Московский Государственный Университет
имени М.В. Ломоносова

Механико-математический факультет
Кафедра высшей алгебры

Курсовая работа

По теме

Классификация автоморфизмов групп Шевалле над локальными кольцами с необратимой двойкой

Выполнил: Меденцов Н.В., гр. 502
Научный руководитель:
проф. д.ф-м.н. Бунина Е.И.

г. Москва, 2020 г.

1 Ход работы

1.1 Abstract

Цель нашей работы - классифицировать все автоморфизмы групп Шевалле типа C_l над локальными кольцами с необратимой двойкой. Сопрягая произвольный автоморфизм некоторыми другими, известными, мы надеемся получить в итоге его представление в виде композиции кольцевых, центральных, внутренних, диаграммных автоморфизмов и специальных «автоморфизмов сопряжения».

1.2 Текущий результат

На данный момент

- Теоретически описан ход работы
- Найден конструктивный способ построения знаков элементов матрицы присоединённого представления баисных элементов алгебры Ли.
- Реализован код по построению матриц $x_\alpha(t)$ для $\alpha \in C_{n\geq 3}$ с учётом правильного выбора знаков, произведено построение.
- Конструктивно проверено коммутирование матриц Q_1 и Q_3 и соотношение $Q^3_{i\in\{1,3\}} = E$

1.3 Следующие шаги

Для получения результата в дальнейшем необходимо:

- Уточнить последний пункт построения изоморфизма.
- Методом линеаризации показать, что A_i из $\varphi''(w_{\alpha_i}) = \dots$ равно 0.
- Методом линеаризации показать, что T' из $\varphi''(x_\alpha(t)) = \dots$ равно 0.

1.4 Теоретические выкладки

Пусть $\varphi : E(R) \to E(R)$ — автоморфизм $G(R)$, группы Шевалле типа C_n над локальным кольцом с необратимой двойкой. Так как R - локальное кольцо, по известной теореме гуппа Шевалле совпадает со своей элементарной подгуппой, поэтому будем рассматривать именно элементарную группу Шевалле. Пока что будем работать с системой корней C_3.

Пусть J - максимальный радикал кольца R, тогда $k := R/J$ - поле, значит, $E_J := E(J)$ является наибольшей нормальной собственной подгруппой. E_J инвариантна относительно

φ, перейдём к индуцированному автоморфизму (объяснение равенства) $\bar{\varphi} : E_{ad}(\Phi, R)/E(J) = E_{ad}(\Phi, k) \to E_{ad}(\Phi, k)$.

Используя факт делаем вывод, что $\bar{\varphi}$ — *стандартный* автоморфизм, то есть, $\bar{\varphi} = i_{\bar{g}}\bar{\rho}, \bar{g} \in N(E_{ad}(\Phi, k)), N$ — нормализатор, ρ - кольцевой автоморфизм k. Заметим, что $\exists g \in GL_n(R)$, такое что после факторизации мы попадаем в класс эквивалентности $\bar{g}$, однако не факт, что $g \in N(E_{ad})$ (если бы это было так, мы могли бы сразу получить представление φ в виде композиции кольцевого и внутреннего, но нет).

Рассмотрим $\varphi' := i_{g^1}\varphi : E_{ad}(\Phi, R) \to GL_n(R)$, такой что его образ при факторизации по $E(J)$ совпадает с $\bar{\rho}$. Пусть $A \in E_{ad}(\Phi, R)$, причём все элементы A — из $R' = \langle 1_R \rangle$. Тогда $B = \varphi'(A) \in GL_n(R')$, $A - B \in M_n(J)$.

Выделим некоторые элементы нашей группы Шевалле, которые понадобятся нам в дальнейшем:

$$w_\alpha(t) := x_\alpha(t)x_{-\alpha}(-t^{-1})x_\alpha(t), \quad t \in R^*$$

$$w_{\alpha_i} := w_{\alpha_i}(1) = x_{\alpha_i}(1)x_{-\alpha_i}(-1)x_{\alpha_i}(1)$$

$$Q_i := w_{\alpha_i}x_{\alpha_i}(1)$$

$$h_\alpha(t) = w_\alpha(t)w_\alpha(1)^{-1}$$

Путём тривиальной проверки получаем, что $Q_i^3 = E$, а Q_1 и Q_3 коммутируют. Некоторыми рассуждениями с присоединением корня третьей степени из единицы получаем, что сопряжением можно перейти от φ' к φ'', сохраняющему Q_1, Q_3. Вспоминая свойства $\bar{\varphi}$, видим, что отсюда очевидно следует, что $\varphi''(w_{\alpha_{i\in\{1,3\}}}) = w_{\alpha_i} + A_i$, где $A_i \in M_n(J)$. Теперь мы хотим показать, что $A_i = 0$, тогда получится, что w_{α_i} и $x_{\alpha_i}(1)$ переходят в себя.

Чтобы показать, что $A_i = 0$, составим систему равенств, вытекающих из условий на коммутаторы x_{α_i}. Уравнения данной системы будут справедливы и после применения к ним изоморфизма φ''. Вычитая из второй системы равенств соответствующие равенства из первой, получаем систему равенств, где в качестве переменных выступают A_i. Мы хотим показать, что единственным её решением является $A_i = 0, \forall i$. Для этого мы воспользуемся методом линеаризации и соответствующей теоремой.

Теперь наша цель - показать, что $\varphi''(x_\alpha(t)) = x_\alpha(\bar{\rho}(t))$, $\bar{\rho}$ — некоторое отображение. Для начала заметим, что имея, что φ'' сохраняет w_{α_i} и $x_{\alpha_i}(1)$, и используя соотношение для уже известных комбинаций корней (R7), мы можем путём представления других x_α через имеющиеся получить, что $\varphi''(x_\alpha) = x_\alpha, \forall\alpha$. Известно, что $\varphi''(x_\alpha(t)) = x_\alpha(\bar{\rho}(t))$ верно над полем. Это означает, что в нашем случае $\varphi''(x_\alpha(t)) = x_\alpha(t') + T'$, где $T' \in M_n(J)$. Достаточно показать, что $T' = 0$ только для $x_{\alpha_{\pm i}}(t)$, где α_i - простой корень (далее мы сможем выразить остальные $x_\alpha(t)$ через уже имеющиеся способом, аналогичным описанному выше). Чтобы показать, что $T' = 0$, нужно снова воспользоваться методом линеаризации.

Для завершения доказательства нужно аккуратно провести рассуждения, показывающие, что $\bar{\rho}$ - не просто отображение $R^* \to R^*$, но кольцевой автоморфизм на R.

2 Используемый код

Для проверки результатов можно использовать следующие действия:

Листинг 1: Chevalley.py

```
E = Chevalley()
# input data accorting to the tips on the screen
3  # The dimention of the vector space
0  # Tag (auxiliary)
1  # Verbosity
E.x(1), E.w(1), E.Q(1)  # Will give x, w and Q elements
E.check_Q_commuting()  # Checks whether Q(1) and Q(3) commute. Must be True.
E.check_Q_cubed(1)  # Checks whether Q(i) ^ 3 == I. Must be true.
```

Используемый код.

Листинг 2: Chevalley.py

```
import numpy as np
from collections import defaultdict
from sympy import *

class Chevalley:
  def __init__(self):
    self.dimention_of_vector_space,\
      self.auxiliary_tag,\
      self.verbosity,\
      self.matrix_dimention = 0, 0, 0, 0
    self.elements = []

    # Initializing
    # t = Symbol('t')
    self.DataInput()
    self.matrix_dimention =\
      2 * self.dimention_of_vector_space ** 2 + self.dimention_of_vector_space

    # Filling "elements"
    self.GenerateAllElements()

    # if check_Q_commuting():
    #    print("\nQ elements are commuting\n")
    # else:
    #    print(f"\nQ elements are NOT commuting:\nQ_1 =\n{Q(1)}\nQ_3={Q(3)}\n\n")

    # print(f"\nQ cubed:\nQ_1 ^ 3 =\n{Q(1) ** 3}\nQ_3 ^ 3 ={Q(3) ** 3}\n\n")



  def GenerateAllElements(self):
    """
    Filling the first 'self.dimention_of_vector_space' cells of 'elements' with the
    following basic elements:
    | alpha_1      = e_1 - e_2      ->  elements[0]
    | alpha_2      = e_2 - e_3      ->  elements[1]
```

```python
    | ...                    ...
    | alpha_(l-1)    = e_l - e_(l-1)    ->  elements[l - 2]
    | alpha_l       = 2 * e_l        ->  elements[l - 1] ,
    denoting self.dimention_of_vector_space as l.
    ...

    In this case the order is just a lexicographical order.
    """

    list_template = [0] * self.dimention_of_vector_space

    # Adding basic elements as described
    basis = []
    for _ in range(self.dimention_of_vector_space - 1):
      basis.append(np.array(list(list_template)))
      basis[_][_], basis[_][_ + 1] = 1, -1
    basis.append(np.array(list(list_template)))
    basis[self.dimention_of_vector_space - 1][-1] = 2
    self.elements.extend(basis)

    # Adding (1, 1, 0); (1, 0, 1); (0, 1, 1)
    for i in range(self.dimention_of_vector_space - 1):
      for j in range(self.dimention_of_vector_space)[i + 1:]:
        self.elements.append(np.array(list(list_template)))
        self.elements[-1][i], self.elements[-1][j] = 1, 1

    # Adding non-basic (1, 0, -1)
    for i in range(self.dimention_of_vector_space - 2):
      for j in range(self.dimention_of_vector_space)[i + 2:]:
        self.elements.append(np.array(list(list_template)))
        self.elements[-1][i], self.elements[-1][j] = 1, -1

    # Adding (2, 0, 0); (0, 2, 0);
    for i in range(self.dimention_of_vector_space - 1):
      self.elements.append(np.array(list(list_template)))
      self.elements[-1][i] = 2

    # Adding all opposite elements
    for i in range(self.dimention_of_vector_space ** 2):
      self.elements.append(np.array(self.elements[i] * -1))


  def IsPositive(self, vec):
    zero = 1e-14
    for coo in vec:
      if coo > zero:
        return True
      if coo < -zero:
        return False


  def GetCorrectAbsValue(self, a, b):
    r = 0
    while (any(not any(b - a * r - _) for _ in self.elements)):
      r += 1
    r -= 1  # that's the true r indeed
```

```
  c_ab = r + 1
  return c_ab


def GetCorrectSign(self, a, b, extraspecial_pairs_signs):
  # Learning whether (a, b) is an special pair
  if self.IsPositive(a) and self.IsPositive(b - a):
    # Finding an exstraspecial pair for this one
    a_minimal, ksi = a, a + b
    for root in self.elements:
      if any([all(cur) for cur in self.elements == ksi - root]) and\
          self.IsPositive(root) and\
          self.IsPositive(ksi - root):
        if self.IsPositive(a_minimal - root):
          a_minimal = root
    # The extrasp. pair we were searching for
    a_minimal, b_minimal = a_minimal, ksi - a_minimal

    t1, t2 = 0, 0
    # The first condition looks suspicious and asymmetric. E.g., why it can be (can
    # it?) negative? Check it once more in the article.
    if any([all(cur) for cur in self.elements == b - a_minimal]):
      t1 = self.GetCorrectAbsValue(a_minimal, b - a_minimal) *\
          self.GetCorrectAbsValue(a, b_minimal - a) *\
          extraspecial_pairs_signs[(tuple(a_minimal), tuple(b - a_minimal))] *\
          extraspecial_pairs_signs[(tuple(a), tuple(b_minimal - a))] *\
          np.square(b - a_minimal).sum() /\
          np.square(b).sum()
    if any([all(cur) for cur in self.elements == a - a_minimal]):
      t2 = self.GetCorrectAbsValue(a_minimal, a - a_minimal) *\
          self.GetCorrectAbsValue(b, b_minimal - b) *\
          extraspecial_pairs_signs[(tuple(a_minimal), tuple(a - a_minimal))] *\
          extraspecial_pairs_signs[(tuple(b), tuple(b_minimal - b))] *\
          np.square(a - a_minimal).sum() /\
          np.square(a).sum()

    return np.sign(t1 - t2)
  else:  # The pair is not special
    m = 1
    if self.verbosity: print(f'Method GetCorrectSign START')
    if self.verbosity: print(f'Method GetCorrectSign. a = {a}, b = {b}, m = {m}')
    if not self.IsPositive(b):
      if self.verbosity: print(f'We say that b < 0')
      a, b, m = -a, -b, -m
      if self.verbosity: print(f'Method GetCorrectSign. a = {a}, b = {b}, m = {m}')
    if not self.IsPositive(a):
      if self.verbosity: print(f'We say that a < 0')
      if self.IsPositive(b + a):
        if self.verbosity: print(f'We say that -a < b')
        a, b = a + b, -a
        if self.verbosity: print(f'Method GetCorrectSign. a = {a}, b = {b}, m = {m}
      else:
        if self.verbosity: print(f'We say that -a >= b')
        a, b = b, -a - b
        if self.verbosity: print(f'Method GetCorrectSign. a = {a}, b = {b}, m = {m}
    if self.IsPositive(a - b):
```

```python
            if self.verbosity: print(f'We say that a > b')
            a, b, m = b, a, -m
            if self.verbosity: print(f'Method GetCorrectSign. a = {a}, b = {b}, m = {m}')
        if self.verbosity: print(f'Method GetCorrectSign FINISH\n')
        return m * self.GetCorrectSign(a, b, extraspecial_pairs_signs)


    def BuildTheBasicElementRepresentationMatrix(self, basic_element_index, matrix):
        # It can be changed arbitrary
        extraspecial_pairs_signs = defaultdict(lambda: 1)
        basic_element_index -= 1  # since we have 0-indexation

        for column in range(self.matrix_dimention):

            # [x_a, h_i] = -alpha(h_i) * x_a, where
            # alpha(h_i) = <x_i, x_a> = 2 * (x_i, x_a) / (x_a, x_a) --- Seems to be wrong!
            #
            # My new variant
            # If we work with the i-th root (i := basic_element_index)
            # [x_{a_i}, h_j] = - <a_i, a_j> * x_{a_i}
            # <a_i, a_j> = 2 (a_i, a_j) / (a_j, a_j)

            # This condition gives us 'elements[column] == h_j'
            if (self.matrix_dimention - self.dimention_of_vector_space <=\
                column < self.matrix_dimention):
                # Fixed variant
                h = self.elements[column - (self.matrix_dimention - self.dimention_of_vector_
                matrix[basic_element_index][column] =\
                  -2 * h.dot(self.elements[basic_element_index]) / h.dot(h)
                continue

            # [x_a, x_b] = e_ab * c_ab * x_(a + b),    where a + b from Φ,    e_ab ? {1, -1}
            if (any(cur.all() for cur in self.elements == self.elements[column] +\
                                     self.elements[basic_element_index]) and\
                column != basic_element_index):
                # temporary renaming
                a, b = self.elements[basic_element_index], self.elements[column]

                # to find the row in the matrix to insert the result (c_ab)
                a_plus_b_element_index = 0
                for _ in self.elements:
                    if (not any(_ - (a + b))):
                        break
                    a_plus_b_element_index += 1

                # b - ra, ..., b + qa - a-series of the root b

                # In fact, these formulee are equivalent. It's stated in Humphreys

                matrix[a_plus_b_element_index][column] = self.GetCorrectAbsValue(a, b) *\
                                       self.GetCorrectSign(a, b, extraspecial_pairs_signs)
                continue

            # [x_a, x_(-a)] = h_i (correspondant to a)
            if (not any(self.elements[column] + self.elements[basic_element_index])):
                if basic_element_index < self.dimention_of_vector_space ** 2:
```

```
            matrix[self.matrix_dimention - self.dimention_of_vector_space +\
                    + basic_element_index][column] = 1
          else:
            delta = -(self.dimention_of_vector_space ** 2)
            matrix[self.matrix_dimention - self.dimention_of_vector_space +\
                    + delta + basic_element_index][column] = -1
          continue


    def bmatrix(self, a):
      """Returns a LaTeX bmatrix

      :a: numpy array
      :returns: LaTeX bmatrix as a string
      """
      if len(a.shape) > 2:
        raise ValueError('bmatrix can at most display two dimensions')
      lines = str(a).replace('\n', '')\
              .replace('[[', '[')\
              .replace('[', '\n')\
              .replace(']', '').splitlines()[1:]
      rv = [r'\[ \begin{bmatrix}']
      rv += ['  ' + ' & '.join(l.split()) + r'\\' for l in lines]
      rv +=  [r'\end{bmatrix} \]']
      return '\n'.join(rv)


    def DataInput(self):
      self.dimention_of_vector_space = int(input("Dimention of vector space: "))
      self.auxiliary_tag = input("Tag (can be empty): ")
      self.verbosity = int(input("Verbosity (1 or 0): "))
      return 0


    def x(self, idx, t = 1):
      resulring_matrix = np.diag([1.] * self.matrix_dimention)

      basic_element_representation_matrix = np.zeros(self.matrix_dimention ** 2).\
                                            reshape(self.matrix_dimention, -1)

      # Filling "basic_element_representation_matrix"
      self.BuildTheBasicElementRepresentationMatrix(idx,\
                                 basic_element_representation_matrix)

      # Calculating the result
      matrix_powered = np.array(basic_element_representation_matrix)
      power, factorial = 1, 1
      while(np.any(matrix_powered != 0)):
        if self.verbosity:
          print("power =", power, ", matrix:\n", self.bmatrix(matrix_powered))
        for i in range(self.matrix_dimention):
          resulring_matrix[i] += (t ** power) * matrix_powered[i]
        power += 1
        factorial *= power
        matrix_powered =\
          matrix_powered.dot(basic_element_representation_matrix) / power
```

```
                # matrix_powered.dot(basic_element_representation_matrix) / factorial
# WRONG!

            if self.verbosity:
                print("The result is:\n", self.bmatrix(resulring_matrix))

            return resulring_matrix


        def w(self, i, t = 1):
            # x(i + self.dimention_of_vector_space ** 2) corresponds to the opposite root
            return self.x(i, 1) * self.x(i + self.dimention_of_vector_space ** 2, -1) *\
                                                                    self.x(i, 1)
            pass


        def Q(self, i):
            return self.w(i) * self.x(i)


        ##### The following functions are for manual checking #####


        def check_R1(self, i, t=1, u=1):
            pass


        def check_R2(self, i, j, t=1, u=1):
            pass


        def check_R6(self, i, j, t=1):
            pass


        def check_R7(self, i, j, t=1):
            pass


        def check_R8(self, i, j, t=1, u=1):
            pass


        def check_Q_commuting(self):
            Q = self.Q
            return True if np.all(Q(1) * Q(3) == Q(3) * Q(1)) else False


        def check_Q_cubed(self, i):
            Q = self.Q(i)
            return np.all(Q ** 3 == np.eye(Q.shape[0]))


        ############################################################
```

```
if __name__ == '__main__':
    Chevalley()
```