial subrings generated by sets of polynomials. At the start of this project, there was no existing implementation of such objects in the SageMath codebase.

Furthermore, in its existing state, the `invariant_generators` function returned a list of polynomial objects representing the entire space, which goes against the usual SageMath design principal of having dedicated objects for mathematical spaces. This list of polynomials clearly doesn't have any of the attached functionality someone using the subring would hope for. Many existing sagemath functions would return group, ring or field objects, so the format of the function in its existing state was not consistent with much of the existing codebase.

9 Containment testing & Required Functionality

Implementing a subring type with polynomial generators is algorithmically nontrivial, since many common operations, like containment testing, are not simple.

E.g. if our subring has generators $y^3 + x^2, 4x^8 + 5y$, which of the following lies within the subring?

$$A = -32x^{16} - 4x^8y^3 - 4x^{10} - 80x^8y - y^6 - 2x^2y^3 - x^4 - 5y^4 - 5x^2y + 10y^3 + 10x^2 - 50y^2 + 2$$
$$B = -16x^{16} - 40x^8y + 4x^8 - y^3 - x^2 - 25y^2 + x + 5y + 12$$
$$C = -16x^{16} - 40x^8y - 4x^8 - y^6 - 2x^2y^3 - x^4 + y^3 + x^2 - 25y^2 - 4y - 1$$

The answer is $A = -(y^3 + x^2)^2 - (y^3 + x^2)(4x^8 + 5y) - 2(4x^8 + 5y)^2 + 10(y^3 + x^2) + 2$, though it should be clear that determining containment is nontrivial. However, with some clever interfacing to a `QuotientRing` backend, much of the logic did already exist, albeit under a different name.

The functionality of describing how to construct an element wrt the generators (as above) is handled in the `construct()` method.

It was thus necessary to add a `MPolynomial_subring` class to handle this logic. To make many of the inherited methods work, it was also necessary to define a class for the subring elements.

10 Implementation

```
class MPolynomial_subring(MPolynomialRing_libsingular):
```

```
def __init__(self, parent_ring, gens):
    self._hom=PolynomialRing(parent_ring.base_ring(), len(gens),
        "a").hom(gens)
    self.generators=[self._element_constructor_(gen) for gen in gens]
    self.parent_ring=parent_ring
    self._ideal=self._hom.kernel()
    self._zero_element = self._element_constructor_(0)
    self._one_element = self._element_constructor_(1)
    self._homdomain = self._hom.domain()

def _element_constructor_(self, element):
    assert element in self
    return MPolynomial_subring_element(self, element)

# both spellings are used in methods inherited from
    MPolynomial_libsingular
def _element_constructor(self, *args, **kwargs):
    return self._element_constructor_(*args, **kwargs)

def __call__(self, *args, **kwargs):
    return self._element_constructor_(*args, **kwargs)

# explicitly describe how to construct an element in terms of the
    ring generators
def construct(self, element, return_string=False):
    try:
        quotientring_element=self._quotientring_element_representation_(element)
        mapping_string=", ".join([f"{qg}=({rg})" for qg,rg in
            zip(self._homdomain.gens(), self.gens())])
        construction_string=f"{quotientring_element}\nWhere
            {mapping_string}"
        if return_string:return construction_string
        else: print(construction_string)
    except ValueError:
        raise ValueError(f"{element} cannot be constructed in
            {self}.")

def _quotientring_element_representation_(self, element):
    assert isinstance(element, MPolynomial_subring_element) and
        element._parent is self
    return self._hom.inverse_image(element.element())


def __repr__(self):
    return f"Subring of {repr(self.parent_ring)} generated by
        {self.gens()}"

def gens(self):
    return self.generators
```

```
    def ngens(self):
        return len(self.generators)

    def hilbert_series(self, algorithm="sage"):
        grading=[s.degree() for s in self.generators]
        return self._ideal.hilbert_series(grading=grading,
            algorithm=algorithm)

    def __contains__(self, other):
        try:
            if isinstance(other, MPolynomial_subring_element):
                self._hom.inverse_image(other.element())
                return True
            else:
                self._hom.inverse_image(other)
                return True
        except ValueError as e:
            return False

    def random_element(self):
        return
            self._element_constructor_(self._hom(self._homdomain.random_element()))

    def _an_element_(self):
        return
            self._element_constructor_(self._hom(self._homdomain._an_element_()))

class MPolynomial_subring_element(MPolynomial_element):

    def __init__(self, parent, x):
        self._parent=parent
        super().__init__(parent, x)

    def degree(self):
        return self.element().degree()

    def construction(self, *args, **kwargs):
        return self._parent.construct(self, *args, **kwargs)

    def __eq__(self, other):
        return self.element()==other.element()

    def as_quotientring_element(self, *args, **kwargs):
        return self._parent._quotientring_element_representation_(self,
            *args, **kwargs)
```

Most of the above code just enables standard SageMath and python functionality to work; the majority of required functionality is derived from the

respective superclasses.

The "interesting" functionality is found in the `__contains__`, `hilbert_series` and `construct` methods.

Part VII
Doctests

SageMath has an automated testing framework, allowing for tests to be written inside docstrings. All tests included in this section are passing at the time of writing.

Lines starting with "`sage:`" or "`...:`" should be interpreted as lines run by the testing framework, and lines without such prefixes should be interpreted as expected outputs. Annotations have been included in the first doctest to demonstrate this.

11 Matrix Group Action on Polynomial Rings

```
sage: G=groups.matrix.Sp(4,GF(2))
sage: R.<w,x,y,z>=GF(2)[]
sage: p=x*y^2 + w*x*y*z + 4*w^2*z+2*y*w^2
sage: g=G.1
sage: g
[0 0 1 0]
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
   sage:  g@p
run by testing framework
   w*x*y*z + w*z^2
expected output from above line
sage: p2=x+y^2
sage: g2=G.0
sage: g2
[1 0 1 1]
[1 0 0 1]
[0 1 0 1]
[1 1 1 1]
sage: g2@p2
x^2 + z^2 + w + z
```

12 Linearly Independent Invariant Generators

```
sage: gens = [matrix(QQ, [[-1,1],[-1,0]]), matrix(QQ, [[0,1],[1,0]])]
sage: G = MatrixGroup(gens)
sage: s = Sequence(G.invariants_of_degree(14))
sage: s.coefficient_matrix()[0].rank()
```

```
3
sage: len(s)
3
```

---

13 Thorough Invariant Generator Tests - Two Approaches

The approach used to test this was to compile a diverse collection of groups, and then run the `test_invariants`, `test_generators` and `test_hilbert` functions on them. Note that in this doctest in particular, it was necessary to balance the trade-off between speed and coverage.

```
sage: K = CyclotomicField(4)
sage: i=K.gen()
sage: tetra=MatrixGroup([(-1+i)/2,(-1+i)/2, (1+i)/2,(-1-i)/2], [0,i,
    -i,0])
sage: test_generators(tetra)
sage: test_invariants(tetra)
sage: test_hilbert(tetra)
sage: for n in range(2,4):
....:     M = MatrixSpace(QQ, n)
....:     identity=MatrixGroup([M.identity_matrix()])
....:     alternating=MatrixGroup([M(g.matrix()) for g in
    AlternatingGroup(n).gens()])
....:     symmetric=MatrixGroup([M(g.matrix()) for g in
    SymmetricGroup(n).gens()])
....:     for g in (identity, alternating, symmetric):
....:         test_generators(g)
....:         test_invariants(g)
....:         test_hilbert(g)
sage: K = CyclotomicField(8)
sage: v=K.gen()
sage: a = v-v**3 #sqrt(2)
sage: i = v**2
sage: octa = MatrixGroup([(-1+i)/2, (-1+i)/2, (1+i)/2,
    (-1-i)/2],[(1+i)/a, 0, 0, (1-i)/a])
sage: test_generators(octa)
sage: test_invariants(octa)
sage: test_hilbert(octa)
sage: K = CyclotomicField(10)
sage: v=K.gen()
sage: z5 = v**2
sage: i = z5**5
sage: a = 2*z5**3 + 2*z5**2 + 1 #sqrt(5)
sage: Ico = MatrixGroup([[z5**3,0, 0,z5**2],[0,1, -1,0],[(z5**4-z5)/a,
    (z5**2-z5**3)/a, (z5**2-z5**3)/a, -(z5**4-z5)/a]])
sage: test_generators(Ico)
```

```
sage: test_invariants(Ico)
sage: test_hilbert(Ico)
sage: K = GF(5)
sage: S = MatrixGroup(SymmetricGroup(4))
sage: G = MatrixGroup([matrix(K, 4, 4, [K(y) for u in m.list() for y in
    u])for m in S.gens()])
sage: test_generators(G)
sage: test_invariants(G)
sage: test_hilbert(G)
sage: i = GF(7)(3)
sage: G = MatrixGroup([[i**3,0, 0,-i**3], [i**2,0, 0,-i**2]])
sage: test_generators(G)
sage: test_invariants(G)
sage: test_hilbert(G)
```

14 Graded Homogeneity Checking

```
sage: R.<a,b,c> = QQ[]
sage: grading_dict = {a:1, b:2, c:3}
sage: p = a**4+b**2
sage: p.is_homogeneous_with_grading(grading_dict)
True
sage: grading_dict = {a:1, b:1, c:3}
sage: p.is_homogeneous_with_grading(grading_dict)
False
```

15 Multivariate Polynomial Subrings

```
sage: R.<a,b,c>=ZZ[]
sage: s=R.subring_generated_by([a**2, b**3+c])
sage: e=s(b^6 - 4*a^2*b^3 + a^4 + 2*b^3*c - 4*a^2*c + c^2 + 2)
sage: e.construction()
a0^2 - 4*a0*a1 + a1^2 + 2
Where a0=(a^2), a1=(b^3 + c)

sage: R.<a,b,c>=ZZ[]
sage: s=R.subring_generated_by([a**2, b**3+c])
sage: e=s(b^6 - 4*a^2*b^3 + a^4 + 2*b^3*c - 4*a^2*c + c^2 + 2)
sage: e.as_quotientring_element()
a0^2 - 4*a0*a1 + a1^2 + 2

sage: R.<a,b>=QQ[]; S=R.subring_generated_by([a**2+b])
sage: S
```

```
Subring of Multivariate Polynomial Ring in a, b over Rational Field
    generated by [a^2 + b]

sage: G.<x,y,z> = GF(5)[]
sage: SG=G.subring_generated_by([x**2+y**3,z**3+y**2])
sage: SG.hilbert_series()
1/(t^6 - 2*t^3 + 1)

sage: R.<a,b,c>=ZZ[]
sage: S=R.subring_generated_by([a**2, b**3+c])
sage: b in S
False
sage: b**3 in S
False
sage: b**3 + c in S
True
sage: S2=R.subring_generated_by([a**2, b**3+c, c])
sage: b**3 in S2
True

sage: R.<a,b,c>=ZZ[]
sage: s=R.subring_generated_by([a**2, b**3+c])
sage: e=s.random_element()
sage: se=s._quotientring_element_representation_(e)
sage: s(s._hom(se))==e
True
```

Part VIII
Credits

Although I was the one who *contributed* all of the code in this document, I owe a large thanks to multiple forum contributors for countless pointers and suggestions, most notably for getting my matrix group action code to work within the coercion framework. Without them, I would have achieved a fraction of what I ended up achieving in this project.

Most importantly, I am hugely grateful to my supervisor for patiently teaching me everything I needed to know to attack these problems, and continually helping me decide which direction to take this project. None of this could have been achieved without his vast mathematical knowledge and intimate familiarity with the SageMath codebase, and I am extremely lucky to have had this opportunity to work with him.

Part IX
Closing Remarks

Before starting my work on SageMath, I had virtually no knowledge on the field of abstract algebra, and I had never contributed to any open-source projects before. I am grateful for Google for giving me this opportunity, and for my fellow SageMath contributors on GitHub for their patience and support. This project has been a fantastic experience, and I'd wholeheartedly recommend participating in Google Summer of Code to anyone interested in open-source development.