Содержание

1 Введение **3**

2 Цели и задачи **5**

3 Определения и понятия **6**

3.1 Элементарные нелинейные функции 6

3.2 Системы элементарных ОДУ 6

4 Полиномиализация **8**

4.1 Полиномиализация с помощью введения алгебраических уравнений 8

4.2 Полиномиализация с помощью введения дифференциальных уравнений 9

4.3 Анализ оригинальных алгоритмов 10

4.4 Абстрактные синтаксические деревья 11

4.5 Алгоритм полиномиализации 12

4.5.1 Прямой обход 14

4.5.2 Обратный обход 15

4.5.3 Сравнение 15

5 Квадратизация **17**

5.1 Квадратизация с помощью введения алгебраических уравнений 17

5.2 Квадратизация с помощью введения дифференциальных уравнений 18

5.3 Анализ алгоритмов квадратизации 19

5.4 Генерация замен переменных 20

5.5 Граф замен 21

5.6 Алгоритмы поиска на графе 22

5.6.1 BFS и DFS 22

5.6.2 Поиск с ограничением глубины (DLS) 22

5.6.3 Поиск в глубину с итеративным углублением (ID-DFS) . 23

5.6.4 Поиск с ограничением глубины и итеративным углублением (ID-DLS) 24

5.7 Эвристический подход к поиску 26

5.8 Эвристики для квадратизации 26
5.9 Правило параллелограмма 28

6 Эксперименты **30**

7 Заключение **31**

1 Введение

Динамические системы являются неотъемлемой частью современной прикладной математики. С помощью динамических систем описываются явления в таких областях как физика [5], химия [6], биология [7], экономика [8] и многих других. Математические методы позволяют проводить с ними такие операции как упрощение [9] и анализировать такие их свойства как устойчивость [10] и достижимость [11]. Многие динамические системы, встречающиеся на практике, описываются с помощью нелинейных дифференциальных уравнений. Анализ таких систем является молодой областью со множеством открытых вопросов, в отличие от анализа линейных систем [12]. Поэтому не удивительно, что многие известные методы для работы с нелинейными системами полагаются на аппроксимацию нелинейных элементов линейными. К сожалению, такой подход часто ведёт к неудовлетворительным результатам. Поэтому большой интерес заслужили подходы, целиком полагающиеся на нелинейную природу систем, которыми они оперируют. Отдельно выделим метод понижения порядка моделей QLMOR [1, 2], который показывает state-of-the-art результаты, сохраняя при этом динамику исходной системы. Данный метод полагается на процесс квадратизации — приведения систем нелинейных дифференциальных уравнений к системе полиномиальных дифференциальных уравнений степени не более двух. Примером квадратизации может послужить следующее преобразование (формальное определение и подробные примеры приведены в главах 3, 4 и 5):

$$\dot{x} = \frac{1}{1+e^x} \Longrightarrow \begin{cases} \dot{x} = y_1 \\ \dot{y}_0 = y_0 y_1 \\ \dot{y}_1 = -y_1 y_2 \\ \dot{y}_2 = y_1 y_2 - 2y_2^2 \end{cases}$$

На данный момент не существует достаточно эффективных или хотя бы реализованных алгоритмов квадратизации — она всегда совершается вручную. Из-за этого, даже для сравнительно небольших систем уравнений проводить её крайне долго и сложно. Более того, нет гарантий, что полученная квадратизация имеет наименьшую возможную размерность, что критично в задаче понижения порядка моделей.

В данной работе мы предлагаем алгоритм квадратизации, описываем его реализацию и модификации, и демонстрируем работу реализации на примерах систем из литературы.

2 Цели и задачи

Цель данной работы — предложить и реализовать алгоритм нахождения оптимальной квадратизации систем нелинейных дифференциальных уравнений. Эта задача разбивается на следующие подзадачи:

1. Формализовать алгоритм квадратизации.
2. Предложить алгоритмы поиска оптимальной квадратизации.
3. Реализовать предложенные алгоритмы внутри системе компьютерной алгебры.
4. Провести эксперименты, чтобы сравнить работу предложенных алгоритмов.

3 Определения и понятия

3.1 Элементарные нелинейные функции

Определение 3.1. Будем понимать под функциями $f_i(\vec{x})$ *элементарные нелинейные функции*, которые могут быть записаны как линейные комбинации элементарных функций $g_k(\vec{x})$.

$$f_i(\vec{x}) = p_i^T \vec{x} + a_{i,1} g_1(\vec{x}) + \cdots + a_{i,m} g_m(\vec{x}), p_i \in \mathbb{R}^n, a_j \in \mathbb{R} \tag{1}$$

Такими функция мы можем покрыть широкий класс проблем, встречающихся на практике. Мы требуем элементарность функций, так как производная элементарной функции может быть найдена за конечное число шагов, что будет критично для нас в дальнейшем.

3.2 Системы элементарных ОДУ

В данной работе мы рассматриваем системы нелинейных ОДУ следующего вида:

1. **Система элементарных ОДУ в нормальной форме**

$$\dot{x_i} = f_i(\vec{x}), \quad i = 1 \cdots N \tag{2}$$

   В теории управления данная форма описывает уравнения состояния для автономных систем.

2. **Система элементарных уравнений и элементарных ОДУ в нормальной форме**

$$\begin{aligned} \dot{x_i} &= f_i(\vec{x}), \quad i = 1 \cdots n \\ f_j(\vec{x}) &= 0, \quad j = 1 \cdots m \end{aligned}, \tag{3}$$

Как легко заметить, система (2) является частным случаем системы (3).

Определение 3.2. Рассмотрим частный случай систем (2) и (3), где функции f_i представляют собой полиномы. Такие системы мы будем называть *полиномиальными*.

Определение 3.3. Полиномиальная система имеет порядок M, если M - наивысшая степень мономов, образованных переменными x_i.

Определение 3.4. Полиномиальные системы порядка 2 назовём *квадратичными.*

4 Полиномиализация

Определение 4.1. Процесс преобразования нелинейных систем (2), (3) к полиномиальному виду будем называть *полиномиализацией*.

Далее мы рассмотрим оригинальные алгоритмы квадратизации, описанные в [1].

4.1 Полиномиализация с помощью введения алгебраических уравнений

Следующий алгоритм проводит полиномиализацию, добавляя к системе алгебраические уравнения:

1. Ввести новую переменную $y_i = g_i(\vec{x})$, где $g_i(\vec{x})$ - неполиномиальная нелинейная элементарная функция.
2. Заменить $g_i(\vec{x})$ на y_i в оригинальном уравнении.
3. Добавить новое уравнение $y_i = g_i(\vec{x})$ в систему.

Показанный алгоритм приводит системы вида (2), (3) к полиномиальным системам вида (3)

Пример. Рассмотрим уравнение $\dot{x} = x^3 + \frac{1}{1+x}$. Тогда шаги алгоритма будут выглядеть следующим образом:

1. Вводим новую переменную $y = \frac{1}{1+x}$
2. Делаем замену в первом уравнении $\dot{x} = x^3 + y$
3. Добавляем новое уравнение в систему $y = \frac{1}{1+x} \Leftrightarrow xy + y - 1 = 0$

Получили полиномиальную систему
$\dot{x} = x^3 + y$
$0 = xy + y - 1$

Примечание. Важно заметить, что данный подход работает для ограниченного класса элементарных нелинейных функций, в частности для рациональных функций. Таким образом мы гарантируем полиномиальность вспомогательных уравнений. В том случае, если мы имеем дело со степенными функциями, логарифмами и многими другими элементарными нелинейностями,

например $g_i(x) = e^x$, то получить полиномиальную систему данным методом мы уже не сможем.

4.2 Полиномиализация с помощью введения дифференциальных уравнений

Второй подход к полиномиализации предлагает добавление дифференциальных уравнений к системе. Алгоритм похож на предыдущий за исключением последнего шага:

1. Ввести новую переменную $y_i = g_i(\vec{x})$, где $g_i(\vec{x})$ - неполиномиальная нелинейная элементарная функция.
2. Заменить $g_i(\vec{x})$ на y_i в оригинальном уравнении.
3. Добавить новое уравнение $\dot{y}_i = \dot{g}_i(\vec{x}) = g_i'(\vec{x})\dot{\vec{x}}$ в систему. Данное уравнение получается как сложная производная от g_i.

Таким образом, мы получим систему общего вида

$$\begin{aligned} \dot{x}_i &= p_i^T\vec{x} + a_{i,1}y_1 + \cdots + a_{i,m}y_m, \quad i = 1, \cdots, n \\ \dot{y}_i &= \pounds_{\dot{\vec{x}}} g_i(\vec{x}) = g_i'(\vec{x})(p_i^T\vec{x} + a_{i,1}y_1 + \cdots + a_{i,m}y_m,), \quad i = 1, \cdots, m \end{aligned}, \tag{4}$$

где $g'(\vec{x}) = \frac{dg(\vec{x})}{d\vec{x}}$, $\pounds_{\dot{\vec{x}}}$ - производная Ли.

Так как $g'(\vec{x})$ состоит только из полиномиальных функций от x и y_i, данная система является полиномиальной.

Данный алгоритм переводит системы к полиномиальному виду, не меняя их структуры: (2) $\longrightarrow$ (2), (3) $\longrightarrow$ (3).

Пример. Полиномиализуем уравнение $\dot{x} = \sin(x)$. Тогда шаги алгоритма будут выглядеть следующим образом:

1. Избавляемся от $\sin(x)$
   (a) Вводим новую переменную $y_1 = \sin(x)$
   (b) Делаем замену в первом уравнении $\dot{x} = y$

(c) Добавляем новое уравнение в систему $\dot{y}_1 = (\sin(x))^{\cdot} = \cos(x)\dot{x} = \sin(x)\cos(x) = y_1\cos(x)$

(d) Получили систему $\begin{cases} \dot{x} = y_1 \\ \dot{y}_1 = y_1\cos(x) \end{cases}$

2. Избавляемся от $\cos(x)$

(a) Вводим новую переменную $y_2 = \cos(x)$

(b) Делаем замену во втором уравнении $\dot{y}_1 = y_1 y_2$

(c) Добавляем новое уравнение в систему $\dot{y}_2 = \dot{\cos}(x) = -\sin(x)\dot{x} = -y_1^2$

Получили полиномиальную систему

$\dot{x} = y_1$

$\dot{y}_1 = y_1 y_2$

$\dot{y}_2 = -y_1^2$

Теорема 4.2. *1. Итеративно применяя алгоритмы полиномиализации с помощью введения алгебраических и дифференциальных уравнений, нелинейную систему с элементарными нелинейностями можно привести к полиномиальной форме.*

2. Размер полученной полиномиальной системы является линейным относительно числа элементарных функций в оригинальной системе.

Доказательство. Доказательство первой части теоремы уже изложено вместе с алгоритмами полиномиализации.

Рассмотрим вторую часть. Заметим, что для того, чтобы избавиться от основной (не композитной) элементарной функции, требуется конечное число замен $O(1)$, обычно 1 или 2, например для $sin(x)$.

Для композитных элементарных функций $g(\vec{x}) = (g_1 \circ g_2 ... \circ g_n)(\vec{x})$ необходимо потратить $O(1 \cdot m)$ замен. Таким образом, размер полученной полиномиальной системы линеен относительно числа элементарных функций в оригинальной системе. □

4.3 Анализ оригинальных алгоритмов

Мы показали, что с помощью оригинальных алгоритмов всегда можно провести полиномиализацию, однако провести её можно разными способами,

вводя при этом разное количество переменных [1]. Тем не менее, из теоремы 4.2 мы знаем, что максимальное число введённых переменных всегда линейно от относительно числа элементарных функций в оригинальной системе. Таким образом, мы можем сосредоточится на скорости алгоритма полиномиализации, оставляя при этом возможность получать достаточно оптимальные преобразования.

4.4 Абстрактные синтаксические деревья

Для того, чтобы удобно работать с математическими выражениями, нам понадобится определить на них структуру. Для этого мы воспользуемся синтаксическим анализом - процессом преобразования последовательности лексем формального языка с его формальной грамматикой. Результатом обычно является дерево разбора, которое отражает синтаксическую структуру входной последовательности лексем.

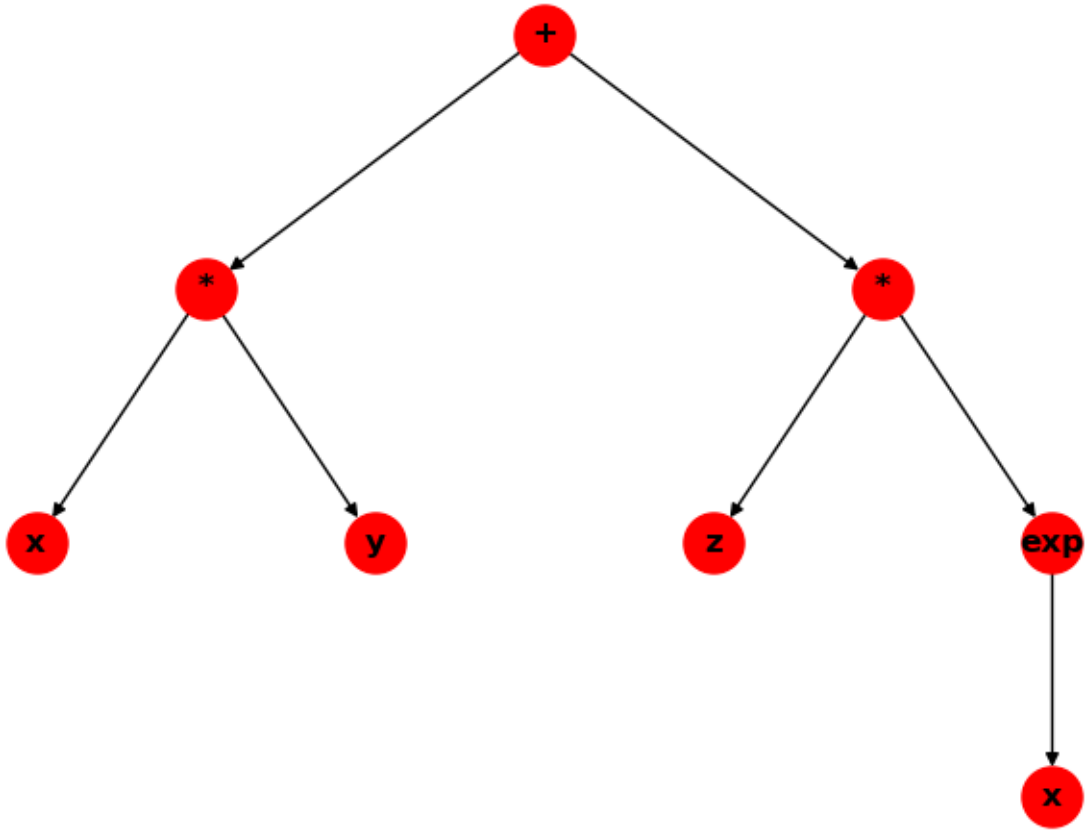


Рис. 1: Абстрактное синтаксическое дерево выражения $x \cdot y + z \cdot exp(x)$

В нашем случае мы получаем на вход строковое представление математи-

ческого выражения, а возвращаем дерево разбора, состоящее из внутренних узлов, представляющий математические функции, например как сложение или взятие логарифма, и листьев, состоящих из переменных и чисел.

Абстрактное синтаксическое дерево (*АСД*, *AST*) отличается от дерева разбора тем, что в нём отсутствуют узла и рёбра, которые не влияют на семантические свойств выражения. Например, для выражения $3+2\cdot4$ не нужно расставлять группирующие скобки, так как приоритет операции умножения задан выше, чем приоритет операции сложения.

Многие языки программирования и системы компьютерной алгебры используют данный подход для обработки для обработки своих выражений. В частности, АСД использует фреймворк SymPy [15], реализованный на языках программирования Python и Julia, который мы будем использовать в дальнейшем.

Подробнее ознакомиться с синтаксическими деревьями можно в [13], [14]

4.5 Алгоритм полиномиализации

Для начала перепишем алгоритмы из секций 4.1, 4.2 в формальном общем виде. Как уже упоминалось в разделе 4.4, нам удобно представлять правые части уравнений системы в виде абстрактных синтаксических деревьев. Из таких деревьев мы формируем лес, которые впоследствии мы будем анали-

зировать.

Algorithm 1: Полиномиализация

Data: System - система вида (2) или (3)

Result: Полиномиализированная система

Function `Polynomialize(`*System*`):`

```
NewSystem = Копия(System);
/* Лес абстрактных синтаксических деревьев                    */
Forest = {  };
/* массив меток, где true - данное дерево не содержит
   неполиномиальных элементов                                 */
ForestIsPoly = {  };
foreach r: правое уравнение системы do
    tree = AST(r);
    Добавить(Forest, tree);
    ForestIsPoly[tree] = false;
end
while ForestIsPoly содержит false do
    g = GetNonPolynomialItem(Forest, ForestIsPoly);
    ДобавитьУравнение(NewSystem, g);
end
return NewSystem;
```

Function `GetNonPolynomialItem(`*Forest, ForestIsPoly*`):`

```
foreach tree: ForestIsPoly[tree] == false do
    nonPolyItem = FindNonPolynomialItem(tree);
    if nonPolyItem ≠ null then
        return nonPolyItem;
    else
        ForestIsPoly[tree] = true;
    end
end
return null;
```

4.5.1 Прямой обход

Разберём вариант реализации алгоритма FindNonPolynomialItem, который предполагает прямой проход по синтаксическому дереву - от корня к листьям.

Algorithm 2: Прямой обход синтаксического дерева

Data: node - узел AST, составленного из правой части уравнения системы (2) или (3)

Result: Неполиномиальный элемент или null, если такой не найдётся

Function `FindNonPolynomialItem(`*node*`):`

```
if неПолиномиальнаяФункция(node) then
    return node;
end
foreach child: дети node do
    nonPolyItem = FindNonPolynomialItem(child);
    if nonPolyItem ≠ null then
        return nonPolyItem;
    end
end
return null;
```

4.5.2 Обратный обход

Теперь рассмотрим вариант реализации алгоритма FindNonPolynomialItem, который предполагает обратный проход по синтаксическому дереву - от листьев к корню.

Algorithm 3: Обратный обход синтаксического дерева

Data: node - узел AST, составленного из правой части уравнения системы (2) или (3)

Result: Неполиномиальный элемент или null, если такой не найдётся

```
Function FindNonPolynomialItem(node):
    foreach child: детu node do
        nonPolyItem = FindNonPolynomialItem(child);
        if nonPolyItem ≠ null then
            return nonPolyItem;
        end
    end
    if неПолиномиальнаяФункция(node) then
        return node;
    end
    return null;
```

4.5.3 Сравнение

Прямой и обратный обходы по синтаксическому дереву похожи на поиск в ширину и поиск в глубину соответственно.

Прямой обход последовательно проходит уровни глубины синтаксического дерева, возвращая первый встреченный неполиномиальный элемент. Таким образом, данный алгоритм работает быстрее, чем обратный подход в том случае, когда нелинейный элемент находится неглубоко, что характерно для для практических задач. Однако важно помнить, что если найденный нелинейный элемент является композицией функций $g = g_1 \circ g_2 \circ \cdots \circ g_k$, то прямой обход вернёт всю композицию. Это является существенным недостатком, если мы используем алгоритм 4.2, требующий вычислять сложную производную функции g, что является весьма дорогой операцией для композиции функ-

ций.

Обратный обход, в свою очередь, ищет самые глубоко вложенные неполиномиальные элементы g_k. Поэтому, он всегда находит базовые элементарные функции, имеющие тривиальные производные, затрачивая, в целом, большее число шагов.

Пример. Для выражения $x+\sin(\exp x)$ прямой обход найдёт найдёт $\sin(\exp x)$, а обратный - $\exp x$.

Таким образом, для полиномиализации 4.1 мы предпочтём прямой обход, а для 4.2 - обратный.

5 Квадратизация

Определение 5.1. Процесс преобразования нелинейных систем (2), (3) к квадратичному виду будем называть *квадратизацией*.

В данном разделе мы будем рассматривать оригинальные алгоритмы квадратизации, применимые только для полиномиальных систем [1]. Мы уже описали способы получения таких систем из систем (2), (3) в секции 4. Так же заметим, что подходы, применяемые для квадратизации систем, весьма похожи на методы полиномиализации.

5.1 Квадратизация с помощью введения алгебраических уравнений

Данный алгоритм квадратизует систему, добавляя к ней алгебраические уравнения.

1. Ввести новую переменную $y_i = m_i(\vec{x})$, где $m_i(\vec{x}) = x_1^{b_1} \cdot \dots \cdot x_n^{b_n}$ - моном, образованный из переменных системы.
2. Заменить $m_i(\vec{x})$ на y_i в оригинальном уравнении.
3. Добавить новое уравнение $y_i = m_i(\vec{x})$ в систему.

Показанный алгоритм приводит полиномиальные системы вида (2), (3) к квадратичным системам вида (3).

Пример. Квадратизуем систему, полученную в примере раздела 4.1:

$$\dot{x} = x^3 + y_1$$
$$0 = xy_1 + y_1 - 1$$

Теперь проведём квадратизацию:

1. Вводим новую переменную $y_2 = x^2$
2. Делаем замену в первом уравнении $\dot{x} = xy_2 + y_1$
3. Добавляем новое уравнение в систему $y_2 - x^2 = 0$

Получили квадратичную систему

$\dot{x} = xy_2 + y_1$

$0 = xy_1 + y_1 - 1$

$0 = y_2 - x^2$

5.2 Квадратизация с помощью введения дифференциальных уравнений

Следующий алгоритм проводит квадратизацию, добавляя в систему ОДУ:

1. Ввести новую переменную $y_i = m_i(\vec{x})$, где $m_i(\vec{x}) = x_1^{b_1} \cdot \dots \cdot x_n^{b_n}$ - моном, образованный из переменных системы.

2. Заменить $m_i(\vec{x})$ на y_i в оригинальном уравнении

3. Добавить новое уравнение $\dot{y}_i = \dot{m}_i(\vec{x}) = m_i'(\vec{x})\dot{\vec{x}}$ в систему, где $m'(\vec{x}) = \frac{dm(\vec{x})}{d\vec{x}}$

Данный алгоритм переводит полиномиальные системы к квадратичному виду, не меняя их структуры: (2) $\longrightarrow$ (2), (3) $\longrightarrow$ (3).

Пример. Рассмотрим систему

$\dot{x} = xz^2 + z$

$\dot{z} = x$

Теперь проведём квадратизацию:

1. Вводим новую переменную $y = z^2$

2. Делаем замену в первом уравнении $\dot{x} = xy + z$

3. Добавляем новое уравнение в систему $\dot{y} = 2z\dot{z} = 2xz$

Получили квадратичную систему

$\dot{x} = xy + z$

$\dot{y} = 2xz$

$\dot{z} = x$

5.3 Анализ алгоритмов квадратизации

В отличии от полиномиализации, число введённых алгоритмами квадратизации 5.1, 5.2 уравнений уже не является линейным относительно возможных замен переменных. Более того, как мы покажем далее, число возможных замен так же заметно больше. Поэтому нам становится важен вопрос об оптимальной по числу введённых уравнений квадратизации. Для этого нам понадобится использовать более продвинутые методы анализа, нежели чем для полиномиализации.

Так же заметим, что мы пользуемся только мономиальными заменами, что, вообще говоря, не даёт нам возможности получить оптимальную квадратизацию в общем случае, так как она может достигаться полиномиальными заменами.

5.4 Генерация замен переменных

Генерировать возможные мономиальные замены переменных из монома мы будем рекурсивно, постепенно деля изначальный моном, а далее мономы из его разложений на все их релевантные делители.

Algorithm 4: Генерация замен переменных

Data: monomial - входной моном, чьи разложения мы хотим получить.

Result: Все релевантные разложения монома.

Function `GenerateReplacements(monomial):`

```
/* Множество всех разложений monomial                    */
AllDecompositions = {  };
GenerateReplacementsRecursive(monomial, {}, AllDecompositions);
return Уникальные(AllDecompositions);
```

Function `GenerateReplacementsRecursive(monomial, decomposition, AllDecompositions):`

```
if decomposition не пусто then
    Добавить(AllDecompositions, decomposition);
end
/* Генерирует все делители монома степенью не менее 2 и не
   более M - 1, где M - степень monomial                 */
делители = СгенерироватьДелители(monomial);
foreach делитель: делители do
    GenerateReplacementsRecursive(monomial/ делитель,
     Добавить(decomposition, делитель), AllDecompositions);
end
```

Таким образом, повторив эту процедуру для каждого уникального монома в полиномной системе, мы получим все возможные для неё замены переменных.

Пример. $x^2y^3 \implies (x, xy^3), (xy, xy^2), (x^2y^3), (y, x^2y^2)$ $(y, x^2, y^2), (x, xy, y^2), (y^2, x^2y), (y, xy, xy)$

Заметим, что приведённый алгоритм иллюстрирует лишь идеею получения замен и не является оптимальным способом это делать, так как алгоритм генерирует много эквивалентных разложений, представляющие собой одно и то же разложение, записанное в разном порядке.

Пример. $(x^2, xy, y^2) \sim (x^2, y^2, xy) \sim (xy, y^2, x^2) \sim \cdots$

Число таких перестановок $P_n - n!$, где n - степень монома. Таким образом, сокращение числа перестановок, которые генерируются алгоритмом 4, можно считать главной задачей для его оптимизации. Мы предлагаем в качестве отправной точки попробовать создать дерево, основанное на сжатых префиксных и суффиксных деревьях. Так же, важно заметить, что моном фиксированной степени и числа переменных имеет единственный набор уникальных разложений. Поэтому, можно заранее вычислить все разложения для мономов, часто встречающихся на практике, и использовать их, не вычисляя в момент исполнения программы.

5.5 Граф замен

Теперь обобщим процесс выбора замен переменных.

Определение 5.2. *Граф замен* - ациклический ориентированный граф, чьи вершины представляют системы уравнений, эквивалентные заданной системе, а дуги - преобразования, совершаемые с помощью замены переменной. Входной вершиной мы полагаем заданную систему, а выходной - систему, обладающей нужными нам свойствами.

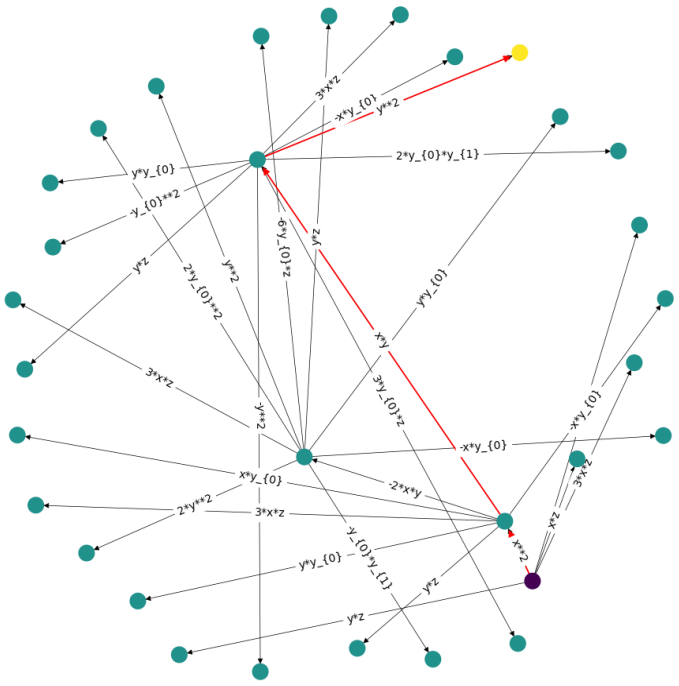


Рис. 2: Граф замен

Таким образом, мы можем трактовать задачу об оптимальной квадратизации как задачу поиска на графе с одним входом и несколькими выходами. Важно заметить, что графы замен, которые нам будут встречаться на практике, часто являются огромными и заранее задан только начальный узел, что делает неприменимыми значительную часть классических алгоритмов поиска.

5.6 Алгоритмы поиска на графе

В данном разделе мы рассмотрим основные алгоритмы поиска на графе. Подчеркнём, что сам путь нас не интересует, поэтому мы приводим алгоритмы поиска пути на графах, которые, в связи с этим упрощаются.

Пусть G - ориентированный граф, V - множество его рёбер, E - множество его дуг, b - средний коэффициент ветвления G, l - минимальная глубина выходного узла.

5.6.1 BFS и DFS

Поиск в глубину (depth-frist search, DFS) и поиск в ширину (breadth-first search BFS) являются базовыми в задаче поиска, нередко оказываясь достаточно эффективными для многих задач. К сожалению, в чистом виде применить данные алгоритмы для нашей задачи не представляется возможным. Действительно, глубина всего графа замен может быть близка к бесконечной при неаккуратном выборе замен, что критично для DFS. В случае BFS нам мешает высокий коэффициент ветвления, который, более того, растёт при для каждого последующего уровня глубины. Таким образом, BFS пригоден для практического использования только в том случае, когда искомая система находится на небольшой глубине.

5.6.2 Поиск с ограничением глубины (DLS)

Определение 5.3. *Поиск с ограничением глубины* (depth-limited search, DLS) [16] - вариант поиска в глубину, для которого определяется конечная глубина l на которую он может опуститься. Таким образом, алгоритм всегда работает конечное число шагов, в отличие от DFS, однако ответ не может быть найден, если глубина выходного узла $d > l$.

Algorithm 5: Поиск с ограничением глубины

Data: G - входной граф
prop - свойство, которым должна обладать выходная вершина
visited - массив, хранящий для каждой вершины бит, показывающий, посещали ли мы эту вершину
l - предельная глубина

Result: Вершина, обладающая свойством prop или null, если вершина не найдена

Function `DLS(`*G, v, prop, depth, l*`)`**:**

```
visited[v] = true;
if v обладает свойством prop then
    return v;
end
foreach w: соседv do
    if visited[w] == false & depth < l then
        return (G, w, prop, depth + 1, l);
    end
end
return null;
```

Данный алгоритм не уходит дальше заданной глубины, что решает проблему DFS и просматривает узлы на желаемой глубине, а не последовательно идя по слоям, как BFS. Таким образом, при удачно выбранной конечной глубине и хорошем выборе замен на каждом шаге, мы сможем решать задачу за сравнительно небольшое число шагов.

Однако, если мы оценили конечную глубину ниже, чем глубину искомой вершины, то решить задачу мы не сможем. Решает эту проблему следующий алгоритм поиска.

5.6.3 Поиск в глубину с итеративным углублением (ID-DFS)

Определение 5.4. *Поиск в глубину с итеративным углублением* (iterative-deepening depth-first search, ID-DFS) [16] запускает DLS на каждой своей итерации и, в случае неудачи, увеличивает конечную глубину l.

Algorithm 6: Поиск в глубину с итеративным углублением

Data: G - входной граф

prop - свойство, которым должна обладать выходная вершина

Result: Вершина, обладающая свойством prop или null, если вершина не найдена

Function `IDDFS(`G, v_{start}, $prop$`):`

```
l = 1;
do
    v = DLS(G, v, prop, 1, l);
    l += 1;
while v == null;
return v;
```

Не смотря на то, что ID-DFS гарантирует нахождение выходной вершины, в отличие от DLS, у него есть 2 существенных недостатка:

1. Необходимость исследовать граф заново, если на текущей итерации выходную вершину найти не получилось.

2. Увеличение глубины происходит всегда на 1, что не всегда является оптимальной стратегией.

Поэтому мы несколько модифицируем ID-DFS, чтобы устранить данные недостатки.

5.6.4 Поиск с ограничением глубины и итеративным углублением (ID-DLS)

Определение 5.5. Введём новый алгоритм *Поиск с ограничением глубины с итеративным углублением* (*ID-DLS*), который отличается от ID-DFS в следующем:

1. Вместо исследования графа заново после неудачной итерации, ID-DLS сохраняет вершины на текущей конечной глубине вместе с их исходящими заменами. В случае неудачного поиска мы сразу начинаем вычислять неисследованные элементы. Платой за это становится удвоенные затраты на память.

2. Текущая конечная глубина более гибко изменяется за счёт функции . Платой за это становится то, что гарантия оптимальности квадратизации ложится на приведённую функцию. Тем не менее, таким образом мы может искать суб-оптимальные решения с погрешностью, также определяемой функцией .

Algorithm 7: Поиск с ограничением глубины с итеративным углублением

Data: G - входной граф

v_{start} - входная вершина

prop - свойство, которым должна обладать выходная вершина

limit - финальная конечная глубина

Result: Вершина, обладающая свойством prop или null, если вершина не найдена

Function `IDDLS(`*G, v_{start}, prop, limit*`):`

```
currentLimit = 1;
/* В данную очередь помещаются элементы на текущей конечной
   глубине вместе с их исходящими заменами. Фактически, мы
   лениво вычисляем вершины со следующего уровня          */
highDepthQueue = { };
do
    if currentLimit > limit then
        return null
    end
    v = DLS(G, v, prop, 1, currentLimit, highDepthQueue);
    currentLimit =
     ОбновитьТекущуюКонечнуюГлубину(currentLimit);
while v == null;
return v;
```

Дополнительным улучшением к данному алгоритму могут послужить эффективные оценки верхней и нижней границ минимальной глубины выходного узла l, которые позволят просматривать как можно меньше неперспективных узлов.

5.7 Эвристический подход к поиску

Эвристический поиск, он же информированный поиск представляет собой семейство стратегий поиска, в котором используются знания о конкретной задаче, зачастую позволяя решать задачу поиска гораздо эффективнее.

Знания о задаче формализуются в качестве эвристических функций. Эвристические функции сравнивают между собой варианты, из которых алгоритм выбирает следующий шаг.

В случае задач поиска на графе, с помощью эвристических функций мы будем сортировать дуги алгоритмов неинформированного поиска.

Приведём эвристические версии алгоритмов неинформированного поиска

1. DFS $\rightarrow$ Поиск по первому наилучшему совпадению (best-first search) [17]
2. Алгоритм Дейкстры $\rightarrow$ Алгоритм A* [18]
3. ID-DFS $\rightarrow$ Алгоритм A* с итеративным углублением (Iterative deepening A*, IDA*) [16]

5.8 Эвристики для квадратизации

Точкой ветвления алгоритма квадратизации является шаг, на котором мы выбираем, в какой порядке исследовать возможные замены переменных. Таким образом, определим семейство эвристических функций в задаче квадратизации как

$$h: \ \mathbb{M} \longrightarrow \mathbb{R}, \tag{5}$$

где $\mathbb{M}$ - группа мономов, образованных над переменными $\vec{x}$ полиномных систем (2), (3).

Эвристика 5.6. *Frequent-First, FF* - эвристика, выбирающая моном, наиболее часто встречающийся в разложениях. Таким образом, замена переменной затронет наибольшее число мономов в системе.

Пример. Для разложений $(x^2, xy), (xy, y^2, y^3, xy^2, xy^3), (x^2, y^2, xy)$ мы выберем моном xy, так как он встречается в трёх разложениях, а остальные не более чем в двух.

Эвристика 5.7. *Free-Variables-Count, FVC* - эвристика, выбирающая моном, образованный из наименьшего числа переменных. Таким образом, в уравнении $y_i = m'(\vec{x})\dot{\vec{x}}$, соответствующем данной замене, будет минимальное число слагаемых.

Пример. Из мономов x^2, xy, xyz мы выберем x^2, имеющий только одну переменную. Покажем ОДУ, образованные данными заменами:

1. $x^2 \longrightarrow \dot{w} = 2x\dot{x}$
2. $xy \longrightarrow \dot{w} = \dot{x}y + x\dot{y}$
3. $xyz \longrightarrow \dot{w} = \dot{x}yz + x\dot{y}z + xy\dot{x}$

Эвристика 5.8. *Auxiliary-Equation-Degree, AED* - эвристика, выбирающая моном, порождающий уравнение с наименьшей степенью.

Пример. Пусть имеем систему со следующим распределение степеней: $degree(\dot{x}) = 1$, $degree(\dot{y}) = 2$. Тогда для замен x^2, xy, y^2 получим степени уравнений:

1. $x^2 \longrightarrow \dot{w} = 2x\dot{x} \longrightarrow 1 + 1 = 2$
2. $xy \longrightarrow \dot{w} = \dot{x}y + x\dot{y} \longrightarrow max(1 + 1, 1 + 2) = 3$
3. $y^2 \longrightarrow \dot{w} = 2y\dot{y} \longrightarrow 1 + 2 = 3$

Из данных замен выберем x^2.

Эвристика 5.9. *Auxiliary-Equation-Quadratic-Discrepancy, AEQD* - эвристика, выбирающая моном, чьё порождённое уравнение наименее отличается от квадратичного. Таким образом, замены, порождающие квадратичные уравнения, имеют самый высокий приоритет.

Пример. Пусть имеем замены, порождающие следующие уравнения

1. $m_1 \longrightarrow \dot{w} = x + y^2 + z^3 \longrightarrow 0 + 0 + 1 = 1$
2. $m_2 \longrightarrow \dot{w} = x + y^2 + xy + z^3 \longrightarrow 0 + 0 + 0 + 1 = 1$
3. $m_3 \longrightarrow \dot{w} = x + y^2 + z^3 + xyz \longrightarrow 0 + 0 + 1 + 1 = 2$

Таким образом, m_1 и m_2 имеют одинаковый приоритет, потому что порождённые ими уравнений имеют минимальную степерь 1.

Эвристика 5.10. *Summary-Monomial-Degree, SMD* - эвристика, представляющая развитие идеи FF. Мы выбираем замену, которая максимально понижает степень системы.

Пример. Рассмотрим систему

$\dot{x} = xy^2 + y^3$

$\dot{y} = xy + x^2y + 1$

Понижение степени системы для замены m вычисляется как $N \cdot (degree(m) - 1)$, где N - число мономов в системе, которые нацело делятся на m.

1. $y^2 \longrightarrow 2 \cdot (2 - 1) = 2$
2. $xy \longrightarrow 3 \cdot (2 - 1) = 3$
3. $y^3 \longrightarrow 1 \cdot (3 - 1) = 2$

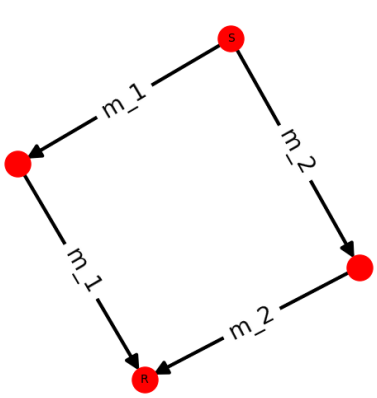
Получили, что замена xy максимально сильно снижает степень системы.

5.9 Правило параллелограмма

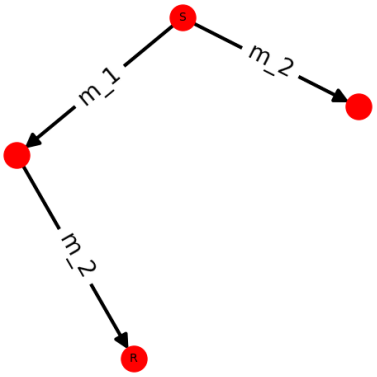
Пусть для начальной полиномиальной системы S имеются две замены m_1 и m_2.

Утверждение 5.11. *Замены m_1 и m_2 приведут к одной и той же системе R вне зависимости от порядка их проведения. Действительно, ведь в таком случае введённые переменные y_1 и y_2 получаются перестановкой $[y_1 = y_2, y_2 = y_1]$.*

Таким образом, мы можем убирать часть рёбер из графа во время его исследования. Данная оптимизация становится весьма важна для параллельных версий алгоритма, так как даёт возможность потокам оптимизировать работу друг друга.



(a) Граф замен для S



(b) Упрощённый по правилу параллелограмма граф 3a

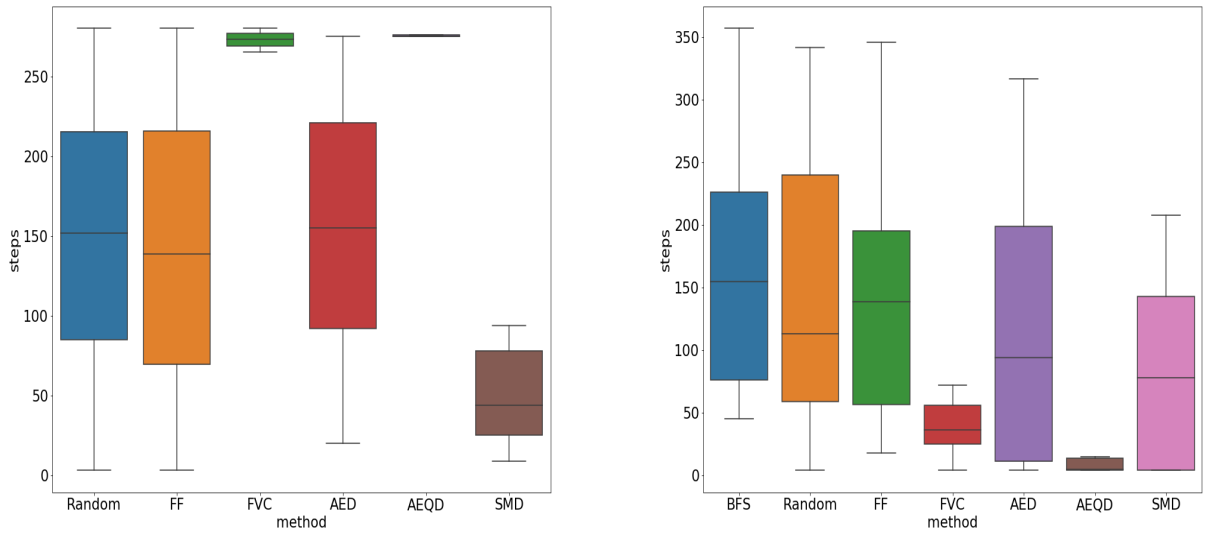
Рис. 3: Правило параллелограмма

6 Эксперименты

Сравним число шагов, которые понадобятся ID-DLS с разным выбором эвристик, чтобы квадратизировать различные нелинейные системы.

Nonlinear system	Random	FF	FVC	AED	AEQD	SMD
xSigmoid	145± 85	140 ± 83	273 ± 4	146 ± 77	275 ± 1	**52 ± 31**
Rabinovich-Fabrikant [19] [20]	150 ± 102	142 ± 88	39 ± 22	113 ± 104	**9 ± 5**	76 ± 79
Blue-sky catastrophe [21] [22] [23]	?	?	?	12240 ± 5231	1284 ± 1138	**11 ± 1**

Таблица 1: Сравнение эвристик для ID-DLS



(a) xSigmoid (b) Rabinovich-Fabrikant

Рис. 4: Сравнение эвристик

Таким образом, наиболее успешно показали себя эвристики *AEQD* и *SMD*. Так же *AED* и *FVC* показывают, в целом, результаты лучшие, чем ID-DLS с отсутствием какой либо эвристики.

7 Заключение

Существующие алгоритмы полиномиализации и квадратизации не были в достаточной степени формальными, чтобы исполняться на вычислительных устройствах. В данной работе мы предложили формализацию данных алгоритмов на языке графов и свели задачу об оптимальной квадратизации к задаче поиска на графе.

Было показано, какие алгоритмы неинформированного поиска могут справиться с задачей лучше всего. После этого лучший алгоритм был модифицирован до своей эвристической версии. К полученному эвристическому алгоритму был предложен ряд эвристик, после чего было проведено их сравнение. Помимо этого, была проведено разрежение графа, на котором осуществляется поиск, с помощью знаний о предметной области. Таким образом удалось добиться значительного ускорения скорости работы по сравнению с базовыми алгоритмами.

Таким образом, предложенные алгоритмы полиномиализации и квадратизации могут использованы для достаточно быстрого преобразования небольших систем уравнений. Полученный программный пакет, реализованный на языке Python, находится в свободном распространении [24].

Для того, чтобы работать с крупными системами, в будущем будут проведены:

1. Оптимизация представления полиномиальных математических выражений, которые могут быть представлены как список кортежей фиксированного размера. Это позволит значительно снизить объём занимаемой памяти и время копирования системы, на данный момент занимающее треть вычислительных ресурсов.

2. Параллелизация алгоритма квадратизации.

3. Добавление новых эвристик и синтез уже существующих.

4. Использование языка программирования Julia.

Список литературы

[1] Gu C. Model order reduction of nonlinear dynamical systems : дис. – UC Berkeley, 2011.

[2] Kramer B., Willcox K. E. Nonlinear model order reduction via lifting transformations and proper orthogonal decomposition //AIAA Journal. – 2019. – Т. 57. – №. 6. – С. 2297-2307.

[3] Harrington H. A., Van Gorder R. A. Reduction of dimension for nonlinear dynamical systems //Nonlinear Dynamics. – 2017. – Т. 88. – №. 1. – С. 715-734.

[4] Korf R. E. Depth-first iterative-deepening: An optimal admissible tree search //Artificial intelligence. – 1985. – Т. 27. – №. 1. – С. 97-109.

[5] Blackmore D. L. et al. Nonlinear dynamical systems of mathematical physics: spectral and symplectic integrability analysis. – World Scientific, 2011.

[6] Karri R. R. Evaluating and estimating the complex dynamic phenomena in nonlinear chemical systems //International Journal of Chemical Reactor Engineering. – 2011. – Т. 9. – №. 1.

[7] Jackson T., Radunskaya A. (ed.). Applications of Dynamical Systems in Biology and Medicine. – Springer, 2015. – Т. 158.

[8] Lines M. (ed.). Nonlinear dynamical systems in economics. - Springer Science & Business Media, 2007. - Т. 476.

[9] Schilders W. H. A., Van der Vorst H. A., Rommes J. Model order reduction: theory, research aspects and applications. – Berlin : Springer, 2008. – Т. 13.

[10] Strogatz S. H. Nonlinear dynamics and chaos with student solutions manual: With applications to physics, biology, chemistry, and engineering. – CRC press, 2018.

[11] Shen K., Scott J. K. Rapid and accurate reachability analysis for nonlinear dynamic systems by exploiting model redundancy //Computers & Chemical Engineering. – 2017. – Т. 106. – С. 596-608.

[12] Antoulas A. C., Sorensen D. C. Approximation of large-scale dynamical systems: An overview. – 2001.

[13] Hopcroft J. E., Motwani R., Ullman J. D. Introduction to automata theory, languages, and computation //Acm Sigact News. – 2001. – Т. 32. – №. 1. – С. 60-65.

[14] Aho A. V., Sethi R., Ullman J. D. Compilers, principles, techniques //Addison wesley. – 1986. – Т. 7. – №. 8. – С. 9.

[15] https://www.sympy.org/ru/index.html

[16] Korf R. E. Depth-first iterative-deepening: An optimal admissible tree search //Artificial intelligence. – 1985. – Т. 27. – №. 1. – С. 97-109.

[17] Dechter R., Pearl J. Generalized best-first search strategies and the optimality of A //Journal of the ACM (JACM). – 1985. – Т. 32. – №. 3. – С. 505-536.

[18] Hart P. E., Nilsson N. J., Raphael B. A formal basis for the heuristic determination of minimum cost paths //IEEE transactions on Systems Science and Cybernetics. – 1968. – Т. 4. – №. 2. – С. 100-107.

[19] Rabinovich M. I., Fabrikant A. L. Stochastic self-modulation of waves in nonequilibrium media //J. Exp. Theor. Phys. – 1979. – Т. 77. – С. 617-629.

[20] Danca M. F., Chen G. Bifurcation and chaos in a complex model of dissipative medium //International Journal of Bifurcation and Chaos. – 2004. – Т. 14. – №. 10. – С. 3409-3447.

[21] Turaev D. V., Shil'nikov L. P. Blue sky catastrophes //Doklady Mathematics-Interperiodica Translation. – 1995. – Т. 51. – №. 3. – С. 404-407.

[22] Shilnikov A., Cymbalyuk G. Transition between tonic spiking and bursting in a neuron model via the blue-sky catastrophe //Physical review letters. – 2005. – Т. 94. – №. 4. – С. 048101.

[23] Van Gorder R. A. Triple mode alignment in a canonical model of the blue-sky catastrophe //Nonlinear Dynamics. – 2013. – Т. 73. – №. 1-2. – С. 397-403.

[24] https://github.com/AndreyBychkov/QBee