AMSI VACATION RESEARCH SCHOLARSHIPS 2019–20

EXPLORE THE MATHEMATICAL SCIENCES THIS SUMMER


# A Computational Approach to the Conjugacy Problem

Amelia Lee

Supervised by Dr. Adam Sierakowski

And Dr. Ben Whale

University of Wollongong

Vacation Research Scholarships are funded jointly by the Department of Education and Training and the Australian Mathematical Sciences Institute.

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

Australian Government Department of Education

**Abstract**

The integral conjugacy problem arises in several fields of mathematics, such as number theory and graph theory in the construction of adjacency matrices. This decision problem was proven to be decidable in 1980, when an algorithm was proposed in a paper by Fritz Grunewald. Very recently, a team of researchers at the University of Sydney implemented the first program that solves the integral congugacy problem using the computer algebra system Magma. They primarily followed the logic of Grunwald's original algorithm, bridging several gaps in the procedure. In this paper we investigate the original and adapted algorithms, and program the beginnings of an open source implementation of the algorithm in Python.

# 1 Introduction

The conjugacy problem is recognised as one of the fundamental decision problems in group theory. First identified in 1911 by Max Dehn [1], the most general definition of the problem is stated below.

**Definition 1** (General Conjugacy Problem)**.** Given a group $G$ and two elements $x, y \in G$, determine if there exists $z \in G$ such that

$$zxz^{-1} = y \tag{1}$$

It is known that the conjugacy problem is decidable for certain classes of groups, and undecidable for others [2]. If the problem is decidable then, in principle, an algorithm exists which can determine if the equivalences hold and this algorithm will finish execution in a finite number of calculations. The algorithm may also produce the conjugating element $z$.

Congruency problems in abstract algebra remain an active area of mathematical research. Very recent work on such problems [2, 3] relies on an algorithm developed by Fritz Grunewald in 1980 to solve the conjugacy problem in certain arithmetic groups [4]. An algorithm based off Grunewald's was implemented in [3] for the special case of the conjugacy problem over $GL_n(\mathbb{Z})$. In this paper *The Conjuacy Problem in* $GL(n, \mathbb{Z})$, Eick, Hofmann and O'Brien identify gaps in Grunwald's procedure and propose enhancements. The final algorithm presented is implemented in Magma [5], a computer algebra system developed by the Computational Algebra Group at the School of Mathematics and Statistics of the University of Sydney. Magma is a closed source software system available only to institutions by paid subscription.

The special case of the conjugacy problem for which the Magma implementation exists is referred to as the *integral conjugacy problem*. It is defined over the general linear group of invertible $n \times n$ matrices as follows.

**Definition 2** (Integral Conjugacy Problem)**.** Given $T, \hat{T} \in GL_n(\mathbb{Q})$, determine if there exists $X \in GL_n(\mathbb{Z})$ such that

$$XTX^{-1} = \hat{T}. \tag{2}$$

The general form of the algorithm that Grunewald describes replaces $GL_n(\mathbb{Q})$ and $GL_n(\mathbb{Z})$ respectively with an algebraic number field $K$ that is a finite extension of the rational numbers and the ring of algebraic integers in $K$. There is no implementation of the general form of Grunewald's algorithm.

This paper follows Grunewald's algorithm with a simple motivating example and presents an original set of Python functions which translate the integral conjugacy problem into ring isomorphism and homomorphism problems. The final step in Grunwald's paper was not completed due to time contraints. Each component of the code was developed using a modular, test driven approach and could be adapted to the general form of the algorithm relatively easily. Python was chosen because it is a widely used language with substantial libraries that support algebraic manipulation. In particular, the NumPy [6] and SymPy [7] libraries provide valuable tools for matrix operations.

## 2 Statement of Authorship

All code presented in this paper is the original work of Amelia Lee, reviewed for style and consistency by Dr. Ben Whale. This report was authored by Amelia Lee under the guidance and supervision of Dr. Adam Sierakowski and Dr. Ben Whale. All the ideas presented in this paper are original or attributable to the referenced sources.

## 3 The 'Direct Approach'

To gain an intuition for the problem, we will consider how we might solve the integral conjugacy problem by a 'direct' method. For problems of a small magnitude, this is relatively computationally inexpensive. In the case of $2 \times 2$ matrices we have $T, \hat{T} \in GL_2(\mathbb{Q})$ and $X \in GL_2(\mathbb{Z})$:

$$T = \begin{pmatrix} t_{1,1} & t_{1,2} \\ t_{2,1} & t_{2,2} \end{pmatrix} \qquad \hat{T} = \begin{pmatrix} \hat{t}_{1,1} & \hat{t}_{1,2} \\ \hat{t}_{2,1} & \hat{t}_{2,2} \end{pmatrix} \qquad X = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}. \tag{3}$$

The inverse of $2 \times 2$ matrix is simple to compute, so we can also write $X^{-1}$ as

$$X^{-1} = \frac{1}{x_{1,1}x_{2,2} - x_{1,2}x_{2,1}} \begin{pmatrix} x_{2,2} & -x_{1,2} \\ -x_{2,1} & x_{1,1} \end{pmatrix}. \tag{4}$$

Expanding the expression $XTX^{-1} = \hat{T}$ gives the following system of quadratic equations.

$$\hat{t}_{1,1} = \frac{x_{2,2}(x_{1,1}t_{1,1} + x_{1,2}t_{2,1}) - x_{2,1}(x_{1,1}t_{1,2} + x_{1,2}t_{2,2})}{x_{1,1}x_{2,2} - x_{1,2}x_{2,1}} \tag{5}$$

$$\hat{t}_{1,2} = \frac{-x_{1,2}(x_{1,1}t_{1,1} + x_{1,2}t_{2,1}) + x_{1,1}(x_{1,1}t_{1,2} + x_{1,2}t_{2,2})}{x_{1,1}x_{2,2} - x_{1,2}x_{2,1}} \tag{6}$$

$$\hat{t}_{2,1} = \frac{x_{2,2}(x_{2,1}t_{1,1} + x_{2,2}t_{2,1}) - x_{2,1}(x_{2,1}t_{1,2} + x_{2,2}t_{2,2})}{x_{1,1}x_{2,2} - x_{1,2}x_{2,1}} \tag{7}$$

$$\hat{t}_{2,2} = \frac{-x_{1,2}(x_{2,1}t_{1,1} + x_{2,2}t_{2,1}) + x_{1,1}(x_{2,1}t_{1,2} + x_{2,2}t_{2,2})}{x_{1,1}x_{2,2} - x_{1,2}x_{2,1}} \tag{8}$$

Suppose that we wish to solve the problem for particular matrices:

$$T = \begin{pmatrix} \frac{1}{2} & 1 \\ 1 & \frac{1}{2} \end{pmatrix} \qquad \hat{T} = \begin{pmatrix} -\frac{7}{2} & 3 \\ -5 & \frac{9}{2} \end{pmatrix}. \tag{9}$$

Substituting the entries of $T$ and $\hat{T}$ defined above, a root finding algorithm can compute $X$:

$$X = \begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix}. \tag{10}$$

We can easily verify that the equivalence holds by hand:

$$\begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} \frac{1}{2} & 1 \\ 1 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 3 & 2 \end{pmatrix}^{-1} = \begin{pmatrix} -\frac{7}{2} & 6 \\ -5 & \frac{9}{2} \end{pmatrix}. \tag{11}$$

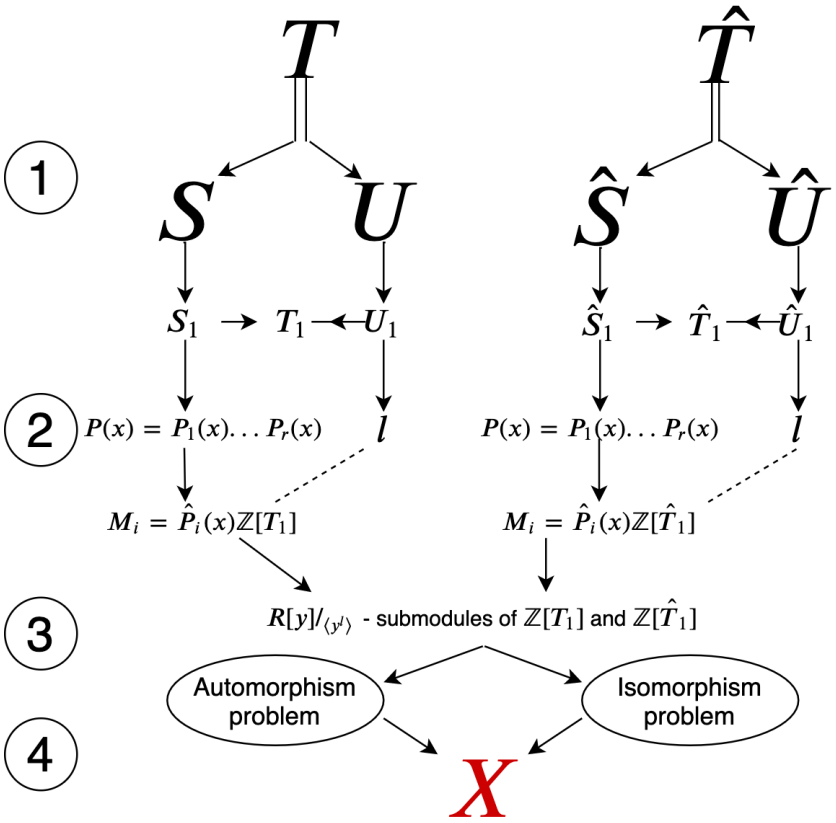
This is a reasonable approach for small, well conditioned probems. However, as $n$ increases in magnitude, it is more computationally expensive to find a closed form solution of the matrix inverse, and we must solve a system of $n^2$ polynomial equations in $n$ variables of degree $n$. In the example above, a numerical solver was used to construct the solution. Since the solution must be over the integers, any degree of error introduced by a numerical method is unacceptable. Further, as James H. Wilkinson demonstrated in 1963 with the 'Wilkinson's polynomial,' root finding algorithms that employ floating point arithmetic fail for ill-conditioned polynomials [8].

There are several known algorithms for solving systems of polynomials over integers, some of which are deterministic. Daniel Lokshtanov et al. discuss a number of deterministic and non-deterministic algorithms for solving systems of polynomials over finite fields, which can be adapted to solving over the ring of integers [9]. All the algorithms presented have exponential complexity. Alternatively, methods such as the Extended Dixon Resultant, DR or XL methods can achieve subexponetial complexity, at

the expense of failing if the system is ill-conditioned [10]. A major drawback of these methods is that most have no 'early exit' points; the algorithms may still run to completion even if the system has no solution over the integers. Early exit point are critical for improving the performance of decision problem algorithms.

## 4 Grunewald's Approach

The following sections demonstrate how the conjugating matrix $X$ can be constructed using Grunewald's approach. Grunewald's paper is divided into two parts. The first part describes how the algorithm produces the required matrix $X$, if it exists, by a three step procedure. The second part describes 20 specific algorithms that are used in the procedure. The paper mixes definitions, lemmada and proofs in with both the algorithms and the procedure. In this paper proofs are omitted and, wherever possible, steps are simplified. The following diagram summarises the general flow of the algorithm, with a focus on the first half.




(12)

The final implementation covers the steps labelled 1 and 2. Step 3 requires that all the standard submodules of a given module must be constructed. Grunewald provides no practical algorithm for this critical step. The ability to construct these standard submodules is the crux of the procedure, which relies on several significant results that are glossed over in the paper. Eick, Hoffman and O'Brien's 2019 paper *The Conjugacy Problem in* $GL(n, \mathbb{Z})$ bridges this gap [3]. The crux proved far more mathematically complex and programmatically difficult to implement in comparison to the preceeding steps in Grunewald's algorithm. Therefore, only the implementation of the first half of the algorithm is presented here.

## 5 The procedure

In any programmatic implementation of an algorithm, there is a trade-off between optimising the performance of the algorithm and reducing development hours. Due to time constraints, this implementation prioritises development time and well structured code, even if this results in a more computationally expensive algorithm. Wherever possible, functions from existing libraries have been used.

To demonstrate the procedure, we will use the following conjugating matrices as a case study:

$$T = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \hat{T} = \begin{pmatrix} 5 & -1 & -1 \\ 4 & 0 & -1 \\ 8 & -2 & -1 \end{pmatrix}. \tag{13}$$

### Step 0

As with any program, user input should be validated before proceeding with the method. In particular, it should be checked that two square matrices of equal dimension, with rational entries and non-zero determinant are provided. By inspection, it is clear that $T$ and $\hat{T}$ fulfil these criterion.

### Step I

1. Find the Jordan-Chevalley decomposition of $T$ and $\hat{T}$.

$$T = S + U \qquad \hat{T} = \hat{S} + \hat{U} \tag{14}$$

$$\text{where } SU = US \quad \text{and} \quad \hat{S}\hat{U} = \hat{U}\hat{S}, \tag{15}$$

and $S, \hat{S}$ are rational semi-simple and $U, \hat{U}$ are rational nilpotent.

In this step, we take advantage of the relationship between the Jordan normal form of a matrix and the Jordan-Chevalley decomposition. A detailed description of the relationship is described in Peter Petersen's *Linear Algebra* [11].

The Jordan normal forms of $T$ and $\hat{T}$ are given below.

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \tag{16}$$

$$\hat{T} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & 1 \\ 4 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & 1 \\ 4 & 0 & 2 \end{pmatrix}^{-1} . \tag{17}$$

In this case, $T$ is already a Jordan matrix and $\hat{T}$ is already semi-simple. Therefore, we have

$$S = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad U = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{18}$$

and $\hat{S} = \hat{T}, \hat{U} = \mathbf{0}$.

For the general case, the Jordan-Chevalley form can be found by decomposing the Jordan matrix into the sum of its diagonal and off-diagonal elements, and expanding. The extract of code below demonstrates how the decomposition is found by this method.

Listing 1: Python implentation of Jordan-Chevalley decomposition

```
1  def jordan_chevalley(self, matrix):
2  """
3  This is algorithm is A3 in Grunewald's paper.
4
5  This algorithm finds the Jordan-Chevalley decomposition of a matrix A
6  by using the jordan normal form.
7
8  Returns [S, U] where:
9
10 S is a semi-simple matrix
11 U is a nilpotent matrix
12 A = S + U
13 SU = US
```

```
"""
        from numpy import diag as extract_diag
        norm, jordan = matrix.jordan_form()
        semi = diag(*extract_diag(jordan))
        nil = jordan - semi
        norm_inverse = Matrix.inv(norm)

        semisimple = norm * semi * norm_inverse
        nilpotent = norm * nil * norm_inverse

        return [semisimple, nilpotent]
```

2. Find a scalar matrix $kI_n$ that satisfies

$$kI_nS, kI_nU \in M_n(\mathbb{Z}) \tag{19}$$

$$kI_n\hat{S}, kI_n\hat{U} \in M_n(\mathbb{Z}). \tag{20}$$

If all the entries in the matrices are written as simplified fractions, then we can take $k$ as the lowest common multiple of the denominators. In this example we have chosen integer matrices, so $k = 1$. Therefore we define

$$T_1 = kI_nT = T \qquad S_1 = kI_nS = S \qquad U_1 = kI_nU = U \tag{21}$$

$$\hat{T}_1 = kI_n\hat{T} = \hat{T} \qquad \hat{S}_1 = kI_n\hat{S} = \hat{S} \qquad \hat{U}_1 = kI_n\hat{U} = \hat{U} \tag{22}$$

In the general case, the following code identifies the scalar $k$.

Listing 2: Python implentation of identifying scalar $k$

```
def find_denominator(self, number):
"""
Helper method that converts a number into a simplified
fraction and returns the denominator.
"""
        n, d = fraction(nsimplify(number, rational=True))
        return d

def find_scalar_multiplier(self, matrix_list):
"""
Finds the least scalar multiplier k such that k*Identity*A is an
integer matrix, where A has rational entries.
```

```
"""
        denominators = []
        for matrix in matrix_list:
                for entry in matrix._mat:
                        denominators.append(self.find_denominator(entry))

        return lcm_list(denominators)
```

Now the problem can be reduced to two problems - one concerning the semi-simple matrices and one concerning the nilpotent matrices, see [3, Lemma 2 and 4]. These problems are reduced further in the following step.

### Step II

This step translates the matrices into modules over quotient rings. Deciding the existence of the integer matrix $X$ is translated into ring isomorphism and homomorphism problems.

1. Compute minimal polynomials $\min(S_1)$, $\min(\hat{S}_1)$. If $\min(S_1) \neq \min(\hat{S}_1)$, then there is no conjugate and the algorithm should terminate. The minimal polynomial of a matrix $S$ has the same roots as the characteristic polynomial of $S$, up to multiplicity, and is the lowest order polynomial $p$ that satisfies $p(S) = 0$. Therefore, to calculate the minimal polynomial, we first compute the characteristic polynomial of each matrix.

$$\text{Char}(S_1) = \text{Char}(\hat{S}_1) = (1-\lambda)^2(2-\lambda) \tag{23}$$

The two possibilities for the minimal polynomial in this case are $(1-\lambda)(2-\lambda)$ or $(1-\lambda)^2(2-\lambda)$. We find that

$$(1-S_1)(2-S_1) = (1-\hat{S}_1)(2-\hat{S}_1) = \mathbf{0}. \tag{24}$$

Therefore the minimal polynomial for both semi-simple matices is $P(\lambda) = (1-\lambda)(2-\lambda)$. Since they are equal, the algorithm should continue. In general, the following Python code computes the minimal polynomial by using the characteristic polynomial.

Listing 3: Python code that determines the minimal polynomial of a given matrix

```
def find_minimal_polynomial(self, matrix):
"""
This is algorithm A2 in Grunewald's paper.
```

```
4
5   This algorithm finds the minimal polynomial of a matrix by first
6   finding the characteristic polynomial, then using exhaustion to try
7   every possible combination of factors from minimal to maximal until one
8   that satisfies P(M) = 0 is met, where P is the polynomial and M is the
9   matrix.
10  """
11          characteristic_poly = matrix.charpoly()
12          factors = factor_list(characteristic_poly)
13          indices = []
14
15          for factor in factors[1]:
16                  indices.append(factor[1])
17
18          for indices in self.compute_trial_indices(indices):
19                  candidate_poly = prod(
20                  factors[1][i][0] ** indices[i] for i in range(len(indices))
21                  )
22
23                  polynomial_evaluated = self.sub_matrix(candidate_poly, matrix)
24
25                  if polynomial_evaluated.is_zero:
26                          return candidate_poly
27
28  def compute_trial_indices(self, max_indices):
29  """
30  This is a helper method that computes all the possible combinations of
31  indices that a minimal polynomial's factors may have, and returns them
32  in increasing order.
33
34  :param max_indices: The power of each factor in the characteristic
35  polynomial
36  :return: All combinations of indices as an iterable product.
37  """
38          import itertools
39
40          possible_values = []
41          for element in max_indices:
42                  possible_values.append(
43                  [1 + range(element)[i] for i in range(element)]
```

```
44                 )
45
46         answer = itertools.product(*possible_values)
47
48         return sorted(answer, key=lambda x: sum(x))
```

The method called on line 23 `sub_matrix` was written because the polynomial and matrix datatypes were not compatible. The problem of incompatible datatypes and unsupported operations became increasingly difficult to overcome as the matrices were translated into problems concerning quotient rings. See Appendix A for the implementation.

2. Assuming that $P(x) = \min(S_1) = \min(\hat{S}_1)$, factorise the polynomial:

$$P(x) = P_1(x)...P_r(x). \tag{25}$$

In our example we have

$$P(x) = P_1(x)P_2(x) = (1-x)(2-x). \tag{26}$$

In the program, a Sympy [7] function was used that supports polynomial factorisation over any given domain.

3. Define $\hat{P}_i(x) = \prod_{j \neq i} P_j(x)$. Compute polynomials $Q_1, \ldots, Q_r$ with integer coefficients such that

$$\sum_{i=1}^{r} Q_i \hat{P}_i = \lambda \tag{27}$$

for a natural number $\lambda$. In this case, we can see that

$$-(1-x) + (2-x) = 1 \tag{28}$$

$$\implies Q_1 = -1, \quad Q_2 = 1. \tag{29}$$

In the general case, a repeated application of Euclid's algorithm will produce the multipliers. The following Python method takes a list of factors as parameters and returns the list of multipliers.

Listing 4: Python implementation that calculates Bezout's identity for a list of polynomials through repeated application of Euclid's algorithm

```
1  def extended_euclidean_alg(self, polynomial_list):
2  """
3  This is algorithm A1 in Grunewald's paper.
4
```

```
    Given a list of coprime polynomials, p0, ... , pk, finds the polynomial
    multipliers q0, ... , qk such that p0q0 + ... + pkqk = 1. This is done
    by repeated application of Euclid's algorithm
    """
            temp_poly = polynomial_list[0]
            q = [1]
            for polynomial in polynomial_list[1:]:
                    reduced_form = gcdex(temp_poly, polynomial, domain=self.domain)
                    q = [i * reduced_form[0] for i in q]  # update existing multipliers
                    q.append(reduced_form[1])  # add the new multiplier
                    temp_poly = reduced_form[0] * temp_poly \
                    + reduced_form[1] * polynomial
            return q
```

4. Define the quotient group

$$R = \mathbb{Z}[x]/\langle P(x)\rangle \tag{30}$$

where $\langle P(x)\rangle$ is the ideal generated by $P(x)$, and $\mathbb{Z}[x]$ is the ring of polynomials with integer coefficients. It is helpful to use the equivalence relationship

$$R \cong \mathbb{Z}[x]/\langle P_1(x)\rangle \times \cdots \times \mathbb{Z}[x]/\langle P_r(x)\rangle. \tag{31}$$

In our example, we have

$$R = \mathbb{Z}[x]/\langle (1-x)(2-x)\rangle \tag{32}$$

$$\cong \mathbb{Z}[x]/\langle 1-x\rangle \times \mathbb{Z}[x]/\langle 2-x\rangle. \tag{33}$$

Each of $\mathbb{Z}[x]/\langle 1-x\rangle$ and $\mathbb{Z}[x]/\langle 2-x\rangle$ are isomorphic to the ring of integers. To see this, consider

$$x \equiv x \mod (1-x) \tag{34}$$

$$\implies x \equiv x + (1-x) \mod (1-x) \tag{35}$$

$$\implies x \equiv 1 \mod (1-x) \tag{36}$$

$$x \equiv x \mod (2-x) \tag{37}$$

$$\implies x \equiv x + (2-x) \mod (2-x) \tag{38}$$

$$\implies x \equiv 2 \mod (2-x). \tag{39}$$

Therefore, every integer polynomial is mapped to the sum of its coefficients in $\mathbb{Z}[x]/\langle 1-x\rangle$. Likewise, any integer polynomial is mapped to the its value at $x=2$ in $\mathbb{Z}[x]/\langle 2-x\rangle$. Now we can express the polynomial ring $R$ as

$$R \cong \mathbb{Z}^2. \tag{40}$$

In the program the Sympy Ring class was used, which has support for quotient rings. See Appendix A for the implementation.

5. Find the least natural number $l$ such that

$$U_1^l = \hat{U}_1^l = 0 \tag{41}$$

This simplest way to do this is by exhaustion. For an $n \times n$ upper-triangular matrix, the $n^{th}$ power must be the zero matrix, so this method is guaranteed to terminate. In our example, we have $\hat{U}_1 = \mathbf{0}$ and can compute $U_1^2 = 0$. Therefore $l = 2$.

6. We can now define $R[y]/\langle y^l\rangle$ - module structures by

$$xv = S_1 v^t \tag{42}$$

$$yv = U_1 v^t \qquad \text{for all } v \in \mathbb{Z}[T_1] \tag{43}$$

See [4, p. 106] for the definition of $\mathbb{Z}[T_1]$ but as a set, $\mathbb{Z}[T_1] = \mathbb{Z}^3$. There is a corresponding definition for $\hat{T}$.

7. Now we decompose $\mathbb{Z}[T_1]$ and $\mathbb{Z}[\hat{T}_1]$. Define modules

$$M_i = \hat{P}_i(x)\mathbb{Z}[T_1] \tag{44}$$

$$\hat{M}_i = \hat{P}_i(x)\mathbb{Z}[\hat{T}_1] \qquad \text{where } \hat{P}_i(x) = \prod_{j\neq i} P_j(x) \tag{45}$$

8. It is here that we reach the 'crux' of the algorithm. Grunewald asks that we use a variant of Guass' algorithm to find $\mathbb{Z}$-bases for each $M_i$ and $\hat{M}_i$.

$$M = M_1 + \cdots + M_r \subseteq \mathbb{Z}[T_1] \tag{46}$$

$$\hat{M} = \hat{M}_1 + \cdots + \hat{M}_r \subseteq \mathbb{Z}[\hat{T}_1] \tag{47}$$

This is where Python program ends. However, since the example we have chosen as a case study is simple, we can proceed. For $M_1$ we have

$$M_1 = (2-x)\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = 2\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \tag{48}$$

$$= \begin{pmatrix} v_0 \\ v_1 \\ 0 \end{pmatrix} \cong \mathbb{Z}^2. \tag{49}$$

Similarly, for $\hat{M}_1$ we have

$$\hat{M}_1 = (2-x)\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = 2\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} - \begin{pmatrix} 5 & -1 & -1 \\ 4 & 0 & -1 \\ 8 & -2 & -1 \end{pmatrix}\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \tag{50}$$

$$= \begin{pmatrix} -3v_0 + v_1 + v_2 \\ -4v_0 + 2v_1 + v_2 \\ -8v_0 + 2v_1 + 3v_2 \end{pmatrix} \cong \mathbb{Z}^2. \tag{51}$$

Therefore $M_1 \cong \hat{M}_1$. Similarly for $M_2$ and $\hat{M}_2$ we have

$$M_2 = (1-x)\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \tag{52}$$

$$= \begin{pmatrix} 0 \\ 0 \\ -v_2 \end{pmatrix} \cong \mathbb{Z} \tag{53}$$

$$\hat{M}_2 = (1-x)\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} - \begin{pmatrix} 5 & -1 & -1 \\ 4 & 0 & -1 \\ 8 & -2 & -1 \end{pmatrix}\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \tag{54}$$

$$= \begin{pmatrix} -4v_0 + v_1 + v_2 \\ -4v_0 + v_1 + v_2 \\ -8v_0 + 2v_1 + 2v_2 \end{pmatrix} \cong \mathbb{Z}. \tag{55}$$

Therefore $M_2 \cong \hat{M}_2$.

In this example, it is simple to see that $M_1 \cong \hat{M}_1$ and $M_2 \cong \hat{M}_2$, which is the isomorphism problem that Grunewald describes in Step III of the algorithm. In the general case, determining the $\mathbb{Z}$-bases and the isomorphisms is not programatically straightforward.

## 6 Proposed enhancements

There are several characteristics of matrices that Grunewald does not use, which could be particularly useful for eliminating unsolvable cases early. Here is one suggested enhancement.

**Theorem 1.** *If the integral conjugacy problem has a solution for two matrices $T, \hat{T} \in GL_n(\mathbb{Q})$, then $det(T) = det(\hat{T})$.*

*Proof.* Each $X \in GL_n(\mathbb{Z})$ is unimodal; that is, $\det(X) = \pm 1$. Since determinants are distributive over multiplication, we have

$$\det(XTX^{-1}) = \det(X)\det(T)\det(X^{-1}) \tag{56}$$

$$= \pm 1 \times \det(T) \times \pm 1 \tag{57}$$

$$= \det(T) \tag{58}$$

Therefore if there exists an $X \in GL_n(\mathbb{Z})$ such that $XTX^{-1} = \hat{T}$, then $\det(T) = \det(\hat{T})$. □

By following Grunewald's algorithm, two matrices that do not satisfy this condition would not cause the program to terminate until the minimal polynomials are computed. The determinant of a matrix is inexpensive to compute, so this step should be included as part of the initial data validation.

## 7 Discussion and Conclusion

The integral congugacy problem remains difficult to solve. There is only one complete programmatic implementation of an algorithm that is guaranteed to produce the solution in a finite number of operations. The authors of this algorithm discuss that it still has drawbacks [3], and that the algorithm may be computationally infeasible for some cases. The aim of this research was to describe and begin to implement an open source algorithm that solves the problem. We hope that the code and ideas presented in this paper may lead to further developments towards creating a more practical and robust algorithm. It may be that hueristic or non-deterministic algorithms for solving the problem by

the 'direct approach' (Section 3) are more practical or performant than Grunewald's ring theoretical approach. This is an area of reasearch worth investigating.

Grunewald's algorithm is composed of many known number theory results. Its performance is highly dependant on the performance of its component parts. This implementation prioritised reducing development hours over algorithmic complexity, leaving significant opportunity for optimisation.

## References

[1] Max Dehn. Über unendliche diskontinuierliche gruppen. *Mathematische Annalen*, 71(1):116 – 144, March 1911.

[2] Mike Boyle and Benjamin Steinberg. Decidability of flow equivalence and isomorphism problems for graph C*-algebras and quiver representations. *arXiv e-prints*, page arXiv:1812.04555, Dec 2018.

[3] Bettina Eick, Tommy Hofmann, and E.A O'Brien. The conjugacy problem in $GL(n, \mathbb{Z})$. *arXiv e-prints*, page arXiv:1811.06190v2, May 2019.

[4] Fritz J. Grunewald. Solution of the conjugacy problem in certain arithmetic groups. In S.I. Adian, W.W. Boone, and G. Higman, editors, *WORD PROBLEMS II*, volume 95 of *Studies in Logic and the Foundations of Mathematics*, pages 101 – 139. Elsevier, 1980.

[5] Computational Algebra Group. Magma computational algebra system, 2020. `http://magma.maths.usyd.edu.au/magma/`.

[6] NumPy Developers. Numpy, 2019. `https://numpy.org/`.

[7] SymPy Development Team. Sympy, 2018. `https://www.sympy.org/`.

[8] James Hardy. Wilkinson and National Physical Laboratory (Great Britain). *Rounding errors in algebraic processes / by J.H. Wilkinson*. H.M.S.O London, 1963.

[9] Philip N. Klein, editor. *Beating Brute Force for Systems of Polynomial Equations over Finite Fields*. Society for Industrial and Applied Mathematics, January 2017.

[10] Xijin Tang and Yong Feng. A new efficient algorithm for solving systems of multivariate polynomial equations. *IACR Cryptology ePrint Archive*, 2005:312, 01 2005.

[11] Peter Petersen. *Linear Algebra*. UCLA, 2007.

# A Python Implementation

Listing 5: Python implentation of selected algorithms that were excluded from the main discussion

```
from sympy import *



class AlgorithmsCalculator:
        def __init__(self, integral_domain):
                self.domain = integral_domain

        def factorise_polynomial(self, polynomial):
        """
        This is algorithm is A0 in Grunewald's paper.

        It takes a polynomial with coefficients in an algebraic number field K,
        and factorises it into irreducible factors over K. Here K is whatever
        the domain of the calculator is.
        """
                return factor(polynomial, domain=self.domain)

        def is_coprime(self, polynomial_list):
        """
        Determines if a list of polynomials is coprime. Returns true if all
        polynomials are mutually coprime,
        false otherwise.
        """
                for i, x in enumerate(polynomial_list[:-1]):
                        for y in polynomial_list[i + 1:]:
                                divisor = gcd(x, y, domain=self.domain)
                                        if divisor != 1:
                                                return False
                return True

        def sub_matrix(self, polynomial, matrix):
        """
        Substitute a square matrix into a polynomial expression and evaluate
        it. Any constant terms are assumed to be multiplied by the identity
        matrix of the appropriate size
        """
                poly_as_matrix_expr = MatrixExpr(polynomial.all_terms.im_self.args[0])
```

```
38                poly_at_matrix = poly_as_matrix_expr.subs(polynomial.one.gen, matrix)
39
40                if poly_at_matrix.args[0].is_Matrix:
41                        return poly_at_matrix.args[0]
42
43                matrix_sum = zeros(matrix.cols, matrix.cols)
44
45                for arg in poly_at_matrix.args[0].args:
46                        matrix_sum = matrix_sum + arg * eye(matrix.cols)
47
48                return matrix_sum
49
50        def quotient_ring_of(self, polynomial):
51        """
52        Find the quotient ring of the ideal generated by a polynomial in one
53        variable.
54        """
55                variable = next(iter(polynomial.free_symbols))
56                return self.domain.old_poly_ring(variable).quotient_ring([polynomial])
57
58        def step_one(self, matrix1, matrix2):
59        """
60        This is step one of Grunewald's three-step procedure for solving the
61        integral conjugacy problem. This method accepts two square, rational
62        matrices of equal dimension with non-zero determinant. The result is
63        two pairs of semi simple and nilpotent matrices that satisfy the
64        following conditions.
65
66        step_one(matrix1, matrix2)
67        > [k*S1, k*U1, k*S2, k*U2]
68
69        matrix1 = S1 + U1
70        matrix2 = S2 + U2
71
72        S1*U1 = U1*S1
73        S2*U2 = U2*S2
74
75        k is the least scalar such that k*S1, k*U1, k*S2, k*U2 are integer
76        matrices.
77        """
```

```
78                  s1, u1 = self.jordan_chevalley(matrix1)
79                  s2, u2 = self.jordan_chevalley(matrix2)
80
81                  scalar = self.find_scalar_multiplier([s1, u1, s2, u2])
82
83                  return scalar * s1, scalar * u1, scalar * s2, scalar * u2
```

# B Automated Tests

Listing 6: Automated tests written to verify the behaviour of the algorithms

```
1   import unittest
2
3   from sympy import *
4
5   import algorithms
6
7   x = symbols('x')
8
9
10  class AlgorithmsSpec(unittest.TestCase):
11          # Set the domain to the rationals
12          calculator = algorithms.AlgorithmsCalculator(QQ)
13
14          def test_polynomial_factorisation(self):
15          """
16          Test that we can factorise a polynomial
17          """
18                  factors = self.calculator.factorise_polynomial(x ** 2 - 1)
19                  self.assertEqual(factors, (x - 1) * (x + 1))
20
21          def test_polynomial_factorisation_is_over_rationals(self):
22          """
23          Test that given a polynomial with some rational and some irrational
24          roots, it will only be factored over the rationals
25          """
26                  factors = self.calculator.factorise_polynomial(x ** 3 - 3 * x)
27                  self.assertEqual(factors, x * (x ** 2 - 3))
28
29          def test_coprime_finds_common_factors(self):
```

```
30          """
31          Test that, given a list of polynomials where two have a common factor,
32          the method returns that the list is not coprime
33          """
34
35                  # given: lists of polynomials with common factors
36                  list_one = [x ** 2 - 1, x ** 3 + 3, x + 1]
37                  list_two = [2, 2 * x ** 2, 5 * x - 3, 7 * x ** 2]
38                  list_three = [x, x ** 2 + 2 * x + 1, x ** 2 - 1]
39
40                  # expect: the method returns that they're not coprime
41                  self.assertFalse(self.calculator.is_coprime(list_one))
42                  self.assertFalse(self.calculator.is_coprime(list_two))
43                  self.assertFalse(self.calculator.is_coprime(list_three))
44
45          def test_coprime_works(self):
46          """
47          Test that, given a list of polynomials where none have a common factor,
48          the method returns that the list is coprime
49          """
50
51                  # given: lists of polynomials with no common factors
52                  list_one = [x ** 2 - 1, x ** 3 + 3, x + 3]
53                  list_two = [11, 2 * x ** 3 - 1, 5 * x - 3, 7 * x ** 2 + 13]
54                  list_three = [x, x ** 2 + 2 * x + 1, x ** 2 - 2]
55
56                  # expect: the method returns that they're coprime
57                  self.assertTrue(self.calculator.is_coprime(list_one))
58                  self.assertTrue(self.calculator.is_coprime(list_two))
59                  self.assertTrue(self.calculator.is_coprime(list_three))
60
61          def test_extended_euclidean_algorithm(self):
62          """
63          Test that, given a list of coprime polynomials p0, ... , pk, we can
64          find the polynomial multipliers that p0q0 + ... + pkqk = 1
65          """
66
67                  # given: the polynomial lists
68                  #    that we proved to be coprime in the above test
69                  list_one = [x ** 2 - 1, x ** 3 + 3, x + 3]
```

```
70                list_two = [11, 2 * x ** 3 - 1, 5 * x - 3, 7 * x ** 2 + 13]
71                list_three = [x, x ** 2 + 2 * x + 1, x ** 2 - 2]
72
73                # when: we calculate Bezout's
74                #    identity through repeated application of euclid's algorithm
75                list_one_cofactors = self.calculator.extended_euclidean_alg(list_one)
76                list_two_cofactors = self.calculator.extended_euclidean_alg(list_two)
77                list_three_cofactors = self.calculator.extended_euclidean_alg(list_three)
78
79                # expect: the method finds Bezout's identity
80                self.assertEqual(
81                str(list_one_cofactors), '[x**2/8 - 3*x/8 + 1/8, 3/8 - x/8, 0]'
82                )
83                self.assertEqual(str(list_two_cofactors), '[1/11, 0, 0, 0]')
84                self.assertEqual(str(list_three_cofactors), '[-x - 2, 1, 0]')
85
86        def test_minimal_polynomial_finder(self):
87        """
88        Test that given an element of GLn(Z), we
89        can compute the minimal polynomial
90        """
91                # given: an integer matrix
92                matrix = Matrix([[1, -1, -1], [1, -2, 1], [0, 1, -3]])
93
94                # when: we compute the minimal polynomial
95                polynomial = self.calculator.find_minimal_polynomial(matrix)
96
97                # then: we get the correct answer
98                expected = PurePoly(x ** 3 + 4 * x ** 2 + x - 1, x)
99                self.assertEqual(polynomial, expected)
100
101       def test_minimal_polynomial_complex_cases(self):
102       """
103       Given a matrix with a minimal polynomial not equal to the
104       characteristic polynomial, we can calculate the minimal polynomial
105       """
106               # given: integer matrices
107               matrix_one = Matrix(
108               [(0, 1, 0, 1), (1, 0, 1, 0), (0, 1, 0, 1), (1, 0, 1, 0)]
109               )
```

```
110                 matrix_two = Matrix([[0, -1, 1], [1, 2, -1], [1, 1, 0]])
111                 matrix_three = Matrix(
112                 [(1, 0, 0, 1), (0, 1, 1, 0), (0, 0, 0, 1), (0, 0, -2, 3)]
113                 )
114                 matrix_four = zeros(5, 5)
115                 matrix_five = eye(5)
116
117                 # when: we compute the minimal polynomials
118                 polynomial_one = self.calculator.find_minimal_polynomial(matrix_one)
119                 polynomial_two = self.calculator.find_minimal_polynomial(matrix_two)
120                 polynomial_three = self.calculator.find_minimal_polynomial(matrix_three)
121                 polynomial_four = self.calculator.find_minimal_polynomial(matrix_four)
122                 polynomial_five = self.calculator.find_minimal_polynomial(matrix_five)
123
124                 # then: we get the correct answer
125                 expected_one = PurePoly(x ** 3 - 4 * x, x)
126                 expected_two = PurePoly(x ** 2 - x, x)
127                 expected_three = PurePoly(x ** 3 - 4 * x ** 2 + 5 * x - 2, x)
128                 expected_four = PurePoly(x, x)
129                 expected_five = PurePoly(x - 1, x)
130
131                 self.assertEqual(polynomial_one, expected_one)
132                 self.assertEqual(polynomial_two, expected_two)
133                 self.assertEqual(polynomial_three, expected_three)
134                 self.assertEqual(polynomial_four, expected_four)
135                 self.assertEqual(polynomial_five, expected_five)
136
137         def test_compute_trial_indices(self):
138         """
139         Test that we can construct a list of all the possible indices that the
140         minimal polynomial can take
141         """
142                 # given: a list of maximum indices
143                 max_ind = [1, 2, 3]
144
145                 # when: we compute all permutations of possible minimal polynomial indices
146                 solution = self.calculator.compute_trial_indices(max_ind)
147
148                 # then: the list is return, properly
149                 #    ordered from minimal to maximal (sum of indices)
```

```
150                 expected_solution = [
151                 (1, 1, 1),
152                 (1, 1, 2),
153                 (1, 2, 1),
154                 (1, 1, 3),
155                 (1, 2, 2),
156                 (1, 2, 3),
157                 (1, 3, 3)
158                 ]
159                 index = 0
160                 for indices in solution:
161                         self.assertEqual(indices, expected_solution[index])
162                         index = index + 1
163
164         def test_compute_trial_indices_again(self):
165         """
166         Test that we can construct a list of all the possible indices that the
167         minimal polynomial can take
168         """
169                 # given: a list of maximum indices
170                 max_ind = [1, 1, 2]
171
172                 # when: we compute all permutations of possible minimal polynomial indices
173                 solution = self.calculator.compute_trial_indices(max_ind)
174
175                 # then: the list is return, properly ordered
176                 #   from minimal to maximal (sum of indices)
177                 expected_solution = [(1, 1, 1), (1, 1, 2)]
178                 index = 0
179                 for indices in solution:
180                 self.assertEqual(indices, expected_solution[index])
181                 index = index + 1
182
183         def test_sub_matrix(self):
184         """
185         Test that we can substitute a matrix into various polynomial expressions
186         """
187                 # given: a matrix and several polynomial expressions
188                 matrix = Matrix([[0, -1, 1], [1, 2, -1], [1, 1, 0]])
189
```

```
190                 poly_one = PurePoly(1 + x, x)
191                 poly_two = PurePoly(1 - 3 * x + x ** 2, x)
192                 poly_three = PurePoly(-4 + x ** 3, x)
193
194                 # when: we substitute in the matrix
195                 sol_one = self.calculator.sub_matrix(poly_one, matrix)
196                 sol_two = self.calculator.sub_matrix(poly_two, matrix)
197                 sol_three = self.calculator.sub_matrix(poly_three, matrix)
198
199                 # then: the expressions are evaluated and simplified
200                 self.assertEqual(sol_one, Matrix([[1, -1, 1], [1, 3, -1], [1, 1, 1]]))
201                 self.assertEqual(sol_two, Matrix([[1, 2, -2], [-2, -3, 2], [-2, -2, 1]]))
202                 self.assertEqual(sol_three, Matrix([[-4, -1, 1], [1, -2, -1], [1, 1, -4]]))
203
204         def test_jordan_chevalley(self):
205         """
206         Test that we can find the Jordan-Chevalley decomposition of a matrix
207         """
208                 # given: a square matrix
209                 matrix = Matrix(
210                 [[5, 4, 2, 1], [0, 1, -1, -1], [-1, -1, 3, 0], [1, 1, -1, 2]]
211                 )
212                 matrix2 = Matrix([[0.5, 1], [1, 0.5]])
213
214                 # when: we compute the jordan-chevalley decomposition
215                 semisimple, nilpotent = self.calculator.jordan_chevalley(matrix)
216                 semisimple2, nilpotent2 = self.calculator.jordan_chevalley(matrix2)
217
218                 # then: the solution is correct
219                 self.assertEqual(semisimple + nilpotent, matrix)
220                 self.assertEqual(semisimple * nilpotent, nilpotent * semisimple)
221                 self.assertTrue(nilpotent.is_nilpotent())
222
223                 self.assertEqual(semisimple2 + nilpotent2, matrix2)
224                 self.assertEqual(semisimple2 * nilpotent2, nilpotent2 * semisimple2)
225                 self.assertTrue(nilpotent2.is_nilpotent())
226
227         def test_find_denominator(self):
228         """
229         Test that we can find the denominator of a simplified fraction, given a
```

```
230         float
231         """
232                 # given: floating point numbers and a fraction
233                 float1 = 0.17
234                 float2 = 1.8342
235                 fraction1 = Rational(1, 3)
236 
237                 # when: calculating the denominator of the simplified fraction
238                 denom1 = self.calculator.find_denominator(float1)
239                 denom2 = self.calculator.find_denominator(float2)
240                 denom3 = self.calculator.find_denominator(fraction1)
241 
242                 # then: expect the correct answer
243                 self.assertTrue(fraction1.is_rational)
244                 self.assertEqual(denom1, 100)
245                 self.assertEqual(denom2, 5000)
246                 self.assertEqual(denom3, 3)
247 
248         def test_finding_scalar_multiplier(self):
249         """
250         Test that we can find the least scalar multiplier that will make a
251         rational matrix an integer matrix
252         """
253                 # given: matrices
254                 matrix1 = Matrix([[1, 2, 3], [0, 0, 1], [1, 1, 1]])
255                 matrix2 = Matrix([[0.6, 0.5], [0.17, 1.8342]])
256 
257                 # when: calculating the least multiple to create an integer matrix
258                 scalar1 = self.calculator.find_scalar_multiplier([matrix1])
259                 scalar2 = self.calculator.find_scalar_multiplier([matrix2])
260                 scalar3 = self.calculator.find_scalar_multiplier([matrix1, matrix2])
261 
262                 # then: the scalars are correctly calculated
263                 self.assertEqual(scalar1, 1)
264                 self.assertEqual(scalar2, 5000)
265                 self.assertEqual(scalar3, 5000)
266 
267         def test_step_one(self):
268                 """
269                 Test the first step of Grunewald's algorithm
```

```
270                 """
271                 # given: two conjugating matrices
272                 matrix1 = Matrix([[0.5, 1], [1, 0.5]])
273                 matrix2 = Matrix([[-3.5, 3], [-5, 4.5]])
274
275                 # when: applying step one of the algorithm
276                 s1, u1, s2, u2 = self.calculator.step_one(matrix1, matrix2)
277
278                 # then: the solution is correct
279                 self.assertEqual(2 * matrix1, s1)
280                 self.assertEqual(2 * matrix2, s2)
281                 self.assertEqual(zeros(2), u1)
282                 self.assertEqual(zeros(2), u2)
283
284         def test_find_quotient_ring(self):
285                 """
286                 Test that we can find the quotient ring of an ideal
287                 """
288                 # given: a polynomial
289                 polynomial = x ** 2
290
291                 # when: finding the quotient ring of the ideal
292                 quotient_ring = self.calculator.quotient_ring_of(polynomial)
293
294                 # then: the quotient ring is correctly calculated
295                 self.assertTrue(isinstance(quotient_ring, polys.domains.quotientring.QuotientRing))
296                 self.assertEqual(quotient_ring, QQ.old_poly_ring(x)/[x**2])
297
298
299         if __name__ == '__main__':
300                 unittest.main()
```