

Оглавление

Введение	2
Резольвентное кубическое уравнение	4
Принцип главного значения квадратного корня радикалов	5
Уравнение четвертой степени	7
Алгоритм Феррари	9
Алгоритм Декарта.....	16
Алгоритм NBS.....	18
Алгоритм Эйлера	25
Алгоритм Ван дер Вардена	32
Эквивалентность решений подавляющих алгоритмов	37
Классификация методов	40
Аналитический метод без использования резольвентных полиномиальных уравнений	40
Метод, итеративно вычисляющий коэффициенты вспомогательных полиномиальных уравнений, решаемых аналитически	41
Аналитический метод, нуждающийся в отыскании всех корней резольвентного полиномиального уравнения	42
Аналитический метод, нуждающийся в отыскании всех корней резольвентного полиномиального уравнения и в отыскании корней вспомогательных полиномиальных уравнений.....	43
Организация экспериментов	44
Гибридно-аналитический способ	45
Численный способ.....	48
Результаты экспериментов (гибридно-аналитический способ).....	49
Результаты экспериментов (численный способ).....	62
Найденные неточности исходной литературы.....	66
Алгоритм Ангера.....	66
Алгоритм Скуайра.....	68
Заключение.....	71
Список литературы	73
Приложение	75

Введение

Актуальность исследования:

Аналитическое вычисление корней многочленов третьей и четвертой степени имеет фундаментальное значение в математике и широкое применение в различных областях, таких как машинная графика, приближения многочленами и дробно-рациональными функциями, численные методы, анализ систем и другие. Точное определение корней многочленов позволяет решать сложные задачи, включая моделирование систем, оптимизацию функций и анализ данных.

Несмотря на наличие множества методов аналитического решения полиномиальных уравнений, постоянный интерес исследователей к этой области объясняется несколькими факторами. Во-первых, аналитическое вычисление корней многочленов третьей и четвертой степени является фундаментальной задачей, и ее решение имеет важное значение для развития математической науки. Во-вторых, существующие методы решения многочленов третьей и четвертой степени ограничены и не всегда обеспечивают высокую устойчивость к погрешностям, возникающим в результате компьютерных вычислений с плавающей точкой.

Одной из основных трудностей, с которыми исследователи сталкиваются при аналитическом вычислении корней многочленов третьей и четвертой степени, является погрешность, связанная с ограничениями точности компьютерных вычислений. Методы, основанные на аналитических формулах, могут быть чувствительны к погрешностям округления и представления чисел с плавающей точкой. Это ограничение приводит к необходимости разработки более эффективных и надежных методов вычисления корней многочленов третьей и четвертой степени.

Таким образом, разработка новых подходов и улучшение существующих методов вычисления корней многочленов третьей и четвертой степени остаются актуальными задачами, привлекающими исследователей со всего мира. Исследования в этой области не только способствуют развитию математики, но

и имеют практическую ценность, обеспечивая более точные и надежные решения для широкого спектра приложений.

Цель исследования:

Целью данной работы является изучение различных методов аналитического вычисления вещественных корней многочленов третьей и четвертой степени, анализ погрешностей этих методов и разработка программной реализации для эффективного вычисления корней многочленов.

Задачи исследования:

В рамках данной работы были поставлены следующие задачи:

- изучение теоретических основ методов вычисления корней многочленов третьей и четвертой степени;
- анализ погрешностей этих методов и определение их точности;
- разработка программной реализации этих методов на языке программирования C++;
- сравнение эффективности и точности различных методов аналитического вычисления корней многочленов третьей и четвертой степени.

Следует отметить, что полиномиальные уравнения третьей степени рассматривается на этапе поиска решений вспомогательных уравнений при решении полиномиального уравнения четвертой степени.

Объект исследования:

Объектом исследования данной работы являются методы вычисления корней многочленов третьей и четвертой степени.

Предмет исследования:

Предметом исследования данной работы являются теоретические основы методов вычисления корней многочленов третьей и четвертой степени, анализ их погрешностей, разработка программной реализации и анализ точности этих методов.

Практическая значимость:

Практическая значимость данной работы заключается в том, что она может быть использована для разработки эффективных алгоритмов вычисления корней многочленов третьей и четвертой степени, которые могут быть применены в различных областях науки и техники. Кроме того, результаты исследования могут быть использованы в качестве основы для дальнейших исследований в области аналитического вычисления корней многочленов.

Резольвентное кубическое уравнение

С помощью резольвентного кубического уравнения [1] алгоритмы, использующие подавление члена третьей степени Феррари, Декарта, Эйлера и Ван дер Вардена, находят решения уравнения четвертой степени. Решения резольвентного кубического уравнения для каждого метода эквивалентны друг другу, так как каждый алгоритм использует преобразования, которые выражаются через другие преобразования:

$$2m = y^2 = 4r_k = -\theta_k.$$

где m – наибольшее решение резольвентного кубического уравнения Феррари, y^2 – наибольшее решение резольвентного бикубического уравнения Декарта, r_k – решения резольвентного кубического уравнения Эйлера, θ_k – решения резольвентного кубического уравнения Ван дер Вардена.

Алгоритмы Феррари, Декарта и Эйлера рассматривают наибольшее решение резольвентного кубического уравнения, тогда как алгоритм Ван дер Вардена рассматривает наименьшее решение:

$$2m = y^2 = 4r_1 = -\theta_1. \quad (1)$$

Резольвентное уравнение является важной частью каждого алгоритма, поскольку его решения используются для нахождения решений T_n подавленного уравнения, к которому сводится общее решение.

Следующие рассуждения представлены на примере алгоритма Феррари. Резольвентное кубическое уравнения Феррари имеет вид:

$$m^3 + b_2 m^2 + \left(\frac{b_2^2}{4} - b_0 \right) m - \frac{b_1^2}{8} = 0. \quad (2)$$

Коэффициент $-\frac{b_1^2}{8}$ является неположительной величиной. Резольвентное уравнение имеет решение $m = 0$ в том случае, когда $b_1 = 0$. В противном случае, если $m = 0$, то кубическое выражение дает отрицательную величину, что не удовлетворяет равенству (2). Поэтому решение m является положительной величиной. Таким образом, решение резольвентного кубического уравнения $m \geq 0$. Исходя из выше сказанного и, беря во внимание (1), получаем:

$$2m = y^2 = 4r_1 = -\theta_1 \geq 0. \quad (3)$$

Выведенные величины являются не отрицательными, далее они используются в рамках вычислений решений T_n подавленного уравнения четвертой степени.

Принцип главного значения квадратного корня радикалов

Принцип главного значения квадратного корня радикалов — это математическая концепция, используемая для определения главного значения квадратного корня из неотрицательного вещественного числа. Согласно этой идее, квадратный корень из неотрицательного вещественного числа всегда берется как положительный квадратный корень. Например, главное значение квадратного корня из некоторого вещественного положительного числа x — это \sqrt{x} , а не $-\sqrt{x}$, так как \sqrt{x} — это положительный квадратный корень из x .

В рамках данной работы, данные идеи будут разобраны в рамках методов Эйлера и Ван Дер Вардена.

Данные принципы применяются в алгоритме Эйлера путем замены $\sqrt{r_3}$ на $\Sigma s \sqrt{r_3}$, где

$$\Sigma = \begin{cases} 1, & b_1 > 0 \\ -1, & \text{иначе} \end{cases}, \quad s = \begin{cases} 1, & \sqrt{r_1}\sqrt{r_2}\sqrt{r_3} \geq 0 \\ -1, & \text{иначе} \end{cases}. \quad (4)$$

Исходя из выведенных функций, следует переопределение и для других членов:

$$b_1 = \Sigma |b_1|, \quad \sqrt{r_1}\sqrt{r_2}\sqrt{r_3} = s|\sqrt{r_1}\sqrt{r_2}\sqrt{r_3}| = s \frac{|b_1|}{8}.$$

Отсюда получаем новые формулы решения T_n уравнения четвертой степени для алгоритма Эйлера:

$$T_{1,2} = \sqrt{r_1} \pm (\sqrt{r_2} - \Sigma s \sqrt{r_3}), \quad T_{3,4} = -\sqrt{r_1} \pm (\sqrt{r_2} + \Sigma s \sqrt{r_3}).$$

Произведение всех членов каждого решения T_n теперь выглядит следующим образом:

$$-\Sigma \sqrt{r_1}\sqrt{r_2}\sqrt{r_3} = -\Sigma s^2 \frac{|b_1|}{8} = -\Sigma \frac{|b_1|}{8} = \frac{b_1}{8},$$

что удовлетворяет требованиям алгоритма Эйлера. Те же действия применимы и для алгоритма Ван дер Вардена.

Условие $\sqrt{r_1}\sqrt{r_2}\sqrt{r_3} < 0$ функции s , описанной в (4), выполняется в том случае, когда один из r_k вещественных членов положителен, а остальные отрицательны:

$$\sqrt{r_2} = i\sqrt{|r_2|}, \sqrt{r_3} = i\sqrt{|r_3|} \Rightarrow \sqrt{r_2}\sqrt{r_3} = -\sqrt{|r_2|}\sqrt{|r_3|} = -\sqrt{r_2r_3} < 0.$$

В последующих пунктах данный принцип применяется для вывода решаемых формул для алгоритмов Эйлера и Ван дер Вардена с учетом модификаций Уолтерса [1].

Уравнение четвертой степени

Данный пункт посвящен изучению структуры и принципов решения уравнения четвертой степени общего вида:

$$Z^4 + A_3 Z^3 + A_2 Z^2 + A_1 Z + A_0 = (Z - Z_1)(Z - Z_2)(Z - Z_3)(Z - Z_4), \forall Z,$$

где A_n — вещественные коэффициенты уравнения, Z_n — искомые корни уравнения.

На вход поступают четыре вещественных коэффициента $A_n, n = 0, 1, 2, 3$.

На выходе требуется получить общее решение Z_n для уравнения:

$$Z_n^4 + A_3 Z_n^3 + A_2 Z_n^2 + A_1 Z_n + A_0 = 0, \quad n = 1, 2, 3, 4. \quad (1.1)$$

Для получения решения необходимо решить эквивалентное подавленное уравнение четвертой степени (1.2):

$$T_n^4 + b_2 T_n^2 + b_1 T_n + b_0 = 0, \quad n = 1, 2, 3, 4. \quad (1.2)$$

Эквивалентное подавленное уравнение четвертой степени применяется для упрощения решения исходного уравнения четвертой степени. Наличие сложных кубических членов в исходном уравнении затрудняет его решение, поэтому подавление степеней используется для избавления от кубического члена и получения эквивалентного уравнения четвертой степени, которое является более простым для решения.

Процедура подавления степеней достигается путем замены переменной, которая связывает исходные корни с новыми переменными. В случае уравнения четвертой степени это достигается преобразованием Чирнхауса [6], которое определяется следующим образом:

$$Z_n = T_n - \frac{A_3}{4} = T_n - C, \quad (1.3)$$

где T_n — новая переменная, C — константа сдвига.

Применение данной замены позволяет избавиться от кубического члена в исходном уравнении и получить эквивалентное подавленное уравнение четвертой степени (1.2), где b_2 , b_1 и b_0 — новые коэффициенты, зависящие от коэффициентов исходного уравнения:

$$\begin{aligned} C &= \frac{A_3}{4}, \\ b_2 &= A_2 - 6C^2, \\ b_1 &= A_1 - 2A_2C + 8C^3, \\ b_0 &= A_0 - A_1C + A_2C^2 - 3C^4. \end{aligned} \tag{1.4}$$

Решение эквивалентного подавленного уравнения четвертой степени дает решения относительно новой переменной T . Обратная замена (1.3) позволяет получить решение исходного уравнения четвертой степени относительно переменной Z .

Преобразование Чирнхауса является одним из методов подавления степеней в уравнениях. Другие методы включают использование подстановок, замен переменных и другие преобразования уравнений. Среди которых можно отметить:

- метод Горнера — численный метод для деления полинома на линейный множитель, использующий схему Горнера для более эффективного вычисления значений полинома;
- метод рекурсивного подстановочного деления — позволяющий упростить уравнения путем разложения на множители и последовательной подстановки полученных решений в исходное уравнение;
- метод Ньютона-Рафсона — численный метод для приближенного нахождения корней уравнений, основанный на итеративном применении формулы Ньютона-Рафсона и линейной аппроксимации функции и её производной.

В данном и последующих пунктах рассматривается анализ структуры алгоритмов, предложенных в работе Уолтерса [1] для решения уравнения четвертой степени. Из представленных далее алгоритмов, за исключением NBS, основная часть рассчитывает коэффициенты (1.4) на основе уравнения (1.2). После решения уравнения (1.2) получаем решение относительно T_n . Путем проведения обратной замены (1.3) получаем решение общего уравнения (1.1).

Алгоритм Феррари

Вначале в алгоритме Феррари рассчитывают коэффициенты (1.4) [2] подавленного уравнения (1.2). Далее, Феррари использует регулируемый параметр t для преобразования подавленного уравнения (1.2) в равенство двух полных квадратов вида:

$$A^2 = B^2 \Rightarrow A^2 - B^2 = 0,$$

где A — квадратично, B — линеен относительно T_n .

Это приводит к преобразованию коэффициентов уравнения четвертой степени в два легко решаемых квадратных уравнения вида:

$$A - B = 0, \quad A + B = 0.$$

Для получения полного квадрата в левой части уравнения (1.2), необходимо добавить $\frac{b_2^2}{4} - b_1 T_n - b_0$ к обоим частям выражения, получая полный квадрат в левой части.

$$\left(T_n^2 + \frac{b_2}{2}\right)^2 = \frac{b_2^2}{4} - b_1 T_n - b_0. \quad (2.1)$$

Левая часть полученного выражения остается полным квадратом, если к $T_n^2 + \frac{b_2}{2}$ внутри скобок прибавить t . Добавляем выражение $2t\left(T_n^2 + \frac{b_2}{2}\right)$ к обеим частям выражения (2.1). Затем, правую часть уравнения (2.1) следует выразить в виде квадратичной формы стандартного вида для T_n :

$$\left(T_n^2 + \frac{b_2}{2} + m\right)^2 = 2mT_n^2 - b_1 T_n + \left(m^2 + b_2 m + \frac{b_2^2}{4} - b_0\right). \quad (2.2)$$

Данное уравнение справедливо для всех значений m .

Выражение (2.2) верно для всех значений m и квадратичное выражение в правой части представляет собой полный квадрат, в том случае, если его дискриминант равен нулю (где дискриминант квадратного уравнения вида: $ax^2 + bx + c = 0$ равен $b^2 - 4ac$). Допустим, что дискриминант равен нулю. В таком случае, получаем следующее выражение:

$$\begin{aligned} (-b_1)^2 - 4(2m)\left(m^2 + b_2 m + \frac{b_2^2}{4} - b_0\right) = \\ -8m^3 - 8b_2 m^2 - 2(b_2^2 - 4b_0)m + b_1^2 = 0. \end{aligned}$$

Данное выражение разделим на -8 , получив резольвентное кубическое уравнение Феррари:

$$m^3 + b_2 m^2 + \left(\frac{b_2^2}{4} - b_0\right)m - \frac{b_1^2}{8} = 0. \quad (2.3)$$

Любое решение m уравнения (2.3) приводит к тому, что правая часть выражения (2.2) становится полным квадратом B^2 . Для избежания вычислений с комплексными числами необходимо выбирать любое неотрицательное решение m , т.к. постоянный коэффициент $-\frac{b_1^2}{8}$ меньше или равен нулю и такое решение всегда существует:

$$m \geq 0.$$

Если m — неотрицательное вещественное решение, то уравнение (2.2) принимает требуемый вид $A^2 = B^2$:

$$\left(T_n^2 + \frac{b_2}{2} + m\right)^2 = (\sqrt{2m}T_n - R)^2, \quad (2.4)$$

$$A = T_n^2 + \frac{b_2}{2} + m, \quad B = \sqrt{2m}T_n - R, \quad (2.5)$$

$$R^2 = m^2 + b_2m + \frac{b_2^2}{4} - b_0, \quad (2.6)$$

$$2\sqrt{2m}R = b_1. \quad (2.7)$$

Уравнения (2.6) и (2.7) позволяют получить два различных решения для переменной R . Уравнение (2.7) демонстрирует, что если $m > 0$, то знак R должен совпадать со знаком b_1 . В случае, когда $m = 0$, знак R может быть произвольным в диапазоне в (2.4) – (2.7). Тем не менее, если R отрицательно, это может стать удобным решением в будущем, подробнее об этом сказано в выводе формулы Z_n для случая, когда $m = 0$. Исходя из сказанного, определяем следующую функцию:

$$\Sigma = \begin{cases} 1, & b_1 > 0 \\ -1, & \text{иначе} \end{cases}. \quad (2.8)$$

Для вышеупомянутых переменных можно получить следующие выражения:

$$b_1 = \Sigma|b_1|, \quad R = \Sigma|R|. \quad (2.9)$$

Решив уравнение (2.7) относительно R , получим:

$$R = \frac{b_1}{2\sqrt{2m}}, \quad m > 0. \quad (2.10)$$

Из уравнений (2.6) и (2.9) следует, что

$$R = \Sigma \sqrt{m^2 + b_2m + \frac{b_2^2}{4} - b_0}. \quad (2.11)$$

При $m > 0$, R будет являться вещественным числом, следовательно, подкоренное выражение из уравнения (2.11) должно быть неотрицательной величиной.

$$m^2 + b_2 m + \frac{b_2^2}{4} - b_0 \geq 0, \quad m > 0. \quad (2.12)$$

С использованием R из (2.10) или (2.11) квадратное уравнение (2.4) разлагается на два квадратных уравнения $A - B = 0$ и $A + B = 0$, используя A и B из (2.5).

$$T_n^2 - \sqrt{2m}T_n + \frac{b_2}{2} + m + R = 0, \quad T_n^2 + \sqrt{2m}T_n + \frac{b_2}{2} + m - R = 0. \quad (2.13)$$

Решением данных уравнений будут следующие выражения:

$$T_{1,2} = \sqrt{\frac{m}{2}} \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} - R}, \quad T_{3,4} = -\sqrt{\frac{m}{2}} \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} + R}. \quad (2.14)$$

Отсюда, применяя преобразование $Z_n = T_n + C$, получаем решение общего уравнения четвертой степени с использованием модификаций Уолтерса [1]:

$$\begin{aligned} Z_{1,2} &= \sqrt{\frac{m}{2}} - C \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} - R}, \\ Z_{3,4} &= -\sqrt{\frac{m}{2}} - C \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} + R}. \end{aligned} \quad (2.15)$$

Для получения решения уравнения четвертой степени общим алгоритмом Феррари выполняется подстановка из уравнения (2.10), что дает значение R и решение уравнения при условии $m > 0$. Полученное решение представляется в виде:

$$Z_{1,2} = \sqrt{\frac{m}{2}} - C \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} - \frac{b_1}{2\sqrt{2m}}},$$

$$Z_{3,4} = -\sqrt{\frac{m}{2}} - C \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} + \frac{b_1}{2\sqrt{2m}}}. \quad (2.16)$$

Рассматривается случай, когда $m = 0$. Уравнение (2.7) показывает, что b_1 также должен быть равен нулю. В таком случае подавленное уравнение четвертой степени (1.2) становится квадратным уравнением относительно T_n^2 , которое имеет решения:

$$T_n^2 = -\frac{b_2}{2} \pm \sqrt{\frac{b_2^2}{4} - b_0}.$$

Применяя преобразование $Z_n = T_n + C$ и извлекая квадратный корень от обеих частей, получаем решение уравнения общего алгоритма Феррари для случая, когда $m = 0$:

$$Z_{1,2} = -C \pm \sqrt{-\frac{b_2}{2} - \sqrt{\frac{b_2^2}{4} - b_0}},$$

$$Z_{3,4} = -C \pm \sqrt{-\frac{b_2}{2} + \sqrt{\frac{b_2^2}{4} - b_0}}. \quad (2.17)$$

Полученные формулы соответствуют результатам, полученным с учетом модификаций Уолтерса [1] (2.15) для $m = 0$. Уравнения (2.7), (2.8) и (2.11) показывают, что $m = 0 \Rightarrow b_1 = 0 \Rightarrow \Sigma = -1 \Rightarrow R = -\sqrt{\frac{b_2^2}{4} - b_0} \Rightarrow Z_n$ формулы из (2.15) при условии, что $m = 0$ действительно имеют место, что подтверждает соответствие с формулами из (2.17).

Если при использовании формулы (2.17) для вычисления решений Z_n подынтегральная функция $\frac{b_2^2}{4} - b_0$ меньше нуля, то возникает необходимость извлечения квадратных корней из комплексных чисел. Для избежания данной ситуации, следует использовать формулу (2.17) только в том случае, если наибольшее вещественное решение резольвентного кубического уравнения $m = 0$. Если существует какое-то другое вещественное решение $m > 0$, то его следует использовать в (2.15) или (2.16) для вычисления общего решения Z_n .

Рассматриваем случай, когда максимальное действительное решение кубического резольвентного уравнения (2.3) $m = 0$. Из предыдущих рассуждений следует, что при $m = 0 \Rightarrow b_1 = 0$. Следовательно, резольвентное уравнение (2.3) записано в виде:

$$m^3 + b_2 m^2 + \left(\frac{b_2^2}{4} - b_0 \right) m = m \left(m^2 + b_2 m + \frac{b_2^2}{4} - b_0 \right) = 0,$$

решениями, которого являются $m = 0$ и $m = -\frac{b_2}{2} \pm \sqrt{b_0}$. В случае, когда $b_0 \geq 0$, два ненулевых решения будут вещественными, и предполагается, что они меньше или равны нулю. Если $b_0 \geq 0$, тогда $-\frac{b_2}{2} + \sqrt{b_0} \leq 0 \Rightarrow \sqrt{b_0} \leq \frac{b_2}{2} \Rightarrow b_0 \leq \frac{b_2^2}{4} \Rightarrow \frac{b_2^2}{4} - b_0 \geq 0$. В случае, когда b_0 отрицательно, предполагаем, что $\frac{b_2^2}{4} - b_0 \geq 0$. Следовательно, когда не существует вещественного решения $m > 0$:

$$\frac{b_2^2}{4} - b_0 \geq 0. \quad (2.18)$$

Этот результат гарантирует, что уравнение (2.17) работает исключительно с вещественными числами, если максимальное действительное решение кубического резольвентного уравнения (2.3) m равно нулю. Алгоритм с учетом модификаций Уолтерса [1] может быть применен к выражению (2.18). Подкоренное выражение R в уравнении (2.11) равно $\frac{b_2^2}{4} - b_0$, при условии, что $m = 0$. Таким образом, уравнения (2.11), (2.12) и (2.18) свидетельствуют о том,

что подкоренное выражение R неотрицателено при условии, что решение $m > 0$ является вещественным и существует.

$$m^2 + b_2 m + \frac{b_2^2}{4} - b_0 \geq 0. \quad (2.19)$$

Алгоритм Феррари имеет неустойчивость вычислений решения m резольвентного кубического уравнения, которое может стремиться к нулю, что приводит к неопределенностям на выходе, в связи с ошибками округления. Например, когда вычисленное значение m , которое содержит ошибку округления и постоянная величина b_1 , которая ее не содержит, стремится к нулю, то ошибка вычисления сильно влияет на m , отсюда алгоритм начинает нестабильно работать. В связи с этим, Уолтерс [1] предлагает свой вариант, в котором заменяет $\frac{b_1}{2\sqrt{2m}}$ на R . Кроме того, добавляя $\frac{b_2^2}{8}$ к обеим частям резольвентного кубического уравнения (2.3), разделив полученное выражение на m и возведя все в квадратный корень, убеждаемся в достоверности произведенной замены. Данный подход способствует устраниению неустойчивости вычислений и повышению точности найденного решения.

Основные рассмотренные моменты для алгоритма Феррари таковы:

- Для решения подавленного уравнения четвертой степени (1.2), заданного коэффициентами b_2, b_1 и b_0 , необходимо перейти к решению резольвентного кубического уравнения (2.3). Далее, полученные решения рассматриваются и используется решение $m > 0$, если оно существует. В противном случае, принимается $m = 0$.
- Для поиска корней с помощью стандартного алгоритма Феррари, используем (2.16), если $m > 0$, или (2.17), когда $m = 0$. При этом, неравенство (2.18) гарантирует, что (2.17) работает исключительно с вещественными числами.
- Для поиска корней с помощью алгоритма Феррари с модификациями Уолтерса, определяем Σ используя (2.8), R с помощью (2.11) и Z_n из (2.15).

При использовании существующего вещественного $m > 0$, неравенство (2.19) гарантирует, что R вещественно и что (2.15) работает исключительно с вещественными величинами.

Исходные тексты программной имплементации алгоритма Феррари приведены в Приложении (см. файл polynomial.cpp).

Алгоритм Декарта

В данном пункте рассматриваются две вариации алгоритма Декарта с учетом модификаций Уолтерса [1] и без [3]. Оба алгоритма имеют схожую логику с алгоритмом Феррари. Модифицированный алгоритм Декарта, преследует те же идеи, что и модифицированный алгоритм Феррари, а именно модификация способствует избавлению от нестабильных решений y^2 . Формулы алгоритма Феррари для решения уравнения Z_n выводятся из формул Декарта, путем замены $\frac{y^2}{2}$ на m и положительного y на $\sqrt{y^2}$. Для этого в кубическом уравнении Феррари (2.3) заменяются $\frac{y^2}{2}$ на m и полученное выражение умножается на 8, получая кубическое уравнение Декарта вида:

$$y^6 + 2b_2y^4 + (b_2^2 - 4b_0)y^2 - b_1^2 = 0. \quad (3.1)$$

Рассматриваем случай, когда $m = \frac{y^2}{2} > 0$, квадратные уравнения из (2.13) и их решения (2.14) из алгоритма Феррари преобразованы в уравнения для алгоритма Декарта путем присваивания R выражения из (2.10), замены $\frac{y^2}{2}$ на m и положительного y на $\sqrt{y^2}$. Таким образом, полученные уравнения и их решения представляются в следующем виде:

$$\begin{aligned} T_n^2 - yT_n + \frac{y + b_2 + \frac{b_1}{y}}{2} &= 0, \\ T_n^2 + yT_n + \frac{y + b_2 - \frac{b_1}{y}}{2} &= 0, \\ (y^2 > 0). \end{aligned} \quad (3.2)$$

$$T_{1,2} = \frac{y}{2} \pm \sqrt{-\frac{y^2}{4} - \frac{b_2}{2} - \frac{b_1}{2y}},$$

$$T_{3,4} = -\frac{y}{2} \pm \sqrt{-\frac{y^2}{4} - \frac{b_2}{2} + \frac{b_1}{2y}},$$

$$(y^2 > 0).$$

Полученные формулы являются решением подавленного уравнения четвертой степени.

$$T_n^4 + b_2 T_n^2 + b_1 T_n + b_0 = 0, \quad n = 1, 2, 3, 4. \quad (3.3)$$

Вместо поиска решений T_n , использовано утверждение Декарта о том, что решения T_n двух квадратных уравнений (3.2) совпадают с четырьмя решениями подавленного уравнения четвертой степени (3.3), при условии, что y^2 является положительным вещественным решением резольвентного кубического уравнения (3.1) [3].

Для подтверждения данного утверждения было составлено произведение двух квадратных уравнений (3.2).

$$\left[T_n^2 - yT_n + \frac{y + b_2 + \frac{b_1}{y}}{2} \right] \left[T_n^2 + yT_n + \frac{y + b_2 - \frac{b_1}{y}}{2} \right] = 0.$$

Раскрыв скобки и упростив полученное выражение, получаем:

$$T_n^4 + b_2 T_n^2 + b_1 T_n + \frac{y^4 + 2b_2 y^2 + b_2^2 - \frac{b_1^2}{y^2}}{4}. \quad (3.4)$$

где y^2 — положительное вещественное решение уравнения (3.1). Стоит отметить, что уравнения (3.3) и (3.4) отличаются лишь постоянными коэффициентами. К обеим частям резольвентного кубического уравнения (3.1) прибавляем $4b_0 y^2$ и делим полученное выражение на $4y^2$:

$$\frac{y^4 + 2b_2y^2 + b_2^2 - \frac{b_1^2}{y^2}}{4} = b_0.$$

Результат выражается в равенстве между постоянными коэффициентами уравнений (3.4) и (3.3). Это означает, что уравнение (3.4), полученное в результате произведения двух квадратных уравнений (3.2), является подавленным уравнением четвертой степени (3.3). Таким образом, совпадение решений двух квадратных уравнений с четырьмя решениями подавленного уравнения четвертой степени, возможно при условии, что y^2 является положительным действительным решением резольвентного кубического уравнения (3.1). Это обосновывает утверждение Декарта.

Исходные тексты программной имплементации алгоритма Декарта приведены в Приложении (см. файл polynomial.cpp).

Алгоритм NBS

В данном пункте рассматривается реализация и структура алгоритма NBS [1]. Данный алгоритм принимает на вход четыре вещественных коэффициента A_3, A_2, A_1 и A_0 , и на выходе рассчитывает четыре величины Z_1, Z_2, Z_3 и Z_4 , удовлетворяющие уравнению:

$$Z^4 + A_3Z^3 + A_2Z^2 + A_1Z + A_0 = (Z - Z_1)(Z - Z_2)(Z - Z_3)(Z - Z_4), \forall Z. \quad (4.1)$$

Таким образом, на выходе алгоритма вычисляются четыре вещественных корня исходного уравнения четвертой степени, заданного уравнением:

$$Z_n^4 + A_3Z_n^3 + A_2Z_n^2 + A_1Z_n + A_0 = 0.$$

Пункт разделен на две части, в каждой из которых были проведены следующие действия. Сначала выводятся расчетные уравнения алгоритма. Далее показываем, что алгоритм требует использования наибольшего вещественного решения резольвентного кубического уравнения, гарантируя работу алгоритма исключительно с вещественными числами.

Приступаем к выводу расчетных уравнений. Метод NBS использует следующий подход для решения уравнения четвертой степени. Левая часть уравнения (4.1) выражается как произведение двух квадратичных выражений с вещественными коэффициентами:

$$(Z^2 + p_1 Z + q_1)(Z^2 + p_2 Z + q_2) = Z_n^4 + A_3 Z_n^3 + A_2 Z_n^2 + A_1 Z + A_0. \quad (4.2)$$

Коэффициенты p_1 , p_2 , q_1 и q_2 выражаются из коэффициентов A_3 , A_2 , A_1 и A_0 , после чего находится решение квадратного уравнения в виде корней двух квадратичных выражений:

$$\begin{aligned} Z_{1,2} &= -\frac{p_1}{2} \pm \sqrt{\frac{p_1^2}{4} - q_1}, \\ Z_{3,4} &= -\frac{p_2}{2} \pm \sqrt{\frac{p_2^2}{4} - q_2}. \end{aligned} \quad (4.3)$$

Коэффициенты p_1 , p_2 , q_1 и q_2 находятся путем раскрытия скобок и упрощения выражения в левой части (4.2):

$$\begin{aligned} Z^4 + (p_1 + p_2)Z^3 + (p_1 p_2 + q_1 + q_2)Z^2 + (p_1 q_2 + p_2 q_1)Z \\ = Z_n^4 + A_3 Z_n^3 + A_2 Z_n^2 + A_1 Z + A_0. \end{aligned}$$

Это приводит к следующей системе уравнений:

$$p_1 + p_2 = A_3, \quad (4.4)$$

$$p_1 p_2 + q_1 + q_2 = A_2, \quad (4.5)$$

$$p_1 q_2 + p_2 q_1 = A_1, \quad (4.6)$$

$$q_1 q_2 = A_0. \quad (4.7)$$

Далее, положим, что $u = q_1 + q_2$, тогда уравнение (4.5) решается относительно $p_1 p_2$:

$$q_1 + q_2 = u, \quad (4.8)$$

$$p_1 p_2 = A_2 - u. \quad (4.9)$$

Для выражения функций p_1 , p_2 , q_1 и q_2 через переменную u учитываем следующее: если даны сумма S и произведение P двух неизвестных величин x_1 и x_2 , то эти находятся как решения квадратного уравнения вида $x_m^2 - Sx_m + P = 0$, $m = 1, 2$. Уравнения (4.4) и (4.9) определяют A_3 и $A_2 - u$ как сумму и произведение p_1 и p_2 . Также уравнения (4.7) и (4.8) определяют u и A_0 как сумму и произведение q_1 и q_2 . Отсюда, квадратные уравнения для p_1 и p_2 и для q_1 и q_2 имеют следующий вид:

$$p_m^2 - A_3 p_m + A_2 - u = 0, \quad q_m^2 - u q_m + A_0 = 0, \quad m = 1, 2.$$

Для решения первого уравнения определяем отрицательный радикал для p_1 :

$$p_1 = \frac{A_3}{2} - \sqrt{\frac{A_3^2}{4} + n - A_2}, \quad p_2 = \frac{A_3}{2} + \sqrt{\frac{A_3^2}{4} + n - A_2}. \quad (4.10)$$

На данном этапе неизвестно, какой из q_1 или q_2 имеет положительный радикал, а какой – отрицательный. Предположим, что Σ_g может принимать значение либо 1, либо -1 , тогда

$$q_1 = \frac{u}{2} + \Sigma_g \sqrt{\frac{u^2}{4} - A_0}, \quad q_2 = \frac{u}{2} - \Sigma_g \sqrt{\frac{u^2}{4} - A_0}. \quad (4.11)$$

Из уравнений (4.4) и (4.9) видно, что подкоренное выражение из (4.10) неотрицательно:

$$\frac{A_3^2}{4} + u - A_2 = \frac{(p_1 + p_2)^2}{4} - p_1 p_2 = \frac{(p_1 - p_2)^2}{4} \geq 0.$$

Из уравнений (4.7) и (4.8) видно, что подкоренное выражение из (4.11) неотрицательно:

$$\frac{u^2}{4} - A_0 = \frac{(q_1 + q_2)^2}{4} - q_1 q_2 = \frac{(q_1 - q_2)^2}{4} \geq 0.$$

Путем подстановки уравнений (4.10) и (4.11) в уравнение (4.6) и упрощения полученных выражений, получаем выражение, с помощью которого определяется Σ_g :

$$2\Sigma_g \sqrt{\left(\frac{A_3^2}{4} + u - A_2\right)\left(\frac{u^2}{4} - A_0\right)} = A_1 - A_3 \frac{u}{2}. \quad (4.12)$$

Отсюда видно, что радикал неотрицателен, а значит, Σ_g соответствует знаку выражения в правой части уравнения. Исходя из этого, Σ_g выражено следующим образом:

$$\Sigma_g = \begin{cases} 1, & A_1 - A_3 \frac{u}{2} > 0 \\ -1, & \text{иначе} \end{cases}. \quad (4.13)$$

Возводим обе части уравнения (4.12) в квадрат и упрощаем полученное выражение до кубического уравнения, получив резольвентное кубическое уравнение относительно u :

$$u^3 - A_2 u^2 + (A_1 A_3 - 4A_0)u + 4A_0 A_2 - A_1^2 - A_0 A_3^2 = 0. \quad (4.14)$$

Из полученного уравнения следует, что существует по меньшей мере одно вещественное решение уравнения u_1 . Это решение используется во всех расчетных формулах, связанных с переменной u . Если уравнение имеет несколько вещественных решений, то выбирается наибольшее решение u_1 , обеспечивая выполнение последующих вычислений только с использованием вещественных чисел.

Основные рассмотренные моменты для алгоритма NBS таковы:

- Исходя из вещественных коэффициентов A_0, A_1, A_2 и A_3 общего уравнения четвертой степени, было решено резольвентное кубическое уравнение (4.14). Было выбрано наибольшее вещественное решение u_1 и определено как переменная u , которая применяется в последующих расчетах.
- Σ_g было определено на основе (4.13).
- Коэффициенты p_1, p_2, q_1 и q_2 были рассчитаны с использованием формул (4.10) и (4.11).
- Решения Z_1, Z_2, Z_3 и Z_4 уравнения четвертой степени были найдены благодаря формулам из (4.3).

Для гарантированной работы алгоритма исключительно с вещественными числами используется наибольшее вещественное решение резольвентного кубического уравнения. Таким образом, удостоверяемся в вещественности решения резольвентного кубического уравнения u_1 и коэффициентов p_1, p_2, q_1 и q_2 . Резольвентное кубическое уравнение (4.14) на выходе дает три решения относительно переменной u , которая определена как (4.8). Коэффициенты q_1 и q_2 являются постоянными величинами в двух квадратичных выражениях левой части (4.2). Таким образом, все величины q_m являются произведением двух корней соответствующего квадратичного выражения. Положим, что Z_1 и Z_2 являются корнями первого квадратичного выражения, а Z_3 и Z_4 – корнями второго квадратичного выражения соответственно, тогда получаем $q_1 = Z_1Z_2$, $q_2 = Z_3Z_4$, откуда $u = Z_1Z_2 + Z_3Z_4$. Полученное величина u соответствует лишь одной из возможных пар решений уравнения четвертой степени, где пара Z_1 и Z_2 является решением квадратичного выражения в левой части (4.2). Решение Z_1 также образует пару с решением Z_3 или Z_4 , в качестве корней квадратичного выражения. Таким образом, все четыре решения Z_n уравнения четвертой степени имеют три возможные комбинации пар, приводящие к трем соответствующим значениям u :

$$u_1 = Z_1Z_2 + Z_3Z_4, \quad u_2 = Z_1Z_3 + Z_2Z_4, \quad u_3 = Z_1Z_4 + Z_2Z_3. \quad (4.15)$$

Наблюдается зависимость между величинами u и парами Z_n , что приводит к различным значениям коэффициентов p_1, p_2, q_1 и q_2 . Для различных значений u выполняются следующие равенства:

$$\begin{aligned} u_1 \Rightarrow p_1 &= -(Z_1 + Z_2), & q_1 &= Z_1 Z_2, \\ p_2 &= -(Z_3 + Z_4), & q_2 &= Z_3 Z_4, \end{aligned} \quad (4.16)$$

$$\begin{aligned} u_2 \Rightarrow p_1 &= -(Z_1 + Z_3), & q_1 &= Z_1 Z_3, \\ p_2 &= -(Z_2 + Z_4), & q_2 &= Z_2 Z_4, \end{aligned} \quad (4.17)$$

$$\begin{aligned} u_3 \Rightarrow p_1 &= -(Z_1 + Z_4), & q_1 &= Z_1 Z_4, \\ p_2 &= -(Z_2 + Z_3), & q_2 &= Z_2 Z_3. \end{aligned} \quad (4.18)$$

Значения p_m и q_m являются вещественными только в случае, когда корни Z_n квадратичного выражения также являются вещественными или комплексно-сопряженной парой. При этом, хотя бы одно из значений u обеспечивает вещественность для p_m и q_m . Определяем u_1 как основное значение для u , и решения Z_1 и Z_2 как корни первого квадратичного выражения в левой части (4.2). Далее выводим алгоритм, который всегда определяет правильное значение u для u_1 . Если все четыре решения Z_n вещественны, то любое из трех значений u_k является подходящим выбором для u_1 , поскольку все u_k вещественны и обеспечивают вещественность для p_m и q_m как вещественных величин. Если все четыре решения Z_n не являются действительными числами, то возможны два случая. В первом случае два решения Z_n являются вещественными числами, а другие два являются комплексно-сопряженной парой. Во втором случае все четыре решения Z_n состоят из двух комплексно-сопряженных пар. В таких случаях значения u_2 и u_3 являются вещественными числами, но соответствующие значения p_m и q_m не являются таковыми. Для представления общего решения Z_n используется сумма его вещественной и мнимой частей: $Z_n = X_n + iY_n$.

Рассматриваем случай, когда два решения Z_n являются вещественными величинами, а остальные две величины являются комплексно-сопряженной

парой. Пусть действительные значения Z_n являются корнями первого квадратного уравнения. Отсюда получаем вывод для каждого Z_n , а именно $Z_1 = X_1, Z_2 = X_2, Z_3 = X_3 + iY_3$ и $Z_4 = X_3 - iY_3$, где $Y_3 > 0$. Решение u_1 вещественно, соответствующие ему значения p_m и q_m в уравнении (4.16) также являются вещественными величинами. Если $X_1 \neq X_2$, то u_2 и u_3 имеют ненулевую мнимую часть, отсюда являются комплексно-сопряженной парой. Подобные комплексные величины u исключаются из рассмотрения, и выбирается вещественное значение u_1 . Если $X_1 = X_2$, то все три значения u являются вещественными, но при этом $u_1 > u_2$ и $u_1 > u_3$. Исходя из выше сказанного получаем следующие выражения для u_1, u_2 и u_3 :

$$u_1 = X_1^2 + X_3^2 + Y_3^2, \quad u_2 = u_3 = 2X_1X_3,$$

$$u_1 = X_1^2 + X_3^2 + Y_3^2 > X_1^2 + X_3^2 = (X_1 - X_3)^2 + 2X_1X_3 \geq 2X_1X_3 = u_2 = u_3,$$

$$u_1 > u_2 = u_3.$$

Несмотря на то, что u_2 и u_3 являются вещественными, избегаем работы с данными величинами поскольку p_m и q_m , являются комплексными величинами исходя из (4.17) и (4.18). Значение p_1 из (4.17) выражается как:

$$p_1 = -(Z_1 + Z_2) = -(X_1 + X_3 + iY_3) = -(X_1 + X_3) - iY_3, \quad Y_3 > 0.$$

Используя наибольшее вещественное решение резольвентного кубического уравнения u_1 , избавляемся от комплексности значений p_m и q_m . Данное решение позволяет работать с вещественными величинами, избегая сложной работы с комплексными вычислениями. Рассматриваем случай, когда все решения уравнения четвертой степени Z_n образуются из двух комплексно-сопряженных пар. В таком случае, каждое из четырех решений Z_n выражается как: $Z_1 = X_1 + iY_1, Z_2 = X_1 - iY_1, Z_3 = X_3 + iY_3$ и $Z_4 = X_3 - iY_3$, где $Y_1 > 0$ и $Y_3 > 0$.

Значения u (4.16) имеют следующее представление:

$$u_1 = X_1^2 + Y_1^2 + X_3^2 + Y_3^2, \quad u_2 = 2(X_1X_3 - Y_1Y_3), \quad u_3 = 2(X_1X_3 + Y_1Y_3).$$

Все рассматриваемые величины u являются вещественными. При этом, величина u_1 не меньше u_2 и u_3 .

$$\begin{aligned} u_1 &= X_1^2 + Y_1^2 + X_3^2 + Y_3^2 = \\ (X_1 - X_3)^2 + 2X_1X_3 + (Y_1 - Y_3)^2 + 2Y_1Y_3 &\geq 2X_1X_3 + 2Y_1Y_3 = \\ u_3 &> 2X_1X_3 - 2Y_1Y_3 = u_2, \\ u_1 &\geq u_3 > u_2. \end{aligned}$$

Согласно выкладкам из (4.16), (4.17) и (4.18), значения p_m и q_m , соответствующие u_1 , являются вещественными числами, в то время как значения, соответствующие u_2 , являются комплексными числами. В случае, если $Z_1 = Z_3$, значения p_m и q_m , соответствующие u_3 , также являются комплексными величинами. При выборе u_1 в качестве наибольшего из трех вещественных решений резольвентного кубического уравнения, не наблюдается появления комплексных величин p_m и q_m . Именно поэтому выбор u_1 в качестве наибольшего действительного решения гарантирует алгоритму последующую вычислительную работу исключительно с вещественными числами.

Исходные тексты программной имплементации алгоритма NBS приведены в Приложении (см. файл polynomial.cpp).

Алгоритм Эйлера

В данном пункте рассматриваются принципы работы и структура алгоритма, представленного Эйлером для поиска корней уравнения четвертой степени. Для начала алгоритм решает подавленное уравнение четвертой степени:

$$T_n^4 + b_2 T_n^2 + b_1 T_n + b_0 = 0, \quad n = 1, 2, 3, 4. \quad (5.1)$$

Эйлер вывел утверждение [4], о том, что решение уравнения четвертой степени может быть представлено в виде выражения:

$$T_n = \sqrt{r_1} + \sqrt{r_2} + \sqrt{r_3}, \quad (5.2)$$

где r_1 , r_2 и r_3 – решения кубического уравнения с вещественными коэффициентами, задаваемого уравнением:

$$r_k^3 + a_2 r_k^2 + a_1 r_k + a_0 = 0. \quad (5.3)$$

Уравнение четвертой степени, имеющее решение (5.2), является подавленным уравнением четвертой степени (5.1), коэффициенты которого b_n зависят от a_m в уравнении (5.3). Путем выражения этих зависимостей можно получить значения a_m из b_n . Кубическое уравнение (5.3), коэффициенты которого выражены через b_n , является резольвентным кубическим уравнением. Его решения r_1 , r_2 и r_3 позволяют вычислить T_n с помощью выражения (5.2). Метод Эйлера не использует принцип главного значения квадратного корня подкоренных выражений, описанный в ранее в соответствующем пункте. Эйлер рассматривает каждое из полученных решений r_k , беря квадратный корень полученной величины $\sqrt{r_k}$ из выражения (5.2). Для каждого из трех возможных значений r_k выводятся два квадратных корня, отсюда получаем восемь комбинаций квадратных корней (5.2) для восьми возможных величин T_n . Отсюда следует, что лишь четыре из полученных восьми комбинаций являются допустимыми для любого заданного коэффициента b_1 из уравнения (5.1). Процесс вычисления начинается с определения r_k из уравнения (5.3).

$$(r - r_1)(r - r_2)(r - r_3) = r_k^3 + a_2 r_k^2 + a_1 r_k + a_0, \forall r.$$

Левая часть полученного выражения разлагается и упрощается. Соответствующие коэффициенты двух частей уравнения приравниваются друг к другу, в итоге получая:

$$-(r_1 + r_2 + r_3) = a_2, \quad (5.4)$$

$$r_1 r_2 + r_1 r_3 + r_2 r_3 = a_1, \quad (5.5)$$

$$-r_1 r_2 r_3 = a_0. \quad (5.6)$$

Определяем уравнение четвертой степени, решения которого даны в (5.2). Производим возвведение (5.2) в квадрат. Применяем (5.4) и группируем полученное выражение, получив:

$$T_n^2 = r_1 + r_2 + r_3 + 2\sqrt{r_1 r_2} + 2\sqrt{r_1 r_3} + 2\sqrt{r_2 r_3},$$

$$T_n^2 + a_2 = 2(\sqrt{r_1 r_2} + \sqrt{r_1 r_3} + \sqrt{r_2 r_3}).$$

Далее возводим обе части полученного выражения в квадрат, получая выражение вида:

$$T_n^4 + 2a_2 T_n^2 + a_2^2 = 4(r_1 r_2 + r_1 r_3 + r_2 r_3) + 8\sqrt{r_1}\sqrt{r_2}\sqrt{r_3}(\sqrt{r_1} + \sqrt{r_2} + \sqrt{r_3}).$$

Применяя (5.5) и (5.2) и группируя полученное выражение, получаем выражение в общем виде:

$$T_n^4 + 2a_2 T_n^2 + a_2^2 = 4a_1 + 8\sqrt{r_1}\sqrt{r_2}\sqrt{r_3}T_n,$$

$$T_n^4 + 2a_2 T_n^2 - 8\sqrt{r_1}\sqrt{r_2}\sqrt{r_3}T_n + a_2^2 - 4a_1 = 0. \quad (5.7)$$

Замечаем, что некоторые корни могут быть комплексными. Однако, в рамках данной работы они не являются значимыми, т.к. в ходе исследования представлены лишь методы поиска вещественных корней полиномиальных уравнений. Поэтому комплексные величины не рассматриваются далее.

Уравнение (5.6) показывает, что:

$$\sqrt{r_1}\sqrt{r_2}\sqrt{r_3} = \sqrt{r_1 r_2 r_3} = \sqrt{-a_0}. \quad (5.8)$$

Уравнение (5.7) принимает вид:

$$T_n^4 + 2a_2 T_n^2 - 8\sqrt{-a_0}T_n + a_2^2 - 4a_1 = 0, \quad (5.9)$$

которое имеет форму (5.1).

Исходя из выше сказанного, получаем резольвентное кубическое уравнение для подавленного уравнения четвертой степени (5.1). Коэффициенты b_n из уравнения (5.1) равны соответствующим коэффициентам из уравнения (5.9). Таким образом, получаем:

$$b_2 = 2a_2, \quad b_1 = -8\sqrt{-a_0}, \quad b_0 = a_2^2 - 4a_1. \quad (5.10)$$

Из полученных равенств выразим коэффициенты a_2 , a_1 и a_0 и подставляем полученные величины в уравнение (5.3).

$$a_2 = \frac{b_2}{2}, \quad a_1 = \frac{b_2^2 - 4b_0}{16}, \quad a_0 = -\frac{b_1^2}{64}. \quad (5.11)$$

Отсюда получаем резольвентное кубическое уравнение Эйлера:

$$r_k^3 + \frac{b_2}{2} r_k^2 + \frac{b_2^2 - 4b_0}{16} r_k - \frac{b_1^2}{64} = 0. \quad (5.12)$$

Постоянный коэффициент a_0 содержит информацию о всех решениях резольвентного уравнения r_1 , r_2 и r_3 . Поскольку значение коэффициента a_0 не превышает нуля, то уравнение имеет как минимум одно положительное вещественное решение, например $r_1 \geq 0$. Уравнения (5.6) и (5.11), относительно a_0 в совокупности показывают, что:

$$r_1 r_2 r_3 = \frac{b_1^2}{64} \geq 0. \quad (5.13)$$

Произведение всех корней резольвентного кубического уравнения является неотрицательной вещественной величиной. Произведение r_2 и r_3 дает неотрицательное вещественное число:

$$r_1 \geq 0 \text{ и } r_1 r_2 r_3 = -\frac{b_1^2}{64} \geq 0 \Rightarrow r_2 r_3 \geq 0.$$

r_2 и r_3 являются либо вещественными решениями, либо образуют комплексно-сопряженную пару решений. В случае, когда данные величины вещественны, то они имеют один и тот же знак.

Важно то, что уравнение (5.12) зависит от модуля величины b_1 , а не от его знака. Отсюда, (5.12) является резольвентным кубическим уравнением для двух уравнений четвертой степени, вида:

$$T_n^4 + b_2 T_n^2 \pm b_1 T_n + b_0 = 0.$$

Исходя из того, что имеется два уравнения четвертой степени, отсюда вытекает вывод, что в общей сумме существует восемь решений T_n (5.2). Выражаем $\sqrt{-a_0}$ через b_1 из (5.10) и подставляем в (5.8), получая:

$$\sqrt{r_1} \sqrt{r_2} \sqrt{r_3} = -\frac{b_1}{8}. \quad (5.14)$$

При условии, что все три радикала удовлетворяют (5.14), уравнение (5.2) дает верное решение T_n . Выбор из двух квадратных корней для двух любых $\sqrt{r_k}$ допускается. Третья величина $\sqrt{r_k}$ выбирается с учетом выполнения (5.14). Требуется лишь проверить, что обе части выражения (5.14) имеют одинаковый знак, т.к. (5.13) гарантирует, что данные значения имеют одинаковую величину по модулю.

$$r_1 r_2 r_3 = \frac{b_1^2}{64} \Rightarrow |\sqrt{r_1} \sqrt{r_2} \sqrt{r_3}| = \left| \frac{b_1}{8} \right|.$$

Если все $\sqrt{r_k}$ удовлетворяют условию (5.14), тогда общее решение уравнения (5.1) имеет следующий вид:

$$T_1 = \sqrt{r_1} + \sqrt{r_2} + \sqrt{r_3}, \quad (5.15)$$

$$T_2 = \sqrt{r_1} - \sqrt{r_2} - \sqrt{r_3}, \quad (5.16)$$

$$T_3 = -\sqrt{r_1} + \sqrt{r_2} - \sqrt{r_3}, \quad (5.17)$$

$$T_4 = -\sqrt{r_1} - \sqrt{r_2} + \sqrt{r_3}. \quad (5.18)$$

Все выражения T_n были выведены верно, т.к. все их члены являются квадратными корнями $\sqrt{r_k}$, а произведение этих членов равно $-\frac{b_1}{8}$, что удовлетворяет (5.14).

Основные рассмотренные моменты для алгоритма Эйлера таковы:

- Проводятся вычисления коэффициентов b_2 , b_1 и b_0 подавленного уравнения четвертой степени (5.1).
- Выполняется поиск решений r_1 , r_2 и r_3 резольвентного кубического уравнения (5.12).
- Для $\sqrt{r_1}$, $\sqrt{r_2}$ и $\sqrt{r_3}$ определяются знаки, удовлетворяющие условию (5.14).
- С помощью выведенных формул (5.15) — (5.18) вычисляются четыре решения T_n исходного уравнения (5.1).

Уолтерс [1] предлагает свою модификацию данного алгоритма, главной целью которой является избавление от работы с комплексными числами, которые иногда возникают в двух решениях резольвентного кубического уравнения. Основные выкладки остаются такими же. Однако, содержательные различия также существуют. Основные формулы для T_n модифицированы с учетом принципа главного значения квадратного корня радикала:

$$T_{1,2} = \sqrt{r_1} \pm (\sqrt{r_2} - \Sigma s \sqrt{r_3}), \quad T_{3,4} = -\sqrt{r_1} \pm (\sqrt{r_2} + \Sigma s \sqrt{r_3}). \quad (5.19)$$

Также, модификация Уолтерса [1] предполагает использование максимального вещественного решения r_1 резольвентного кубического уравнения. Произведение двух значений r_2 и r_3 является неотрицательной вещественной величиной. Исходя из этого, скажем, что $r_2 = x_2 + iy_2$ и $r_3 = x_3 + iy_3$ являются вещественными значениями в том случае, если $y_2 = 0$ и $y_3 = 0$, а комплексно-сопряженной парой тогда, когда $x_2 = x_3$ и $y_2 = -y_3$. В случае, когда

r_2 и r_3 являются вещественными величинами, то они всегда имеют одинаковые знаки. Выше сказанное подразумевает то, что выражение в скобке (5.19) является либо вещественной, либо комплексной величиной.

Дальнейшее преобразование формул заключается в замене каждого выражения в скобках на квадрат радикала:

$$\begin{aligned} T_{1,2} &= \sqrt{r_1} \pm \sqrt{r_2 + r_3 - 2\sum s\sqrt{r_2}\sqrt{r_3}}, \\ T_{3,4} &= -\sqrt{r_1} \pm \sqrt{r_2 + r_3 + 2\sum s\sqrt{r_2}\sqrt{r_3}}. \end{aligned} \quad (5.20)$$

Значения $T_{1,2}$ вычисленные из (5.20) совпадают со значениями $T_{1,2}$, полученными из (5.19), за исключением случаев, когда $\sqrt{r_2} - \sum s\sqrt{r_3}$ является отрицательным вещественным или комплексным значением. Если подобная ситуация имеет место быть, тогда решение T_1 из (5.19) совпадает с решением T_2 из (5.20), а T_2 из (5.19) совпадает с решением T_1 из (5.20) соответственно. Те же рассуждения применяются и для решений $T_{3,4}$, которые зависят от $\sqrt{r_2} + \sum s\sqrt{r_3}$.

Далее Уолтерс [1] предлагает следующую идею: выбираются $r_2 = x_2 + iy_2$ и $r_3 = x_3 + iy_3$ так, чтобы $|x_2| \geq |x_3|$ и $y_2 = -y_3 \geq 0$. Данная идея гарантирует, что выражения в скобках из (5.19) являются неотрицательными вещественными либо комплексными величинами. Подобные рассуждения не влияют на выражения из (5.20) т.к. они симметричны относительно r_2 и r_3 . Благодаря данным действиям, Уолтерс [1] добился того, что решения T_n из (5.19) и (5.20) не меняются между собой, как было описано ранее.

Ссылаясь на полученные результаты, преобразуем выражение $s\sqrt{r_2}\sqrt{r_3}$, которое теперь выводим следующим образом:

$$s\sqrt{r_2}\sqrt{r_3} = s^2 |\sqrt{r_2 r_3}| = |\sqrt{r_2 r_3}| = \sqrt{r_2 r_3}.$$

В конечном итоге получаем формулы решения T_n уравнения четвертой степени для алгоритма Эйлера с учетом модификаций Уолтерса [1]:

$$T_{1,2} = \sqrt{r_1} \pm \sqrt{r_2 + r_3 - 2\sum\sqrt{r_2 r_3}},$$

$$T_{3,4} = -\sqrt{r_1} \pm \sqrt{r_2 + r_3 + 2\sum\sqrt{r_2 r_3}}.$$

Исходные тексты программной имплементации алгоритма Эйлера приведены в Приложении (см. файл polynomial.cpp).

Алгоритм Ван дер Вардена

В данном пункте разобраны структура и методы решения уравнения четвертой степени с помощью алгоритма Ван дер Вардена [5]. Алгоритм рассматривает решения подавленного уравнения вида:

$$T_n^4 + b_2 T_n^2 + b_1 T_n + b_0 = 0, \quad n = 1, 2, 3, 4. \quad (6.1)$$

Выражаем подавленное уравнение (6.1) как произведение двух квадратичных выражений с вещественными коэффициентами, вида:

$$(T^2 + p_1 T + q_1)(T^2 + p_2 T + q_2) = T^4 + b_2 T^2 + b_1 T + b_0. \quad (6.2)$$

Раскрываем скобки и упрощаем левую часть полученного выражения, получая:

$$\begin{aligned} T^4 + (p_1 + p_2)T^3 + (p_1 p_2 + q_1 + q_2)T^2 + (p_1 q_2 + p_2 q_1)T + q_1 q_2 = \\ T^4 + b_2 T^2 + b_1 T + b_0. \end{aligned}$$

Приравниваем между собой коэффициенты из левой части и коэффициенты правой части, получая следующие равенства:

$$p_1 + p_2 = 0, \quad (6.3)$$

$$p_1 p_2 + q_1 + q_2 = b_2, \quad (6.4)$$

$$p_1 q_2 + p_2 q_1 = b_1, \quad (6.5)$$

$$q_1 q_2 = b_0.$$

Пусть θ_k задано как:

$$\theta_k = p_1 p_2. \quad (6.6)$$

Тогда выражение (6.4) выглядит следующим образом:

$$q_1 + q_2 = b_2 - \theta_k.$$

Выражения (6.3) и (6.6) определяются как сумма и произведение величин p_1 и p_2 и равны 0 и θ_k соответственно. Отсюда видно, что решением двух квадратных выражений являются величины p_1 , p_2 , q_1 и q_2 :

$$p_m^2 + \theta_k = 0, \quad q_m^2 - (b_2 - \theta_k)q_m + b_0 = 0, \quad m = 1, 2.$$

Выразим решения:

$$\begin{aligned} p_1 &= -\sqrt{-\theta_k}, & q_1 &= \frac{b_2 - \theta_k + \Sigma' \sqrt{(b_2 - \theta_k)^2 - 4b_0}}{2}, \\ p_2 &= \sqrt{-\theta_k}, & q_2 &= \frac{b_2 - \theta_k - \Sigma' \sqrt{(b_2 - \theta_k)^2 - 4b_0}}{2}, \end{aligned}$$

где Σ' принимает значение 1 либо -1 .

Подставим полученные выражения в (6.5) и упростим полученное выражение:

$$\Sigma' \sqrt{-\theta_k [(b_2 - \theta_k)^2 - 4b_0]} = b_1.$$

Перенесем правую часть влево и возведем полученное выражение в квадрат, получая резольвентное кубическое уравнение Ван дер Вардена:

$$\theta_k^3 - 2b_2\theta_k^2 + (b_2^2 - 4b_0)\theta_k + b_1^2 = 0. \quad (6.7)$$

Все решения θ_1 , θ_2 и θ_3 удовлетворяют следующему равенству:

$$(\theta - \theta_1)(\theta - \theta_2)(\theta - \theta_3) = \theta_k^3 - 2b_2\theta_k^2 + (b_2^2 - 4b_0)\theta_k + b_1^2 = 0, \forall \theta.$$

Раскрываем скобки и упрощаем полученное выражение. Далее приравниваем полученные коэффициенты в левой части к коэффициентам правой части, получая следующие равенства:

$$\theta_1 + \theta_2 + \theta_3 = 2b_2,$$

$$\theta_1\theta_2 + \theta_1\theta_3 + \theta_2\theta_3 = b_2^2 - 4b_0,$$

$$-\theta_1\theta_2\theta_3 = b_1^2.$$

Любое решение θ_k соответствует двум парам состоящих из четырех решений T_n . Каждая из этих пар является решением квадратичных выражений из левой части (6.2). Решением выражения $T^2 + p_1T + q_1$ является пара T_1 и T_2 , а выражения $T^2 + p_2T + q_2$ является пара T_3 и T_4 . Обе пары соответствуют решения θ_1 .

$$(T - T_1)(T - T_2) = T^2 + p_1T + q_1, \quad (T - T_3)(T - T_4) = T^2 + p_2T + q_2,$$

$$p_1 = -(T_1 + T_2), \quad p_2 = -(T_3 + T_4).$$

Исходя из полученных формул для p_1 , p_2 и ранее выведенных уравнений (6.3) и (6.6), можно наблюдать:

$$T_1 + T_2 + T_3 + T_4 = 0, \tag{6.8}$$

$$\theta_1 = (T_1 + T_2)(T_3 + T_4) = -(T_1 + T_2)^2 = -(T_3 + T_4)^2. \tag{6.9}$$

Две другие пары T_1 , T_3 и T_1 , T_4 соответствуют решениям θ_2 и θ_3 .

$$\theta_2 = (T_1 + T_3)(T_2 + T_4) = -(T_1 + T_3)^2 = -(T_2 + T_4)^2, \tag{6.10}$$

$$\theta_3 = (T_1 + T_4)(T_2 + T_3) = -(T_1 + T_4)^2 = -(T_2 + T_3)^2. \tag{6.11}$$

Формулы (6.8) – (6.11) определяют решения T_n подавленного уравнения четвертой степени. Из выражений (6.9) – (6.11) получаем следующие выражения:

$$T_1 + T_2 = -(T_3 + T_4) = \sqrt{-\theta_1},$$

$$T_1 + T_3 = -(T_2 + T_4) = \sqrt{-\theta_2},$$

$$T_1 + T_4 = -(T_2 + T_3) = \sqrt{-\theta_3}.$$

Преобразуем полученные выражения следующим образом, учитывая (6.8), и получаем общее решение уравнения четвертой степени для алгоритма Ван дер Вардена:

$$T_1 = \frac{T_1 + T_2 + T_1 + T_3 + T_1 + T_4}{2} \Rightarrow T_1 = \frac{\sqrt{-\theta_1} + \sqrt{-\theta_2} + \sqrt{-\theta_3}}{2}, \quad (6.12)$$

$$T_2 = \frac{T_1 + T_2 + T_2 + T_4 + T_2 + T_3}{2} \Rightarrow T_2 = \frac{\sqrt{-\theta_1} - \sqrt{-\theta_2} - \sqrt{-\theta_3}}{2}, \quad (6.13)$$

$$T_3 = \frac{T_3 + T_4 + T_1 + T_3 + T_2 + T_3}{2} \Rightarrow T_3 = \frac{-\sqrt{-\theta_1} + \sqrt{-\theta_2} - \sqrt{-\theta_3}}{2}, \quad (6.14)$$

$$T_4 = \frac{T_3 + T_4 + T_2 + T_4 + T_1 + T_4}{2} \Rightarrow T_4 = \frac{-\sqrt{-\theta_1} - \sqrt{-\theta_2} + \sqrt{-\theta_3}}{2}. \quad (6.15)$$

Решения T_n подавленного уравнения четвертой степени удовлетворяют следующему равенству:

$$(T - T_1)(T - T_2)(T - T_3)(T - T_4) = T^4 + T^2 b_2 + Tb_1 + b_0, \forall T.$$

Иначе говоря, все решения T_n удовлетворяют следующим равенствам:

$$\begin{aligned} T_1 + T_2 + T_3 + T_4 &= 0, \\ T_1 T_2 + T_1 T_3 + T_1 T_4 + T_2 T_3 + T_2 T_4 + T_3 T_4 &= b_2, \\ -T_1 T_2 T_3 - T_1 T_2 T_4 - T_1 T_3 T_4 - T_2 T_3 T_4 &= b_1, \\ T_1 T_2 T_3 T_4 &= b_0. \end{aligned} \quad (6.16)$$

Однако, равенство (6.16) выполняется только в том случае, когда:

$$\sqrt{-\theta_1} \sqrt{-\theta_2} \sqrt{-\theta_3} = b_1. \quad (6.17)$$

Основные рассмотренные моменты для алгоритма Ван дер Вардена таковы:

- Было найдено решение θ_1, θ_2 и θ_3 резольвентного кубического уравнения (6.7) с коэффициентами (6.1).
- Выбор знаков для $\sqrt{-\theta_k}$ был определен с учетом выполнения равенства (6.17).
- Были получены формулы (6.12) – (6.15) решений T_n подавленного уравнения четвертой степени (6.1).

Далее рассматриваем модификацию предложенную Уолтерсом для данного алгоритма. Модификация применяет те же самые принципы, описанные в предыдущем пункте, обозревающем алгоритм Эйлера. Уолтерс [1] рассматривает θ_1 как неположительное вещественное решение резольвентного кубического уравнения (6.7). Данное решение существует, т.к. постоянный член $b_1^2 > 0$. В результате $-\theta_1$ является неотрицательным вещественным числом. Перестраиваем формулы для T_n в соответствии с принципом главного значения квадратного корня радикалов, получая новые формулы вида:

$$T_{1,2} = \frac{\sqrt{-\theta_1} \pm (\sqrt{-\theta_2} - 2\sum s \sqrt{-\theta_3})}{2}, T_{3,4} = \frac{-\sqrt{-\theta_1} \pm (\sqrt{-\theta_2} + 2\sum s \sqrt{-\theta_3})}{2}.$$

Ссылаясь на рассуждения и идеи, предложенные в пункте «Алгоритм Эйлера», выразим:

$$s \sqrt{-\theta_2} \sqrt{-\theta_3} = s^2 |\sqrt{\theta_2 \theta_3}| = \sqrt{\theta_2 \theta_3}.$$

В конечном итоге получая формулы для T_n с учетом модификаций Уолтерса [1]:

$$T_{1,2} = \frac{\sqrt{-\theta_1} \pm \left(\sqrt{-\theta_2 - \theta_3 - 2\Sigma\sqrt{\theta_2\theta_3}} \right)}{2},$$

$$T_{3,4} = \frac{-\sqrt{-\theta_1} \pm \left(\sqrt{-\theta_2 - \theta_3 + 2\Sigma\sqrt{\theta_2\theta_3}} \right)}{2}.$$

Исходные тексты программной имплементации алгоритма Ван дер Вардена приведены в Приложении (см. файл polynomial.cpp).

Эквивалентность решений подавляющих алгоритмов

Алгоритмы Феррари, Декарта, Эйлера и Ван дер Вардена являются эквивалентными между собой. В данной главе рассмотрены принципы выражения одного метода через другой, что демонстрируют их эквивалентность. Это иллюстрируется на примере алгоритмов Эйлера и Феррари. В связи с тем, что в данной работе были реализованы соответствующие алгоритмы с учетом модификации Уолтерса, в дальнейшем используются именно эти версии алгоритмов.

Рассматривается резольвентное уравнение Эйлера, решения которого удовлетворяют следующему равенству:

$$(r - r_1)(r - r_2)(r - r_3) = r^3 + \frac{b_2}{2}r^2 + \frac{b_2^2 - 4b_0}{16}r - \frac{b_1^2}{64}, \forall r.$$

Раскрывая скобки в левой части и упрощая полученное выражение, приравниваем соответствующие коэффициенты друг другу, что приводит к выражениям вида:

$$-(r_1 + r_2 + r_3) = \frac{b_2}{2}, \quad r_1r_2 + r_1r_3 + r_2r_3 = \frac{b_2^2 - 4b_0}{16}.$$

Затем выражаем $r_2 + r_3$ и r_2r_3 относительно r_1 .

$$r_2 + r_3 = -r_1 - \frac{b_2}{2}, \quad r_2 r_3 = r_1^2 + \frac{b_2}{2} r_1 + \frac{b_2^2 - 4b_0}{16}.$$

Подставляя полученные выражения в формулу T_n алгоритма Эйлера, модифицированную Уолтерсом, упрощаем полученные формулы и получаем:

$$T_{1,2} = \sqrt{r_1} \pm \sqrt{-r_1 - \frac{b_2}{2} - \Sigma \sqrt{4r_1^2 + 2b_2 r_1 + \frac{b_2^2}{4} - b_0}},$$

$$T_{3,4} = -\sqrt{r_1} \pm \sqrt{-r_1 - \frac{b_2}{2} + \Sigma \sqrt{4r_1^2 + 2b_2 r_1 + \frac{b_2^2}{4} - b_0}}.$$

Из пункта "Резольвентное уравнение" следует, что решения резольвентных уравнений разных методов эквивалентны друг другу в соответствии с (3). Отталкиваясь от этого, можно утверждать, что решение $r_1 = \frac{m}{2}$. В результате применения данной замены, формулы Эйлера преобразуются в формулы Феррари, представленные следующим образом:

$$T_{1,2} = \sqrt{\frac{m}{2}} \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} - R},$$

$$T_{3,4} = -\sqrt{\frac{m}{2}} \pm \sqrt{-\frac{m}{2} - \frac{b_2}{2} + R},$$

где

$$R = \Sigma \sqrt{m^2 + b_2 m^2 + \frac{b_2^2}{4} - b_0}.$$

Согласно предыдущему высказыванию, аналогичная замена используется и для остальных двух методов. Заменяя $\frac{m}{2}$ на $-\frac{\theta}{4}$, получаем формулы Ван дер Вардена,

а заменяя $-\frac{\theta}{4}$ на $\frac{y^2}{4}$, то получены формулы Декарта. Таким образом, было доказано свойство эквивалентности всех полученных подавляющих алгоритмов.

Классификация методов

В ходе данной работы были исследованы и программно имплементированы 11 алгоритмов поиска корней полиномиальный уравнений. Среди представленных, алгоритмы: Феррари [2], Декарта [1, 3], NBS [7], Эйлера [4], Ван дер Вардена [5], Чирнхауса [8], Скуайра [9], FQS [10], Мерримена [11], Ангера [12] и Зальцера [13].

Данные алгоритмы были классифицированы по способу построения вычислений:

- чисто аналитические без использования резольвентных полиномиальных уравнений (алгоритмы Мерримена и Ангера);
- итеративно вычисляющие коэффициенты вспомогательных полиномиальных уравнений, решаемого аналитически (алгоритмы FQS и Скуайра);
- чисто аналитические, нуждающиеся в отыскании всех корней резольвентного полиномиального уравнения (алгоритмы Феррари, Декарта, NBS, Эйлера, Ван дер Вардена и Зальцера);
- чисто аналитические, нуждающиеся в отыскании всех корней резольвентного полиномиального уравнения и в отыскании корней вспомогательных полиномиальных уравнений (алгоритм Чирнхауса).

Аналитический метод без использования резольвентных полиномиальных уравнений

Данный класс методов представляет собой чисто аналитический подход к решению полиномиальных уравнений. Принцип работы методов данного класса представляет собой последовательное выполнение трех этапов:

- 1) обработка входных данных;
- 2) выполнение аналитических выкладок;
- 3) вывод итогового решения.

Принцип работы данного класса методов представлен на следующей схеме:

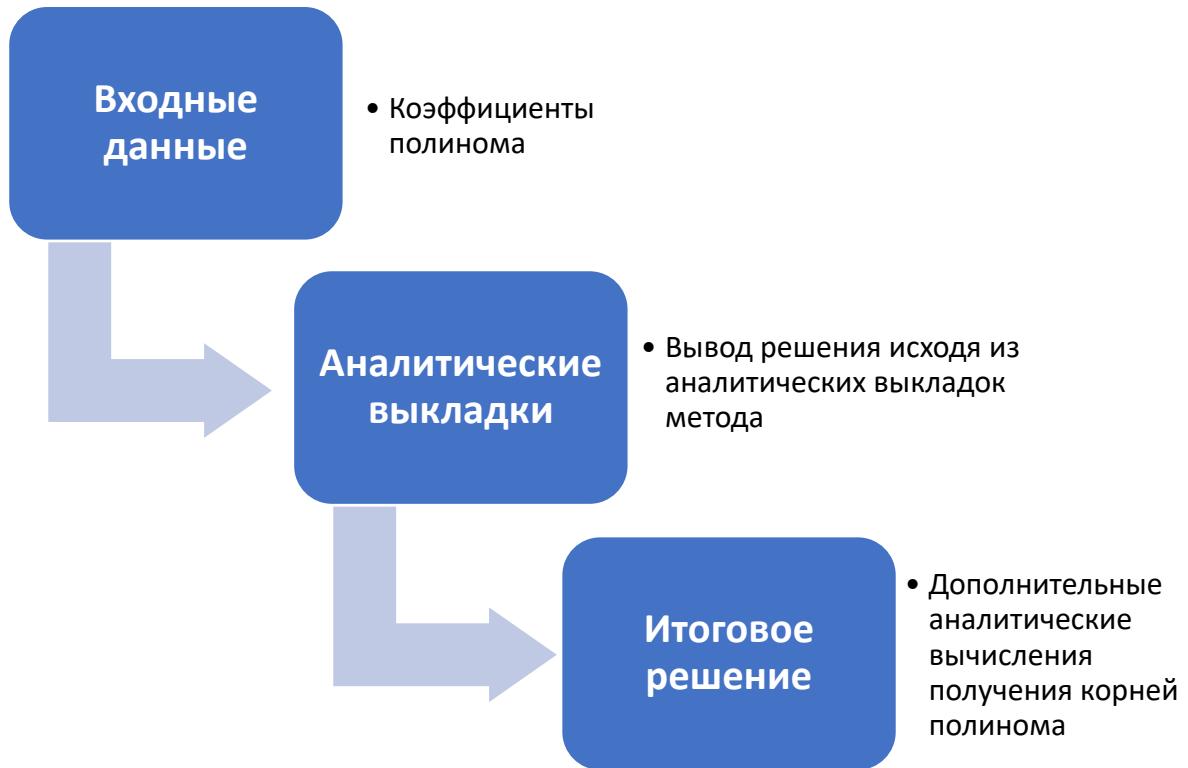


Рисунок 1. Принцип работы алгоритмов использующие чисто аналитический подход.

Метод, итеративно вычисляющий коэффициенты вспомогательных полиномиальных уравнений, решаемых аналитически

Следующий класс методов использует итеративный подход к определению коэффициентов вспомогательных уравнений. Идея работы методов данного класса представляет собой последовательное выполнение следующих этапов:

- 1) обработка входных данных;
- 2) определение коэффициентов вспомогательных уравнений с помощью итеративного метода;
- 3) поиск корней вспомогательных уравнений;
- 4) вывод итогового решения исходя из выполнения дополнительных аналитических выкладок.

Принцип работы данного класса методов представлен на следующей схеме:



Рисунок 2. Принцип работы алгоритмов использующие итеративные методы для определения коэффициентов вспомогательных уравнений.

Аналитический метод, нуждающийся в отыскании всех корней резольвентного полиномиального уравнения

Третий класс методов нуждается в поиске всех корней резольвентного уравнения. Принцип работы методов данного класса представляет собой последовательное выполнение трех этапов:

- 1) обработка входных данных;
- 2) определение коэффициентов резольвентного уравнения и поиск его решения;
- 3) вывод итогового решения исходя из выполнения дополнительных аналитических выкладок.

Принцип работы данного класса методов представлен на следующей схеме:

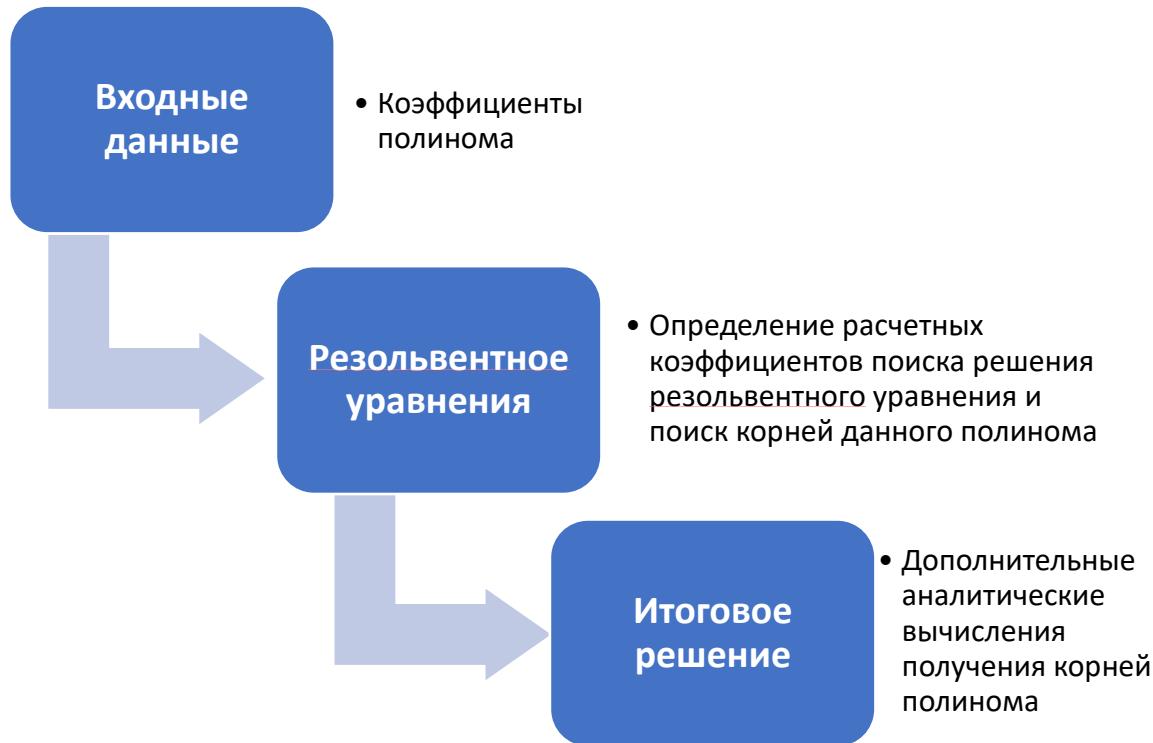


Рисунок 3. Принцип работы алгоритмов требующие отыскания всех корней резольвентного уравнения.

Аналитический метод, нуждающийся в отыскании всех корней резольвентного полиномиального уравнения и в отыскании корней вспомогательных полиномиальных уравнений

Заключающий класс методов требует поиска всех корней резольвентного и вспомогательных уравнений. Идея работы методов данного класса методов схожа с предыдущим классом методов и представляет собой последовательное выполнение следующих этапов:

- 1) обработка входных данных;
- 2) определение коэффициентов резольвентного уравнения и поиск его решения;
- 3) определение коэффициентов вспомогательных уравнений исходя из решения резольвентного уравнения и поиск их решений;
- 4) вывод итогового решения исходя из выполнения дополнительных аналитических выкладок.

Принцип работы данного класса методов представлен на следующей схеме:



Рисунок 4. Принцип работы алгоритмов требующие отыскания всех корней резольвентного и вспомогательных уравнений.

Организация экспериментов

В ходе представленной работы, было проведено:

- теоретическое исследование относительной погрешности вычисления корней многочленов с применением плавающей точки разными методами;
- экспериментальное исследование относительной погрешности вычисления корней многочленов с применением плавающей точки разными методами;
- экспериментальное сравнение временных затрат на вычисления корней многочленов разными методами.

Были проведены эксперименты для 11 программно реализованных методов поиска корней полиномиальных уравнений. Среди представленных, алгоритмы:

Феррари, Декарта, NBS, Эйлера, Ван дер Вардена, Чирнхауса, Зальцера, Скуайра, FQS, Меримена и Ангера.

Гибридно-аналитический способ

В данном пункте были описаны принципы организации экспериментов с использованием гибридно-аналитического подхода. Основная концепция данного подхода заключается в выполнении аналитических выкладок, включающих переопределение коэффициентов и вспомогательных переменных с учетом заданных погрешностей, а также определение арифметических операций с соответствующими им погрешностями.

В рамках исследования были использованы следующие модели:

1) Коэффициенты исходного полинома, заданные с погрешностями (ε_n – погрешность заданной величины):

- $\tilde{a} = a * (1 + \varepsilon_a),$
- $\tilde{b} = b * (1 + \varepsilon_b),$
- $\tilde{c} = c * (1 + \varepsilon_c),$
- $\tilde{d} = d * (1 + \varepsilon_d),$
- $\tilde{e} = e * (1 + \varepsilon_e);$

2) Переопределение арифметических операций с соответствующими им погрешностями ($\varepsilon_{operation}$ – погрешность заданной операции):

- $add(a, b) = (a + b)(1 + \varepsilon_{add}),$
- $sub(a, b) = (a - b)(1 + \varepsilon_{sub}),$
- $mul(a, b) = (a * b)(1 + \varepsilon_{mul}),$
- $div(a, b) = (a/b)(1 + \varepsilon_{div}),$
- $fma(a, b, c) = (a * b + c)(1 + \varepsilon_{fma}),$

- $fms(a, b, c, d) = (a * b - c * d)(1 + \varepsilon_{fms}),$
- $sqrt(a) = \sqrt{a} \left(1 + \frac{\varepsilon_a}{2} + \varepsilon_{sqrt}\right),$
- $cbrt(a) = \sqrt[3]{a} \left(1 + \frac{\varepsilon_a}{3} + \varepsilon_{cbrt}\right).$

Для упрощения полученных выражений и сохранения линейности относительно ε , введен метод `simplify_eps(expr)` осуществляющий обработку выражения `expr`, вследствие чего происходит обнуление перемножений и возведений в степень ε -содержащих элементов, т.к. подобные операции над ε -содержащими элементами влекут за собой малые значения, которые следует обнулять.

Демонстрация принципов работы метода `simplify_eps(expr)`:

- $expr = \varepsilon_a \varepsilon_b + c \varepsilon_c + \sqrt{d},$
- $simplify_eps(expr) \rightarrow c \varepsilon_c + \sqrt{d}.$

В результате выполнения выкладок требуется определить наихудший случай для полученной формулы. Для этой цели разработан метод `abs_coeff_eps(expr)`, основанный на следующих принципах:

- 1) группировка всех коэффициентов относительно каждого элемента, содержащего погрешность ε_{temp} ;
- 2) вычисление модуля каждого сгруппированного коэффициента;
- 3) замена всех элементов, содержащих погрешность ε_{temp} , единой константой — ε .

Демонстрация принципов работы метода `abs_coeff_eps(expr)`,

- $expr = \varepsilon_a(d + ac) - \varepsilon_b d + ab - c,$
- $abs_coeff_eps(expr) \rightarrow \varepsilon(|d + ac| + |d|) + ab - c.$

В результате выполнения вышеперечисленных действий, была выведена формула решения для наихудшего случая, представленная следующим образом:

$$Z_n = \varepsilon * expr + root,$$

- *expr* – некоторое выражение, помноженное на погрешность ε ;
- *root* – аналитическое выражение найденного корня, не содержащее погрешность ε .

После определения наихудшего случая необходимо осуществить подстановку численных данных в полученную формулу. Для этой цели разработаны методы с приставкой “*calc*”, вызов которых определен следующим образом: *calc_“название_метода”(Zn, coeffs, epsilon, delta, gamma)*. Каждый из этих методов осуществляет подстановку переданных численных данных и возвращает массив решений, содержащий наихудшую погрешность.

Следующим шагом, является определение оценки полученных данных с использованием метода *find_max_error(foundRoots, trueRoots)*. Данный метод определяет наихудшую относительную погрешность среди четырех корней заданного полинома, по формуле:

$$\frac{|trueRoot[i][j] - foundRoot[i][j]|}{|trueRoot[i][j]| + \varepsilon}.$$

На выходе получая массив наихудшей относительной погрешности для каждого полинома из выборки.

Заключительным этапом является построение графика поверхности, иллюстрирующего зависимость двух коэффициентов полинома и логарифмированной величины относительной погрешности. Для выполнения поставленной задачи применяется реализованный метод *plot_3d_surface(coeff_1, coeff_2, rel_errors)*. В результате выводится шесть графиков поверхностей, отображающих зависимость каждого

коэффициента полинома и массива логарифмированных значений относительной погрешности.

Исходные тексты программной имплементации вывода выкладок, определения наихудшего случая, вычисления худшей относительной погрешности и вывода графика поверхности приведены в Приложении (см. файл framework.ipynb).

Численный способ

В данном пункте представлен обзор принципов организации экспериментов с использованием численного метода. Основная задача заключается в оценке результатов работы реализованных на языке программирования C++ методов. Далее приведено описание организации экспериментов.

В первую очередь необходимо сгенерировать корни полиномиального уравнения. Для этого был разработан метод `generate_polynomial()`. В первую очередь необходимо сгенерировать корни полиномиального уравнения. Для этого был разработан метод:

- 1) кластеризированные корни – генерируются случайным образом на расстоянии не более заданного друг от друга;
- 2) некластеризированные корни – генерируются на интервале $[-n, n]$, случайным образом;
- 3) кратные корни – генерируются случайным образом на интервале $[-n, n]$, с учетом, что все корни являются эквивалентными друг другу.

Следующим шагом после генерации корней, является вычисление коэффициентов на основе сгенерированных корней. Коэффициенты вычисляются с использованием следующих формул:

- $a = -Z_1 - Z_2 - Z_3 - Z_4,$
- $b = Z_1Z_2 + Z_1Z_3 + Z_1Z_4 + Z_2Z_3 + Z_2Z_4 + Z_3Z_4,$
- $c = -Z_1Z_2Z_3 - Z_1Z_2Z_4 - Z_1Z_3Z_4 - Z_2Z_3Z_4,$

- $d = Z_1Z_2Z_3Z_4$.

Полученные коэффициенты передаются в качестве входных данных в имплементированный метод, который возвращает решение исходного полинома. Затем производится оценка относительной погрешности полученных решений с помощью имплементированного метода `compare_roots()`, который возвращает величину наихудшей относительной погрешности корней заданного полинома.

В ходе всех проведенных экспериментов определяется наихудшая относительная погрешность, полученная за все эксперименты. Для реализации данной задачи был разработан метод `testQuarticPolynomial(testCount, maxDistance, P1, P2)`, где входными параметрами являются переменные, определяющие количество экспериментов, максимальное расстояние между корнями (если количество кластеризованных корней больше 0), количество кластеризованных корней и количество кратных корней.

Исходные тексты программной имплементации генерации коэффициентов полинома и метода определения наихудшей относительной погрешности приведены в Приложении (см. файл excerpt.cpp).

Результаты экспериментов (гибридно-аналитический способ)

В данном пункте представлены результаты проведенных экспериментов гибридно-аналитическим способом. Исходная выборка 10 000 полиномов. Выведены по 6 графиков поверхности для каждого из 11 имплементированных методов. Представленные графики демонстрируют зависимость между двумя коэффициентами полинома и логарифмированной величиной относительной погрешности решения.

Оценка проводилась для некластеризованных корней, с заданной точностью `float` и заданными погрешностями $\varepsilon =$

`numeric_limits<float>::epsilon ()` $\approx 1.19209e - 07$, $\delta = \sqrt{\varepsilon}$, $\gamma = \varepsilon$,
где ε – погрешность метода, δ – погрешность решения резольвентного
уравнения, γ – погрешность решения вспомогательного уравнения.

Алгоритм Феррари

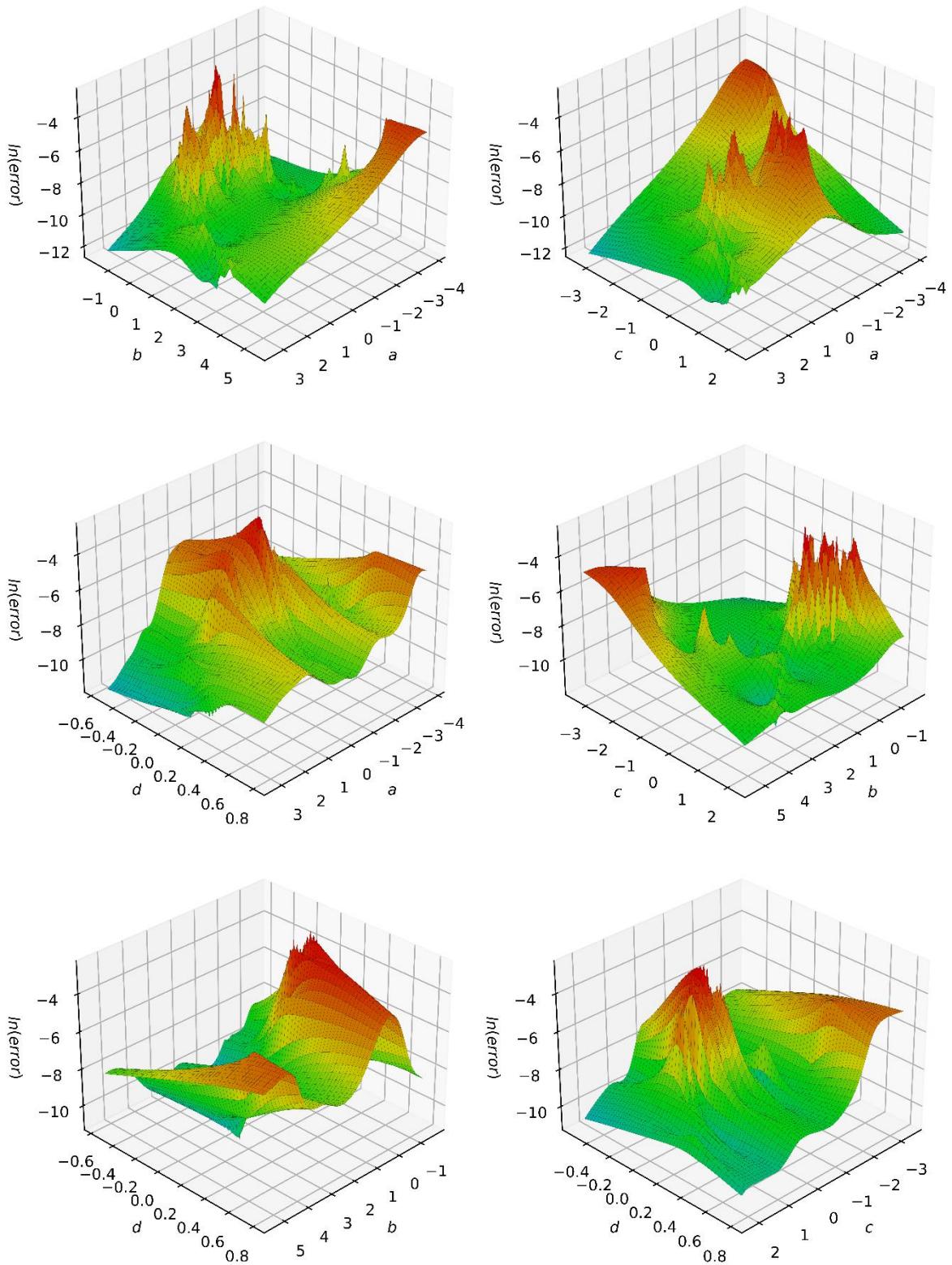


Рисунок 5. Результаты алгоритма Феррари.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e-07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Декарта

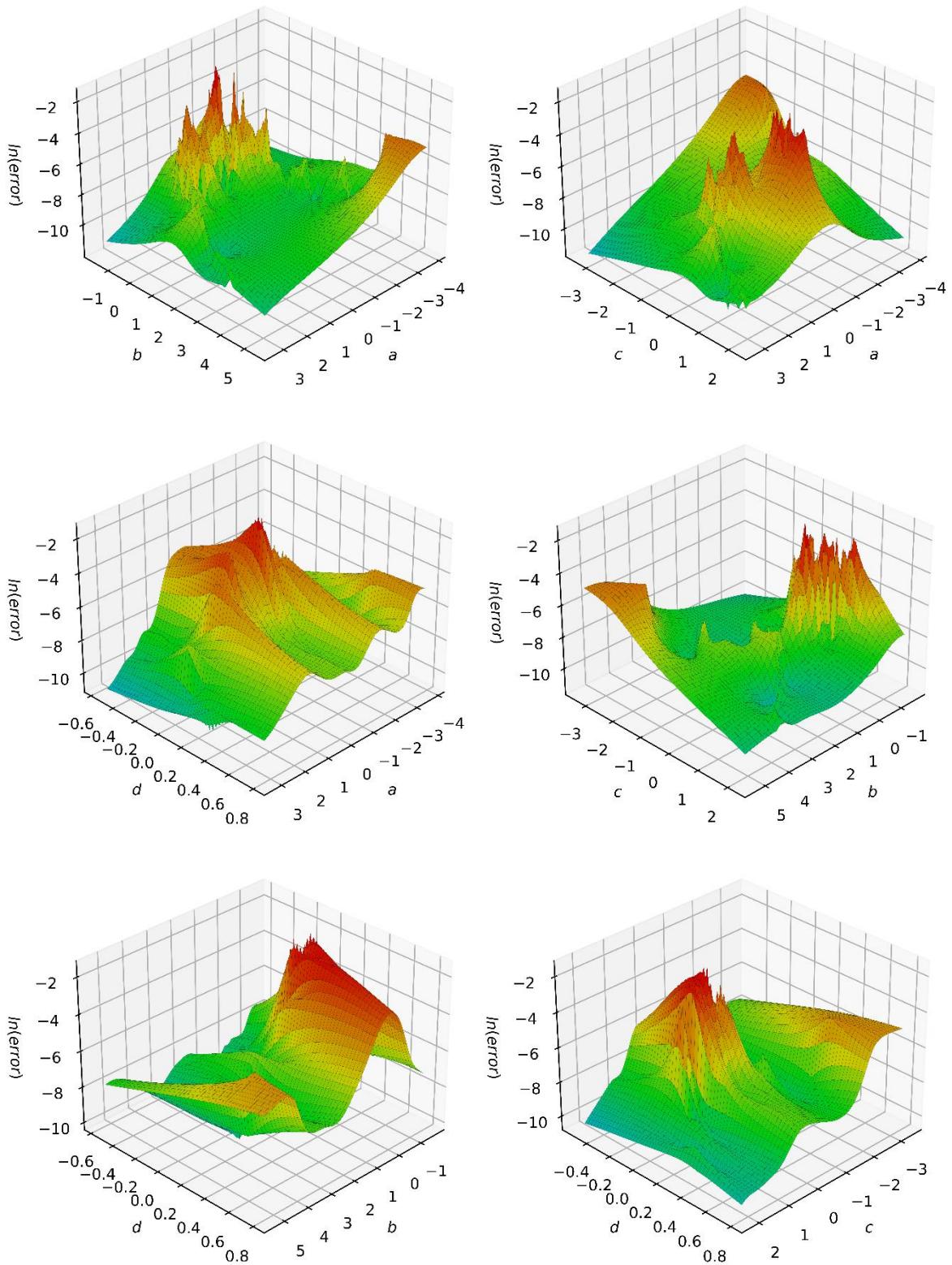


Рисунок 6. Результаты алгоритма Декарта.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e-07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм NBS

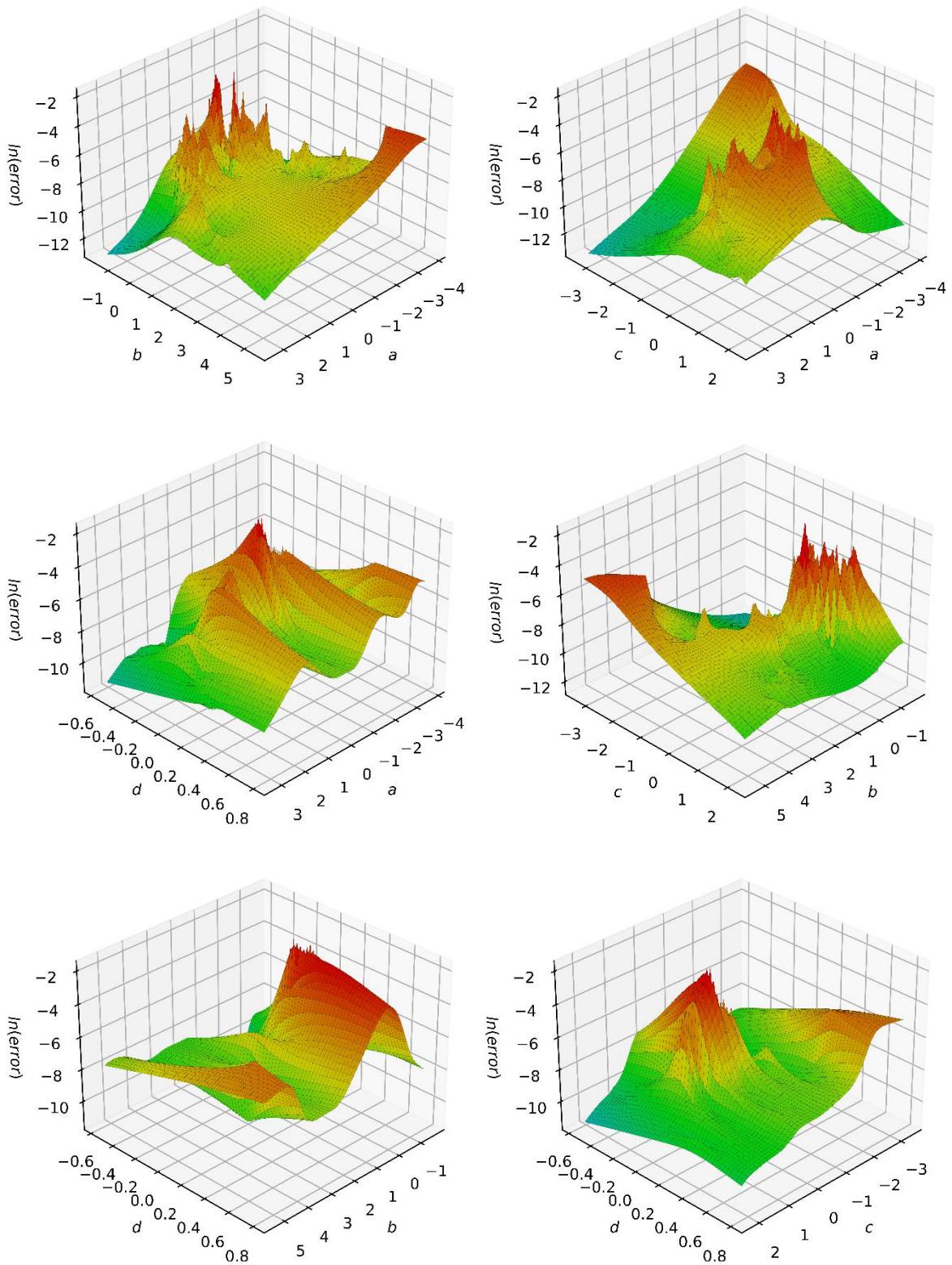


Рисунок 7. Результаты алгоритма NBS.

Оценка некластер. корней сгенерированных с точностью $float$
 (заданный $\varepsilon = numeric_limits<float>::epsilon() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Эйлера

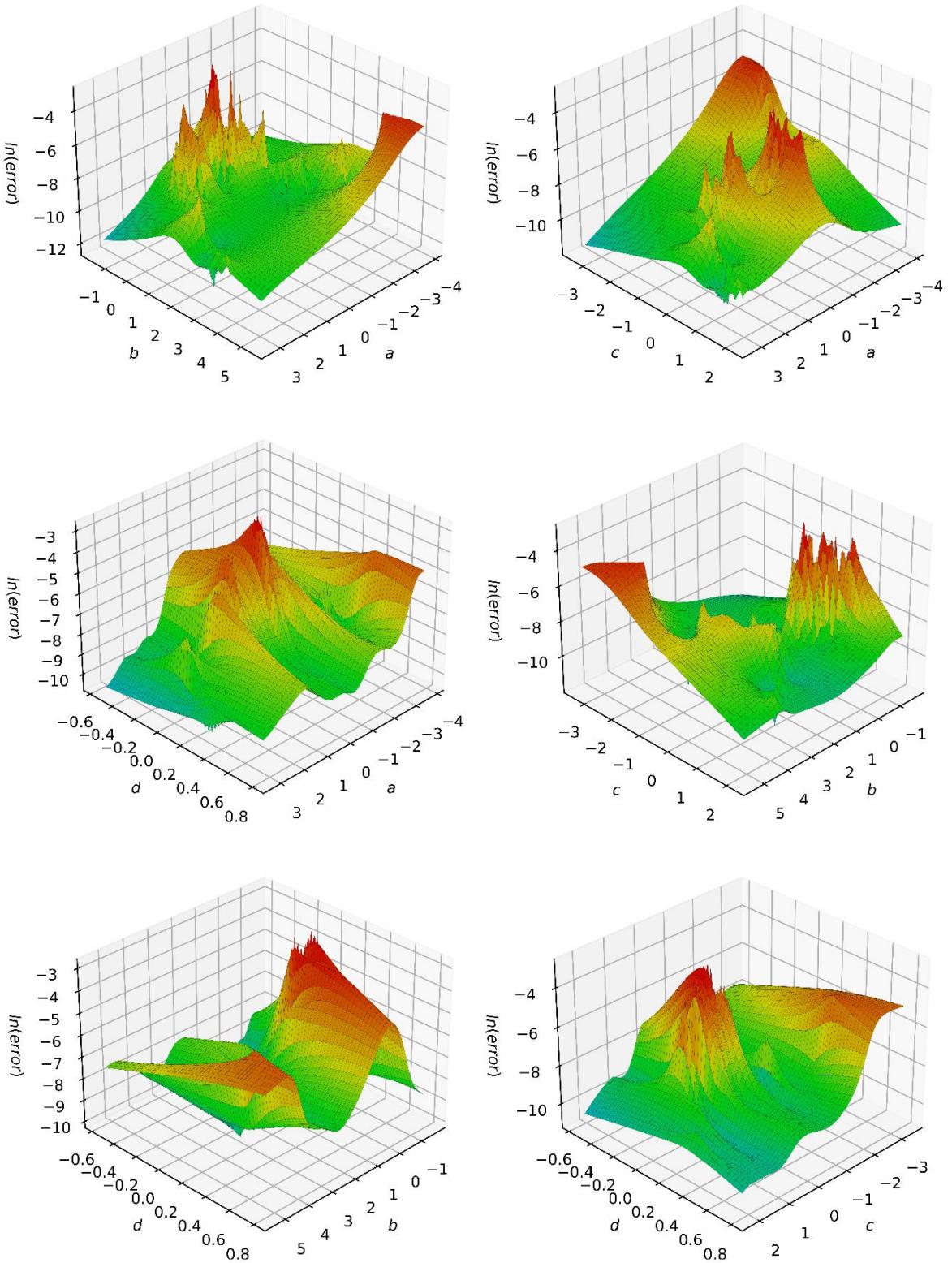


Рисунок 8. Результаты алгоритма Эйлера.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e-07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Ван дер Вардена

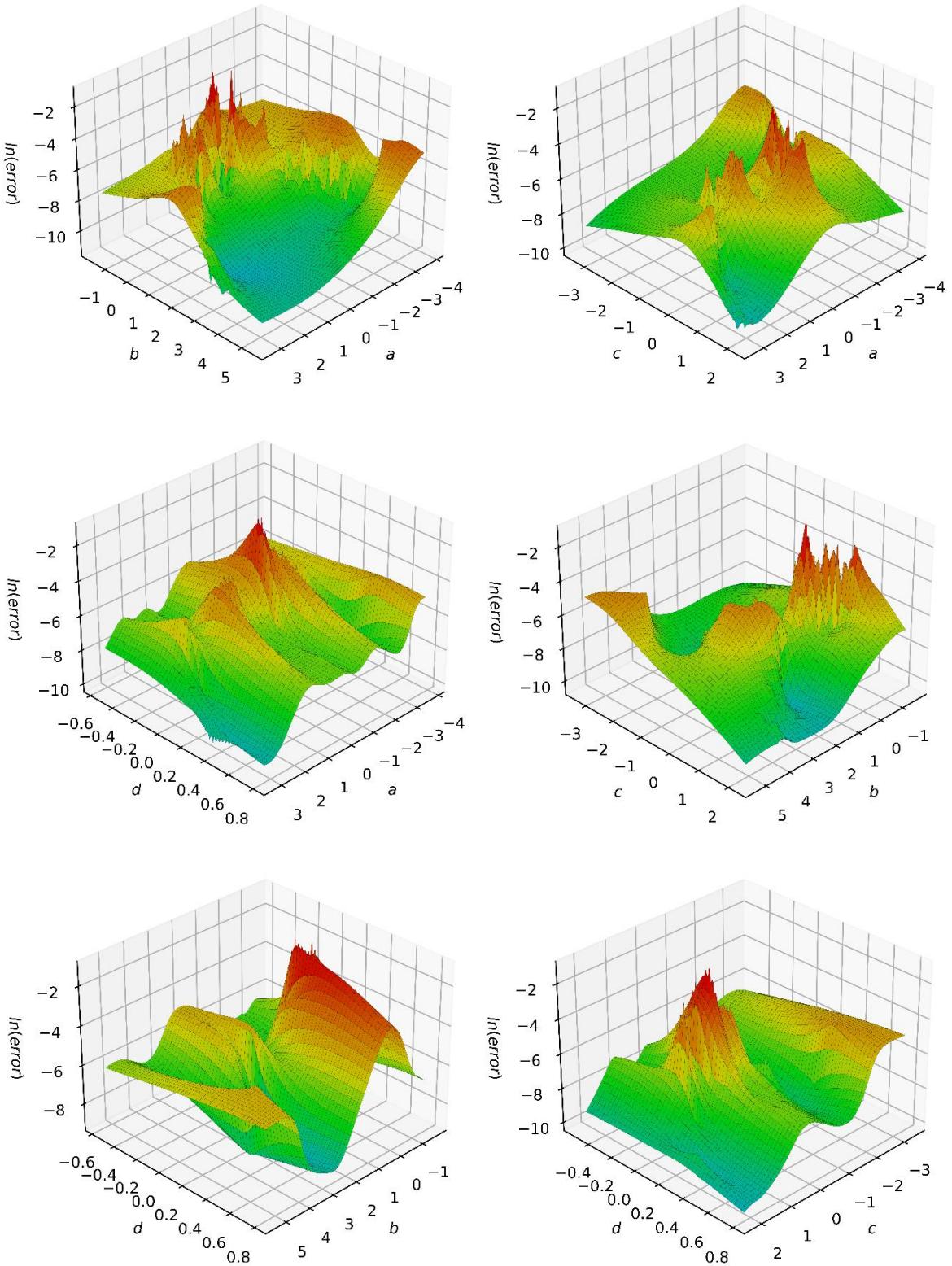


Рисунок 9. Результаты алгоритма Ван дер Вардена.
Оценка некластер. корней сгенерированных с точностью float
(заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Чирнхауса

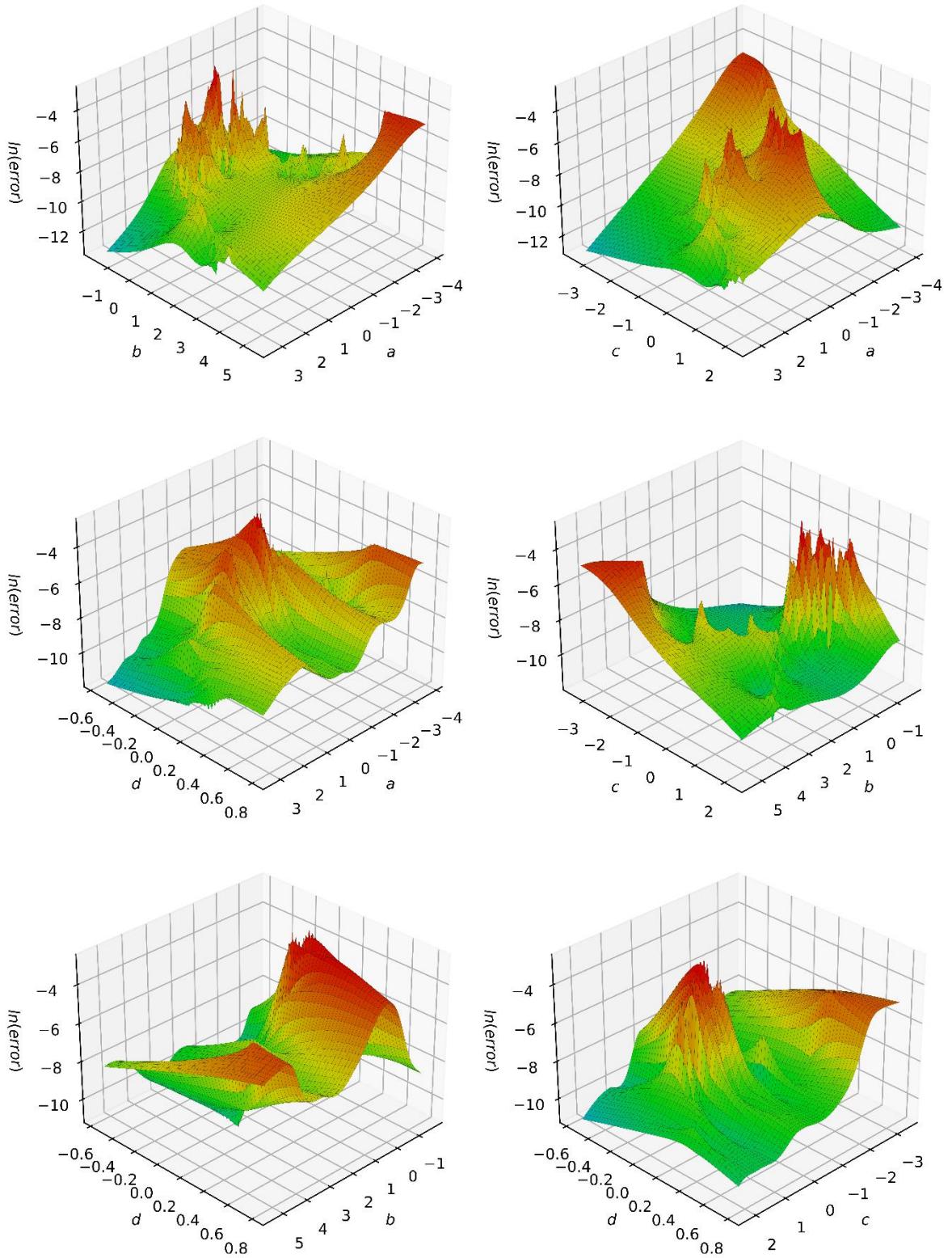


Рисунок 10. Результаты алгоритма Чирнхауса.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e-07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Скуайра

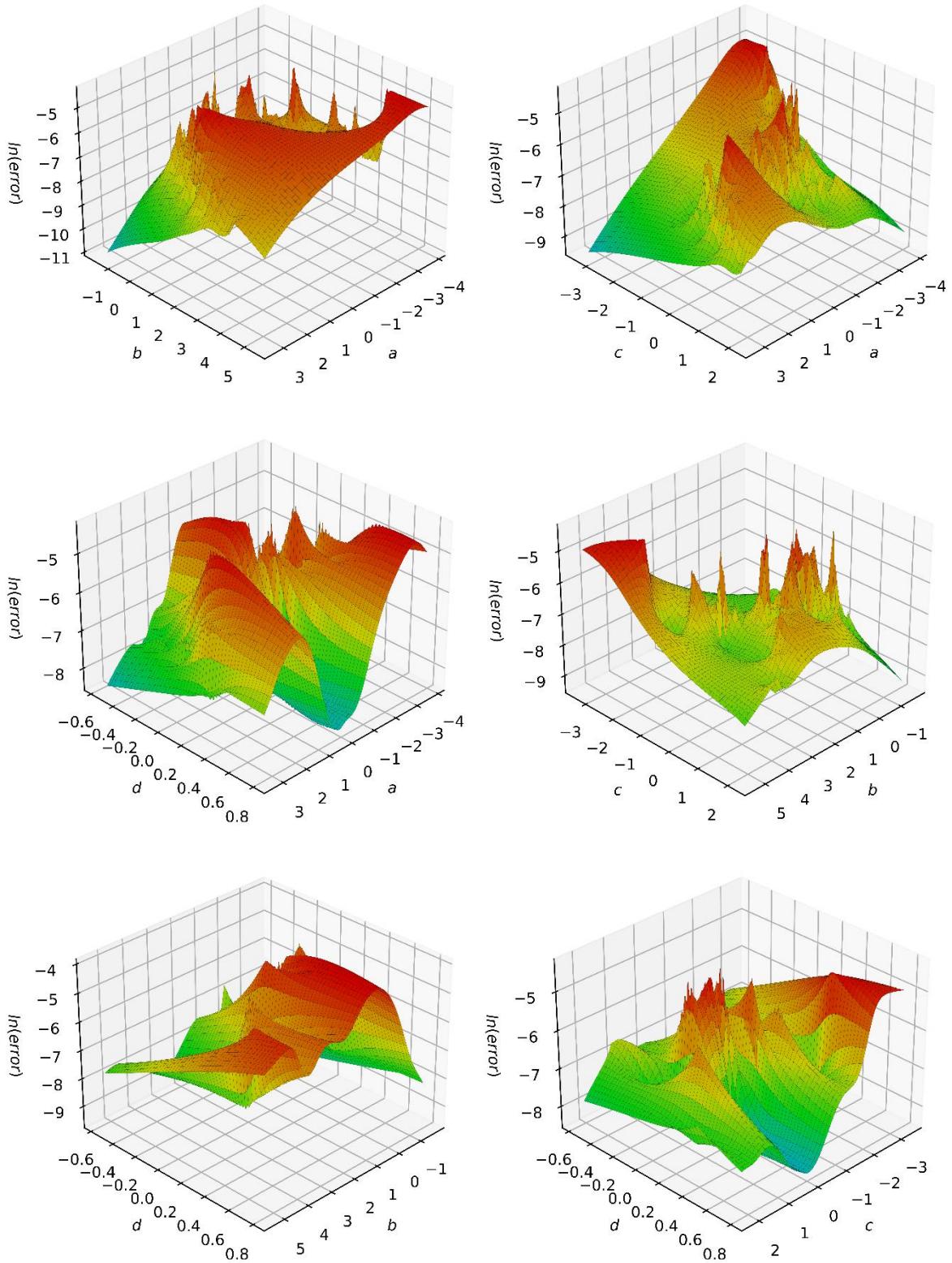


Рисунок 11. Результаты алгоритма Скуайра.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Зальцера

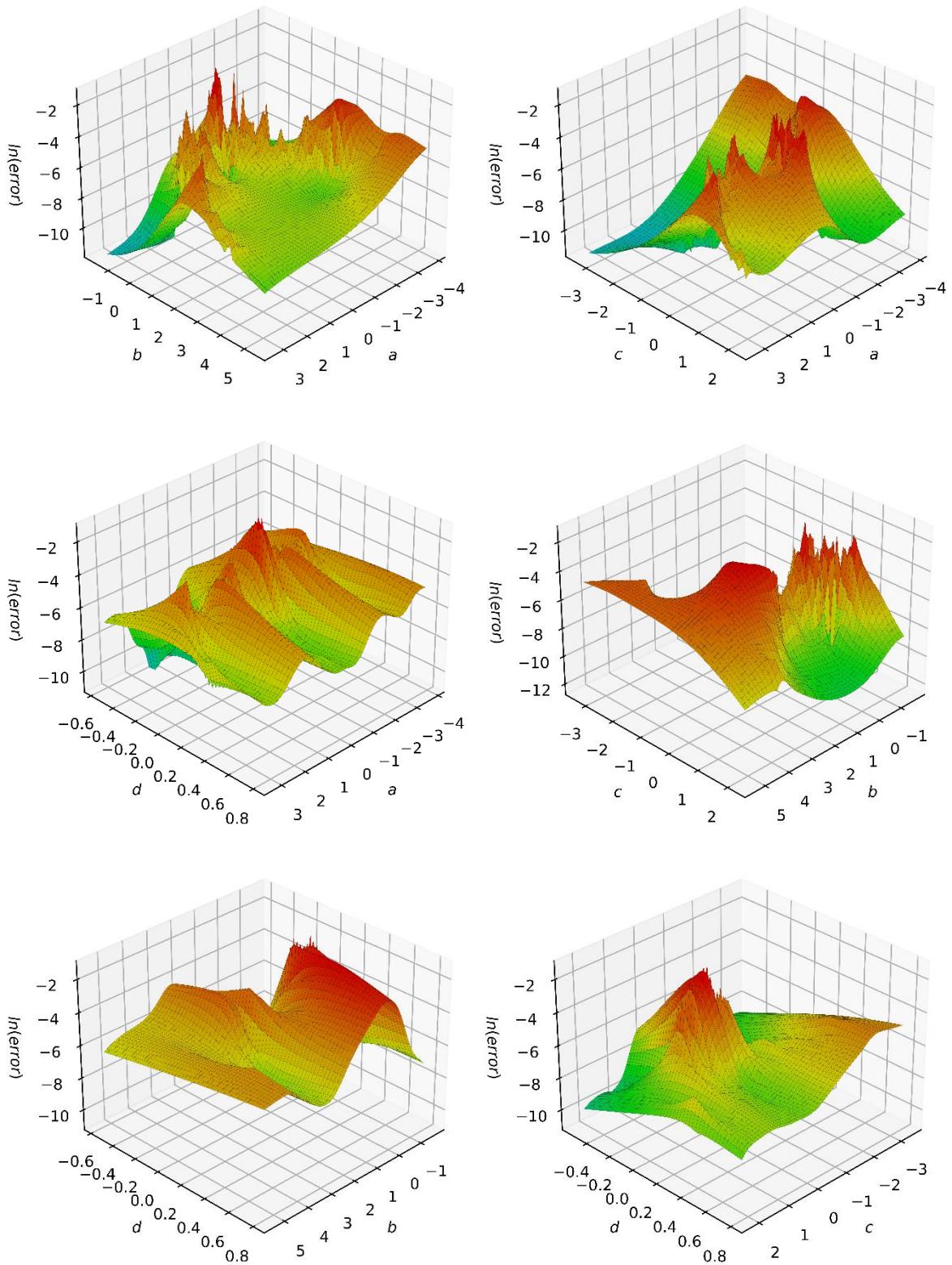


Рисунок 12. Результаты алгоритма Зальцера.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм FQS

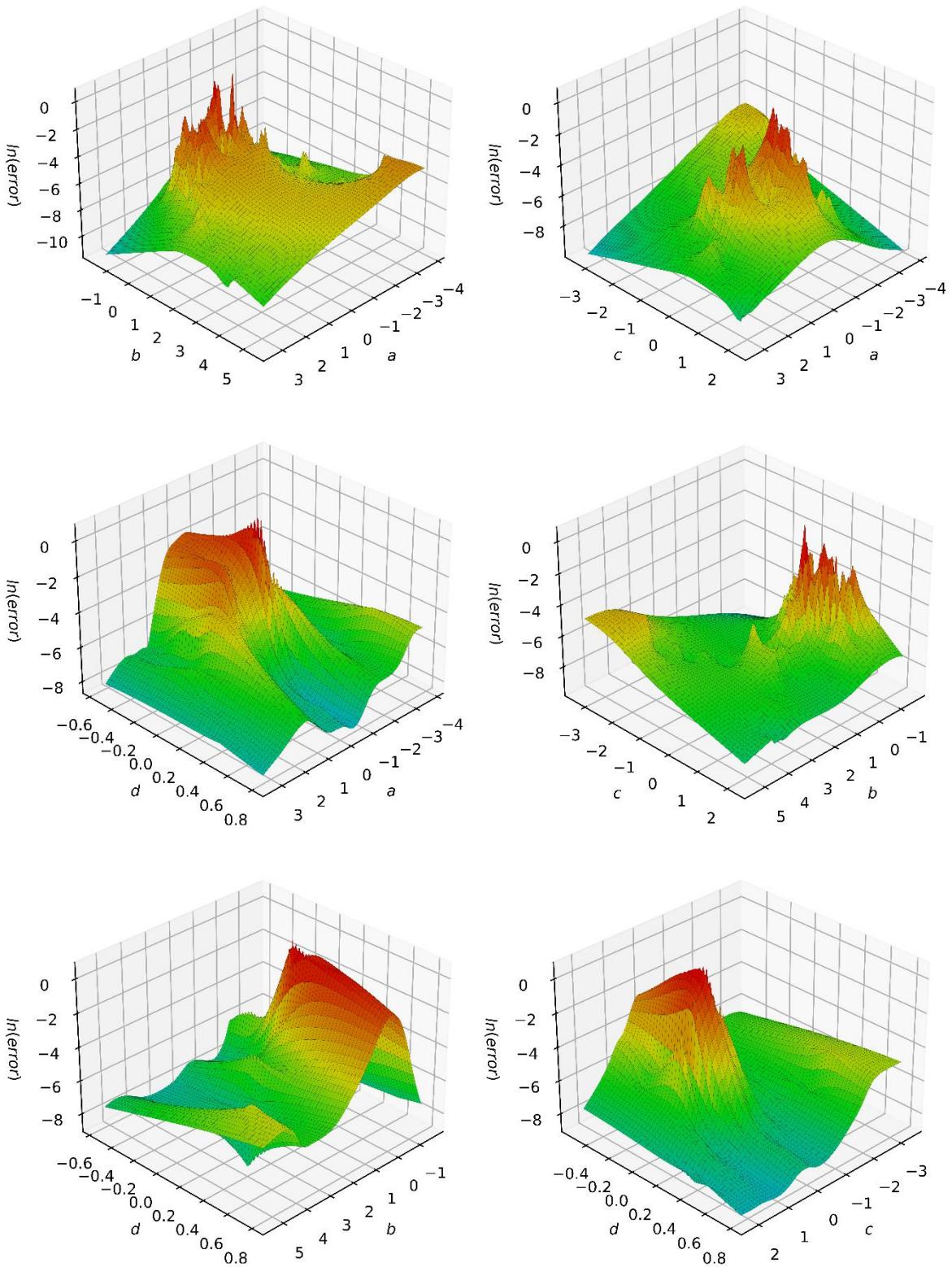


Рисунок 13. Результаты алгоритма FQS.

Оценка некластер. корней сгенерированных с точностью $float$
 (заданный $\varepsilon = numeric_limits<float>::epsilon() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Мерримена

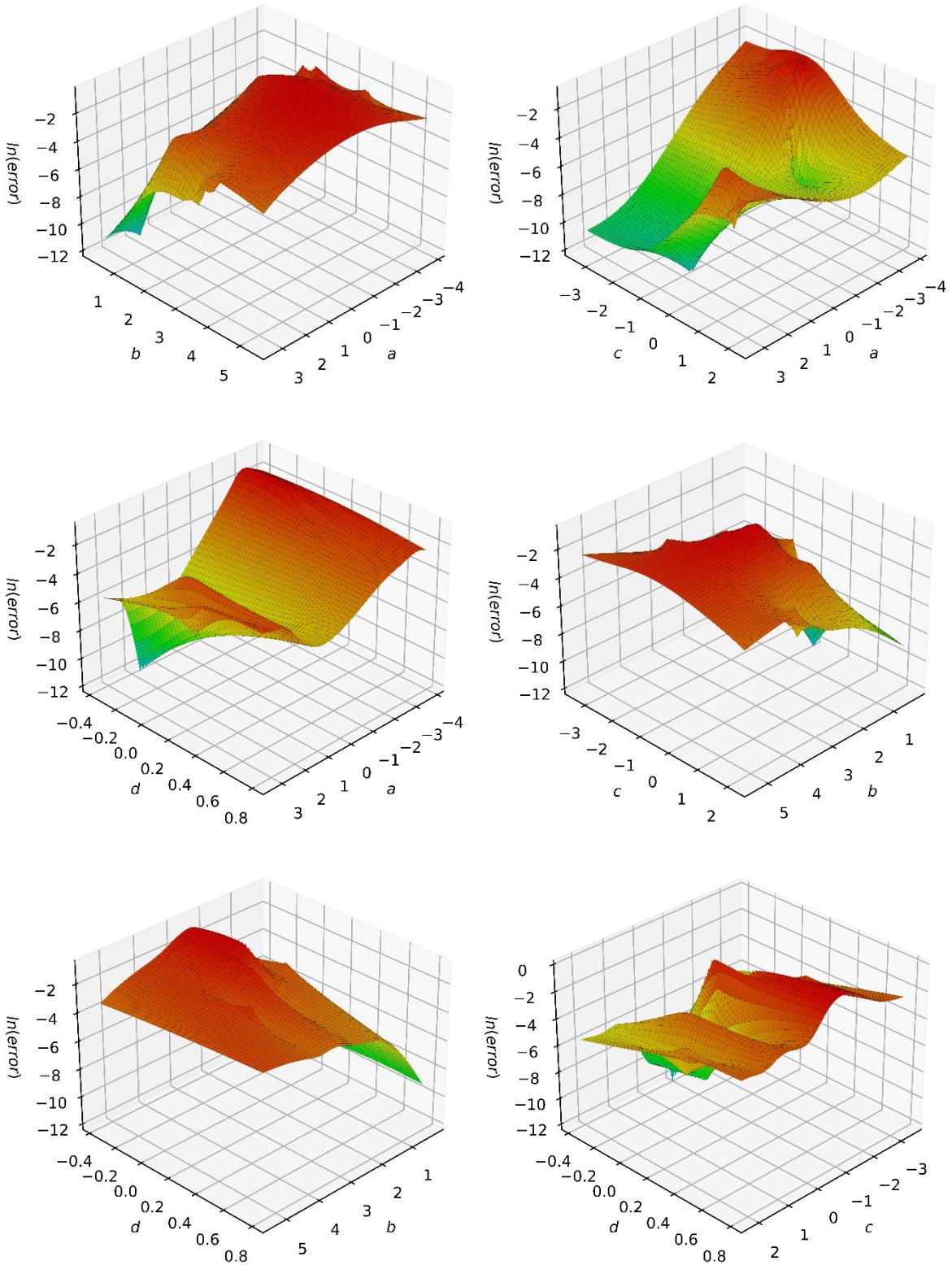


Рисунок 14. Результаты алгоритма Мерримена.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

Алгоритм Ангера

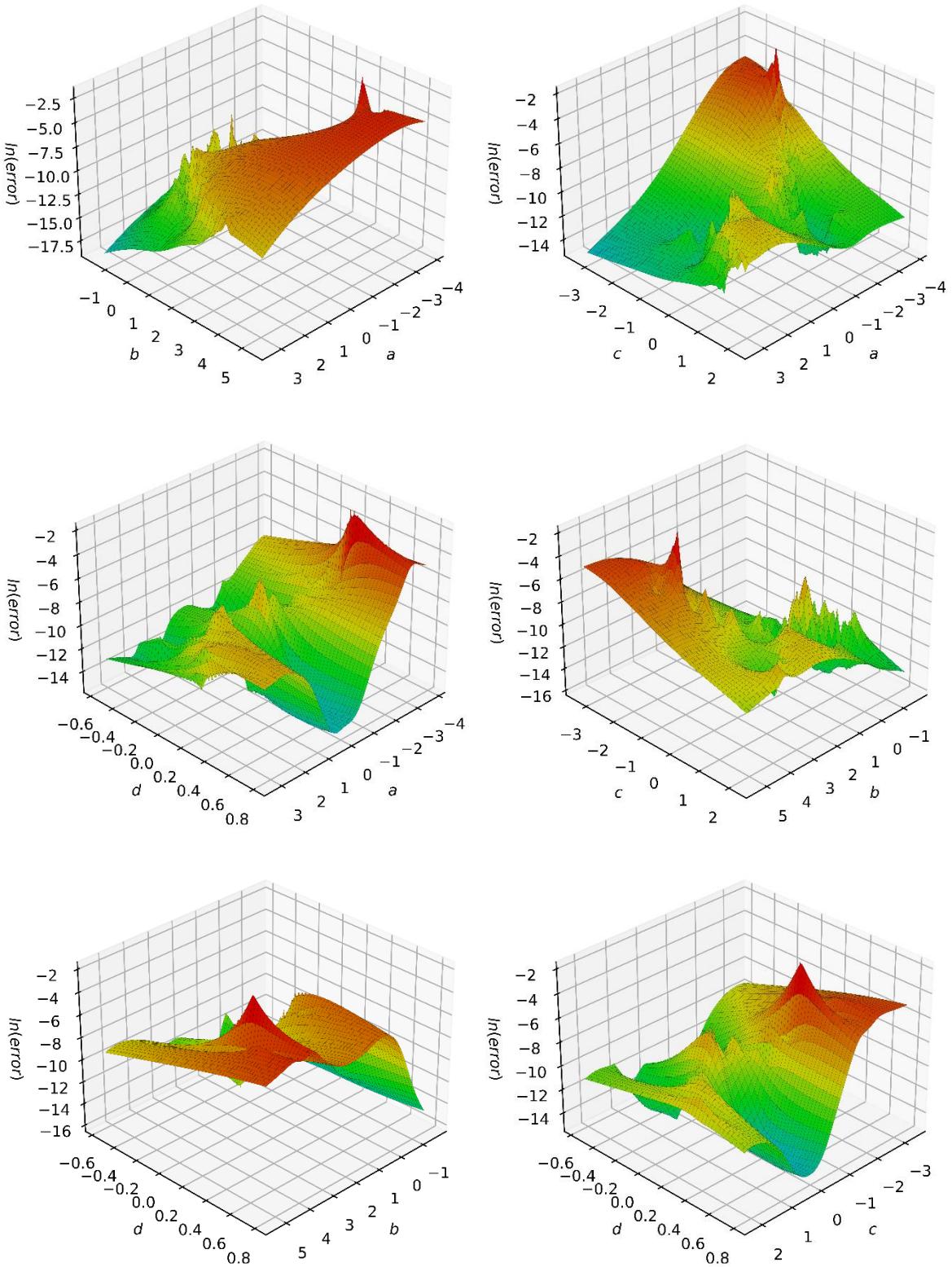


Рисунок 15. Результаты алгоритма Ангера.

Оценка некластер. корней сгенерированных с точностью float
 (заданный $\varepsilon = \text{numeric_limits}\langle\text{float}\rangle::\text{epsilon}() \approx 1.19209e - 07$,
 $\delta = \sqrt{\varepsilon}, \gamma = \varepsilon$)

На представленных графиках наблюдаются всплески, о которых будет подробнее рассказано в следующем пункте, где рассматриваются численные эксперименты.

На основании имеющейся информации можно заметить, что многие методы демонстрируют схожие результаты. Однако наблюдаются аномалии, связанные с низким количеством построенных вершин на графиках, отражающих результаты алгоритма Мерримена. Это связано с тем, что алгоритм Мерримена проводит проверку следующего условия:

$$m^2 - n \geq 0.$$

Если условие выполняется, то метод проводит дальнейшее аналитическое решение. В случае, если условие не выполняется, Мерримен утверждает, что заданный полином имеет четыре вещественных или мнимых корня. Однако в данной ситуации алгоритм Мерримена не имеет аналитического решения для поиска данных корней. Отсюда и следует потеря львиной доли решений.

Результаты экспериментов (численный способ)

В данном пункте представлены результаты экспериментов, проведенных с использованием имплементированных методов на языке программирования C++. Исходная выборка состоит из 10 000 000 полиномов. Результаты представлены в виде таблиц, в которых отражены результаты для всех возможных случаев, включая кластеризованные, некластеризованные и кратные корни.

Феррари	Декарт	NBS	Van дер Варден	Эйлер
1	1	0.98	0.9	0.91
Чирнхаус	Зальцер	Скуайр	FQS	Мерримен
0.9	0.96	0.79	0.71	0.83
				0.90

Таблица 1. Относительная погрешность наихудшего случая для всех кластеризованных корней.

Феррари	Декарт	NBS	Ван дер Варден	Эйлер
0.99	0.99	0.93	0.81	0.88
Чирнхаус	Зальцер	Скуайр	FQS	Мерримен
0.79	0.9	0.72	0.72	0.8

Таблица 2. Относительная погрешность наихудшего случая для всех некластеризованных корней.

Феррари	Декарт	NBS	Ван дер Варден	Эйлер
0.99	0.98	0.98	0.92	0.93
Чирнхаус	Зальцер	Скуайр	FQS	Мерримен
0.82	0.91	1	0.78	0.86

Таблица 3. Относительная погрешность наихудшего случая для всех кратных корней.

Наблюдается тревожная тенденция, при которой полученные величины явно стремятся к 1. Данный феномен связан с вычитанием близких корней, что приводит к получению величины близкой к 1, и даже более, если требуется работать с корнями близкими к 0. В связи с этим экспериментально был определен квантиль-случаев, которые исключаются из рассмотрения. Для кластеризованных и кратных корней из рассмотрения исключается - около 10%, а для некластеризованных корней - около 1%.

После исключения из рассмотрения σ квантиль-случаев, можем наблюдать следующую картину

Феррари	Декарт	NBS	Ван дер Варден	Эйлер
0.02	0.02	0.27	0.02	0.02
Чирнхаус	Зальцер	Скуайр	FQS	Мерримен
0.03	0.05	0.02	0.01	0.1

Таблица 4. Относительная погрешность наихудшего случая для всех кластеризованных корней ($\sigma = 10\%$).

Феррари	Декарт	NBS	Ван дер Варден	Эйлер
2.3e-4	1.9e-4	6.8e-4	5.8e-4	6.3e-4
Чирнхаус	Зальцер	Скуайр	FQS	Мерримен
3.5e-4	1.4e-3	1.1e-4	9.2e-5	9.3e-3

Таблица 5. Относительная погрешность наихудшего случая для всех некластеризованных корней ($\sigma = 1\%$).

Феррари	Декарт	NBS	Ван дер Варден	Эйлер
0.02	0.02	0.27	0.02	0.02
Чирнхаус	Зальцер	Скуайр	FQS	Мерримен
0.03	0.01	0.02	0.01	0.1

Таблица 6. Относительная погрешность наихудшего случая для всех кратных корней ($\sigma = 10\%$).

Полученная картина стала более наглядной, что позволяет провести объективный анализ полученных данных. Как видно из результатов, лучшие результаты демонстрируют методы, основанные на итеративном подходе для определения коэффициентов вспомогательного уравнения. Среди всех рассмотренных методов наилучшие показатели продемонстрировал алгоритм FQS. Хуже всех справились методы, использующие исключительно аналитический подход для поиска корней полиномиального уравнения. Алгоритм Мерримена показал самые слабые результаты среди всех рассмотренных методов. Методы, требующие поиска всех решений резольвентного уравнения с поиском и без поиска всех решений вспомогательных уравнений, показали практически одинаковые результаты. Однако, для каждого случая наблюдается лучший и худший представитель. Алгоритм Декарта является предпочтительным при работе с некластеризованными корнями, в то время как метод Зальцера показывает наихудшие результаты выполнения данной задачи. Другая ситуация наблюдается в случае работы с кратными корнями, где метод Зальцера показывает лучший результат среди своих одноклассников, а наихудшим представителем является алгоритм NBS. В случае работы с кластеризованными корнями отсутствует явный фаворит, однако стоит отметить менее эффективное выполнение данной задачи методом Зальцера, который уступает другим представителем в своем классе почти в два раза.

Также представлена таблица лучших представителей для каждого случая среди своего класса:

	Итеративные	Аналитические	Резольвентное уравнения	Резольвентное и вспомогательные уравнения
Кластеризованные	FQS	Ангер	—	Чирнхаус
Некластеризованные	FQS	Ангер	Декарт	Чирнхаус
Кратные	FQS	Ангер	Зальцер	Чирнхаус

Таблица 7. Лучшие представители для каждого случая среди всех классов.

Найденные неточности исходной литературы

Алгоритм Ангера

В ходе исследования данного метода, были выявлены неточности исходной литературы [12], посвященной обзору данного алгоритма.

1. Рассматриваем формулы (3.7) [12]:

$$(3\beta)^3 = \frac{R + \sqrt{R^2 - 4Q^3}}{2}, \quad (7.1)$$

$$(3\gamma)^3 = \frac{R - \sqrt{R^2 - 4Q^3}}{2}. \quad (7.2)$$

В формулах (7.1) и (7.2) все выражение в знаменателе делится на 2, когда в следующих выкладках (3.8) [12] автора не наблюдаем данной операции в выражении представленного радиканда:

$$\beta = \frac{1}{3} \sqrt[3]{R + \frac{\sqrt{R^2 - 4Q^3}}{2}}, \quad (7.3)$$

$$\gamma = \frac{1}{3} \sqrt[3]{R - \frac{\sqrt{R^2 - 4Q^3}}{2}}, \quad (7.4)$$

И в последующих выкладках (4.8) и (B) [12] наблюдаем подобную картину:

$$\beta_0 = \frac{4}{3} \sqrt[3]{R + \frac{\sqrt{R^2 - 4Q^3}}{2}}, \quad (7.5)$$

$$\gamma_0 = \frac{4}{3} \sqrt[3]{R - \frac{\sqrt{R^2 - 4Q^3}}{2}}, \quad (7.6)$$

$$T = 3a_1^2 - 8a_2 + 8 \operatorname{Re} \sqrt[3]{R + \frac{\sqrt{R^2 - 4Q^3}}{2}}. \quad (7.7)$$

Т.к. все выражение радиканда делится на 2, что следует из (7.1) и (7.2), получаем исправленные выражения для (7.3) – (7.7):

$$\beta = \frac{1}{3} \sqrt[3]{\frac{R + \sqrt{R^2 - 4Q^3}}{2}},$$

$$\gamma = \frac{1}{3} \sqrt[3]{\frac{R - \sqrt{R^2 - 4Q^3}}{2}},$$

$$\beta_0 = \frac{4}{3} \sqrt[3]{\frac{R + \sqrt{R^2 - 4Q^3}}{2}}, \quad (7.8)$$

$$\gamma_0 = \frac{4}{3} \sqrt[3]{\frac{R - \sqrt{R^2 - 4Q^3}}{2}}, \quad (7.9)$$

$$T = 3a_1^2 - 8a_2 + 8 \operatorname{Re} \sqrt[3]{\frac{R + \sqrt{R^2 - 4Q^3}}{2}}.$$

2. Данное выражение (4.10) [12] имеет неточность, связанную с определением коэффициента $\frac{16}{8}$:

$$\beta_0 \gamma_0 = \frac{16}{8} Q. \quad (7.10)$$

Исходя из (7.8) и (7.9) замечаем, что истинным значением коэффициента перед Q является величина $\frac{4}{3} * \frac{4}{3} = \frac{16}{9}$. Отсюда исправленная формула для (7.10) имеет следующий вид:

$$\beta_0 \gamma_0 = \frac{16}{9} Q.$$

3. Рассматриваем следующее выражение (B) [12]:

$$R = 27a_1^2 a_4 - 9 a_1 a_2 a_3 + 2a_2^3 - 27a_2 a_4 + 27a_3^2.$$

При расчете R , коэффициент перед $a_2 a_4$ равен -27, а должен быть равен -72. Данная величина была определена аналитически. Отсюда формула (7.11) имеет следующий вид:

$$R = 27a_1^2 a_4 - 9 a_1 a_2 a_3 + 2a_2^3 - 72a_2 a_4 + 27a_3^2.$$

Алгоритм Скуайра

В ходе исследования данного метода, были выявлены неточности исходной литературы [9], посвященной обзору данного алгоритма.

1. Рассматриваем формулу полиномиального уравнения шестой степени (3) [9]:

$$\begin{aligned} S(p) = \\ a_0^3 - a_0^2 a_2 p + a_0(a_1 a_3 - a_0)p^3 + a_0(a_3^2 - 2a_2 + a_1^2)p^3 + \\ (a_1 a_3 - a_0)p^4 - a_2 p^5 + p^6 = 0. \end{aligned} \quad (8.1)$$

В данной формуле (8.1) имеются неточности, связанные с определением коэффициентов и степени искомой переменной. Исправленное выражение имеет следующий вид:

$$\begin{aligned} S(p) = \\ a_0^3 - a_0^2 a_2 p + a_0(a_1 a_3 - a_0)p^2 + (-a_0 a_3^2 + 2a_0 a_2 - a_1^2)p^3 + \\ (a_1 a_3 - a_0)p^4 - a_2 p^5 + p^6 = 0. \end{aligned} \quad (8.2)$$

2. Рассматриваем следующие формулы (4b) и (4c) [9]:

$$q^1 = \frac{[p(a_3 p - a_1)]}{p^2 - a_0}, \quad (8.3)$$

$$q = \frac{[(a_1 p - a_3 a_0)]}{p^2 - a_0}. \quad (8.4)$$

Здесь формулы (8.3) и (8.4) перепутаны местами. Истинные выражения имеют следующий вид:

$$q = \frac{[p(a_3 p - a_1)]}{p^2 - a_0},$$

$$q^1 = \frac{[(a_1 p - a_3 a_0)]}{p^2 - a_0}.$$

3. Следующее выражение (5b) [9] содержит множество проблем:

$$q = q^1 = 0 \cdot 5a_3 \pm (a_3^2 - 4a_2 + 8p)^{\frac{1}{2}}. \quad (8.5)$$

Во-первых, выражение (8.5) неверно. Исправленное выражение имеет следующий вид:

$$q = q^1 = \frac{a_3 \pm \sqrt{a_3^2 - 4(a_2 - 2p)}}{2}. \quad (8.6)$$

Во-вторых, данная формула получена вследствие решения квадратного уравнения:

$$x^2 - a_3 x + (a_2 - 2p) = 0, \quad (8.7)$$

где q и q^1 являются его корнями.

Подобное решение (8.6) с использованием формулы дискриминанта влечет за собой большую потерю точности. Соответственно коэффициенты q и q^1 следует вычислять как корни указанного квадратного уравнения (8.7).

В-третьих, не рассматривается случай, когда дискриминант отрицателен. В этом случае коэффициенты q и q^1 образуют комплексно-сопряженную пару, что,

вероятно, не мешает получить корни исходного уравнения путем решения двух полученных в итоге квадратных уравнений с комплексными коэффициентами.

Заключение

В заключении данной работы обобщены результаты, полученные в процессе исследования, направленного на анализ вычислений корней полиномиальных многочленов.

1. Из теоретического исследования следует, что решения подавляющих алгоритмов при отсутствии вычислительной погрешности эквивалентны, поскольку каждый подавляющий алгоритм может быть преобразован в другой.

2. В ходе работы были классифицированы представленные методы на основе их подходов к решению задачи. В частности, выделены следующие классы алгоритмов:

- алгоритмы, основанные на чисто аналитическом подходе;
- алгоритмы, использующие итеративные методы для определения коэффициентов вспомогательных уравнений, которые решаются аналитически;
- алгоритмы, требующие поиска всех корней резольвентного уравнения, решаемого аналитически;
- алгоритмы, требующие поиска всех корней резольвентного уравнения и всех решений вспомогательных уравнений, решаемых аналитически.

3. Была разработана программная имплементация рассмотренных методов с использованием языка программирования C++.

4. Была разработана программная имплементация методов аналитической оценки наихудшей погрешности с использованием языка программирования Python и библиотеки символьных вычислений SymPy.

5. В ходе проведения экспериментов, которые объединяли гибридно-аналитический и численный подходы, было выяснено, что методы, использующие итеративные методы для определения коэффициентов вспомогательных уравнений (в частности, алгоритм FQS), лучше всего

справляются с поставленной задачей. С другой стороны, методы, основанные на чисто аналитическом подходе (в частности, алгоритм Мерримена), справляются с данной задачей хуже остальных представленных алгоритмов.

6. Было обнаружено, что катастрофическая потеря точности связана с операцией вычитания близких величин.

7. В ходе работы были обнаружены аномалии и неточности в исходных работах авторов представленных методов, что подчеркивает необходимость дальнейшего исследования и усовершенствования этих методов.

В целом, результаты данной работы предоставляют полезную информацию о различных алгоритмах и их эффективности в решении полиномиальных уравнений. Полученные результаты могут послужить основой для дальнейших исследований в данной области и помочь в разработке более точных и эффективных методов для поиска корней полиномиальных уравнений.

Список литературы

1. Wolters, D. J. (2020). Practical Algorithms for Solving the Quartic Equation. URL: <https://quarticequations.com/Quartic2.pdf> [Электронный ресурс]. стр. 3-7, 10-19. Дата обращения: 01.02.2023.
2. Cardano, Girolamo. (1993). The Rules of Algebra (Ars Magna) [1545], translated and edited by T. Richard Witmer. Dover Publications, Inc. стр. 231-265.
3. Descartes, René. (2016). The Geometry of Rene Descartes [1637], translated by David Eugene Smith and Marcia L. Latham. Dover Publications, Inc. стр. 180-187.
4. Euler, Leonhard. (2015). Elements of Algebra (Vollständige Anleitung zur Algebra) [1765], based on the 1828 edition of John Hewlett's 1822 translation. CreateSpace, Inc. & Kindle Direct Publishing. стр. 250-269.
5. Van der Waerden, B.L. (1991). Algebra, Vol 1 [1930], translated from the German by Fred Blum and John R. Schulenberger, (7th ed.). Springer-Verlag, New York. стр. 186-203.
6. Victor S. A. & David J. J. (2003). Polynomial Transformations of Tschirnhaus, Bring and Jerrard. URL: <https://www.uwo.ca/apmaths/faculty/jeffrey/pdfs/Adamchik.pdf> [Электронный ресурс]. стр. 90-91. Дата обращения: 14.02.2023.
7. National Bureau of Standards, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables Ed. by Milton Abramowitz and Irene A. Stegun (1964). Tenth printing with corrections, U.S. Government Printing Office, Washington, D.C., 1972, URL: https://personal.math.ubc.ca/~cbm/aands/abramowitz_and_stegun.pdf [Электронный ресурс]. стр. 96-128. Дата обращения: 10.04.2023.
8. A novel Tschirnhaus method to get only the true solutions of quartic equations. Raghavendra G. Kulkarni. PES University, India 2018. URL: <https://dialnet.unirioja.es/descarga/articulo/6523979.pdf> [Электронный ресурс]. стр. 6-8. Дата обращения: 13.04.2023.
9. Solution of quartic equations. William Squire. International Journal of Mathematical Education in Science and Technology. 1979. URL:

- <https://www.tandfonline.com/doi/abs/10.1080/0020739790100223> [Электронный ресурс]. стр. 299-301. Дата обращения: 18.04.2023.
10. The fast quartic solver. Peter Strobach. 2010, Journal of Computational and Applied Mathematics. URL: https://www.academia.edu/es/27122162/The_fast_quartic_solver [Электронный ресурс]. стр. 8-24. Дата обращения: 18.04.2023.
11. The Deduction of Final Formulas for the Algebraic Solution of the Quartic Equation. Mansfield Merriman. American Journal of Mathematics, Vol. 14, No. 3 (1892). URL: <https://www.jstor.org/stable/2369666> [Электронный ресурс]. стр. 237-245. Дата обращения: 23.04.2023.
12. A UNIFIED APPROACH FOR SOLVING QUADRATIC, CUBIC AND QUARTIC EQUATIONS BY RADICALS. A. A. UNGAR. Department of Mathematics, North Dakota State University, Fargo, ND 58105, U.S.A. 1989. URL: <https://core.ac.uk/download/pdf/82351384.pdf> [Электронный ресурс]. стр. 35-39. Дата обращения: 29.04.2023.
13. A Note on the Solution of Quartic Equations. Herbert E. Salzer. 1960. URL: <https://www.ams.org/journals/mcom/1960-14-071/S0025-5718-1960-0117882-6/S0025-5718-1960-0117882-6.pdf> [Электронный ресурс]. стр.279-281. Дата обращения: 01.05.2023.

Приложение

1. Исходный код имплементации алгоритмов поиска корней уравнения четвертой степени (файл polynomial.cpp)

```
/*
    Имплементация метода решения уравнения четвертой степени – Ferrari's Method
    Информация о методе – https://quarticequations.com/Quartic2.pdf (стр. 3)
    Работу выполнил – Погосов Даниэль (https://github.com/DarklIleS)
*/
template<typename fp_t>
unsigned int ferrari(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>& roots)
{
    // Нормировка коэффициентов
    if (isZero(n) || isinf(a /= n))
        return solveCubic(a, b, c, d, roots);
    if (isinf(b /= n))
        return 0;
    if (isinf(c /= n))
        return 0;
    if (isinf(d /= n))
        return 0;

    // Объявление констант
    static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
    static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
    static constexpr fp_t EPS = numeric_limits<fp_t>::epsilon();

    // Количество вещественных корней
    unsigned numberOfRoots = 0;

    // Вычисляем расчетные коэффициенты
    fp_t C = a * ONE_QUARTER;
    fp_t a_ = fma(static_cast<fp_t>(-6.0L) * C, C, b); // b - 6 * C^2
    fp_t b_ = fma(fms(static_cast<fp_t>(8.0L) * C, C, static_cast<fp_t>(2.0L)), b),
    C, C); // c - 2 * b * C + 8 * C^3
    fp_t c_ = fma(fma(static_cast<fp_t>(-3.0L) * C, C, b), C, -c), C, d); // d
    - c * C + b * C^2 - 3 * C^4

    // Решаем резольвентное кубическое уравнение вида: m^3 + a_ * m^2 + (a_^2 / 4 -
    c_) * m - b_^2 / 8 = 0
    vector<fp_t> cubicRoots(3);
    unsigned numberOfCubicRoots = solveCubic(static_cast<fp_t>(1.0L), a_,
    fma(ONE_QUARTER * a_, a_, -c_), -b_ * b_ * static_cast<fp_t>(0.125L), cubicRoots);

    // Выбираем корень, который удовлетворяет условию: m > 0 и является
    вещественным корнем, иначе m = 0
    fp_t m = cubicRoots[numberOfCubicRoots - 1] > 0 ? cubicRoots[numberOfCubicRoots
    - 1] : static_cast<fp_t>(0.0L);
```

```

// Определяем знак радикала R
fp_t sigma = b_ > 0 ? static_cast<fp_t>(1.0L) : static_cast<fp_t>(-1.0L);

fp_t subR = fma(m, m, fma(fma(ONE_QUARTER, a_, m), a_, -c_)); // m^2 + (a_ / 4
- c_) * a_ - c_
// Если отрицательно, то имеем комплексный радикал, отсюда уравнение будет
иметь исключительно комплексное решение
if (subR < 0)
    return 0;

// Радикал R
fp_t R = sigma * sqrt(subR);

fp_t radicandPart = fms(-ONE_HALF, m, ONE_HALF, a_); // -m / 2 - a_ / 2

// Вычисляем радиканды будущего решения
fp_t radicand = radicandPart - R;
fp_t radicand_ = radicandPart + R;

fp_t sqrtM = sqrt(m * ONE_HALF);

fp_t rootPart = sqrtM - C;

// Если полученный радиканд (radicand) > 0 либо близок к 0, то уравнение имеет
два либо больше вещественных решений
if (radicand >= 0)
{
    fp_t radical = sqrt(radicand);

    roots =
    {
        rootPart + radical,
        rootPart - radical
    };

    number_of_roots += 2;
}
else if (abs(radicand) <= EPS)
{
    roots =
    {
        rootPart,
        rootPart
    };

    number_of_roots += 2;
}

rootPart = -sqrtM - C;

```

```

// Если полученный радиканд (radicand_) > 0 либо близок к 0, то уравнение имеет
// два либо больше вещественных решений
if (radicand_ >= 0)
{
    fp_t radical = sqrt(radicand_);

    roots[numberOfRoots] = rootPart + radical;
    roots[numberOfRoots + 1] = rootPart - radical;

    numberOfRoots += 2;
}
else if (abs(radicand_) <= EPS)
{
    roots[numberOfRoots] = rootPart;
    roots[numberOfRoots + 1] = rootPart;

    numberOfRoots += 2;
}

return numberOfRoots;
}

/*
Имплементация метода решения уравнения четвертой степени – Descartes' Method
Информация о методе – https://quarticequations.com/Quartic2.pdf (стр. 4)
Работу выполнил – Погосов Даниэль (https://github.com/Darklles)
*/
template<typename fp_t>
unsigned int descartes(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>& roots)
{
    // Нормировка коэффициентов
    if (isZero(n) || isinf(a /= n))
        return solveCubic(a, b, c, d, roots);
    if (isinf(b /= n))
        return 0;
    if (isinf(c /= n))
        return 0;
    if (isinf(d /= n))
        return 0;

    // Объявление констант
    static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
    static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
    static constexpr fp_t EPS = numeric_limits<fp_t>::epsilon();

    // Количество вещественных корней
    unsigned numberOfRoots = 0;

    // Вычисляем расчетные коэффициенты
    fp_t C = a * ONE_QUARTER;
}

```

```

    fp_t a_ = fma(static_cast<fp_t>(-6.0L) * C, C, b); // b - 6 * C^2
    fp_t b_ = fma(fms(static_cast<fp_t>(8.0L) * C, C, static_cast<fp_t>(2.0L), b),
C, c); // c - 2 * b * C + 8 * C^3
    fp_t c_ = fma(fma(static_cast<fp_t>(-3.0L) * C, C, b), C, -c), C, d); // d
- c * C + b * C^2 - 3 * C^4

    // Решаем резольвентное бикубическое уравнение вида: y^6 + 2 * a_ * y^4 + (a_^2
- 4 * c_) * y^2 - b_^2 = 0
    vector<fp_t> cubicRoots(3);
    unsigned number_of_cubicRoots = solveCubic(static_cast<fp_t>(1.0L),
static_cast<fp_t>(2.0L) * a_, fms(a_, a_, static_cast<fp_t>(4.0L), c_), -b_ * b_,
cubicRoots);

    // Выбираем корень, который удовлетворяет условию: y^2 > 0 и является
вещественным корнем, иначе y^2 = 0
    fp_t yy = cubicRoots[number_of_cubicRoots - 1] > 0 ?
cubicRoots[number_of_cubicRoots - 1] : static_cast<fp_t>(0.0L);

    // Положительный вещественный корень бикубического уравнения
    fp_t y = sqrt(yy);

    // Определяем знак радикала R
    fp_t sigma = b_ > 0 ? static_cast<fp_t>(1.0L) : static_cast<fp_t>(-1.0L);

    fp_t subR = fma(ONE_QUARTER * a_, a_, fma(fms(ONE_QUARTER, yy, -ONE_HALF, a_),
yy, -c_)); // a_^2 / 4 + ((y^2 / 4 + a_ / 2) * y^2 - c_)
    // Если отрицательно, то имеем комплексный радикал, отсюда уравнение будет
иметь исключительно комплексное решение
    if (subR < 0)
        return 0;

    // Радикал R
    fp_t R = sigma * sqrt(subR);

    fp_t radicandPart = fms(-ONE_QUARTER, yy, ONE_HALF, a_); // -y^2 / 4 - a_ / 2

    // Вычисляем радиканды будущего решения
    fp_t radicand = radicandPart - R;
    fp_t radicand_ = radicandPart + R;

    fp_t rootPart = fma(ONE_HALF, y, -C); // y / 2 - C

    // Если полученный радиканд (radicand) > 0 либо близок к 0, то уравнение имеет
два либо больше вещественных решений
    if (radicand >= 0)
    {
        fp_t radical = sqrt(radicand);

        roots =
    {

```

```

        rootPart + radical,
        rootPart - radical
    };

    numberOfRoots += 2;
}
else if (abs(radicand) <= EPS)
{
    roots =
    {
        rootPart,
        rootPart
    };

    numberOfRoots += 2;
}

rootPart = fma(-ONE_HALF, y, -C); // -y / 2 - C

// Если полученный радиканд (radicand) > 0 либо близок к 0, то уравнение имеет
два либо больше вещественных решений
if (radicand_ >= 0)
{
    fp_t radical = sqrt(radicand_);

    roots[numberOfRoots] = rootPart + radical;
    roots[numberOfRoots + 1] = rootPart - radical;

    numberOfRoots += 2;
}
else if (abs(radicand_) <= EPS)
{
    roots[numberOfRoots] = rootPart;
    roots[numberOfRoots + 1] = rootPart;

    numberOfRoots += 2;
}

return numberOfRoots;
}

/*
Имплементация метода решения уравнения четвертой степени – NBS Method
Информация о методе – https://quarticequations.com/Quartic2.pdf (стр. 5)
Работу выполнил – Погосов Даниэль (https://github.com/Darklles)
*/
template<typename fp_t>
unsigned int nbs(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>& roots)
{
    // Нормировка коэффициентов
}

```

```

if (isZero(n) || isinf(a /= n))
    return solveCubic(a, b, c, d, roots);
if (isinf(b /= n))
    return 0;
if (isinf(c /= n))
    return 0;
if (isinf(d /= n))
    return 0;

// Объявление констант
static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
static constexpr fp_t EPS = numeric_limits<fp_t>::epsilon();

// Количество вещественных корней
unsigned numberOfRoots = 0;

// Вычисляем расчетные коэффициенты
fp_t a_ = -b;
fp_t b_ = fms(a, c, static_cast<fp_t>(4.0L), d); // a * c - 4 * d
fp_t c_ = fms(fms(static_cast<fp_t>(4.0L), b, a, a), d, c, c); // (4 * b - a^2)
* d - c^2

// Решаем резольвентное кубическое уравнение вида: u^3 + a_ * u^2 + b_ * u + c_ = 0
vector<fp_t> cubicRoots(3);
unsigned numberOfCubicRoots = solveCubic(static_cast<fp_t>(1.0L), a_, b_, c_, cubicRoots);

// Выбираем наибольший вещественный корень
fp_t u = cubicRoots[numberOfCubicRoots - 1];

// Определяем знак радиканов
fp_t sigma = fma(-ONE_HALF * a, u, c) > 0 ? static_cast<fp_t>(1.0L) :
static_cast<fp_t>(-1.0L);

// Вычисляем радиканды переменных p и q
fp_t radicandP = fma(ONE_QUARTER * a, a, u) - b; // a^2 / 4 + u - b
fp_t radicandQ = fma(ONE_QUARTER * u, u, -d); // u^2 / 4 - d

if (abs(radicandP) <= EPS)
{
    radicandP = static_cast<fp_t>(0.0L);
}

if (abs(radicandQ) <= EPS)
{
    radicandQ = static_cast<fp_t>(0.0L);
}

```

```

// Если хоть один из радиканов отрицателен, то имеем исключительно комплексное
решение
if (radicandP < 0 || radicandQ < 0)
    return 0;

fp_t radicalP = sqrt(radicandP);
fp_t radicalQ = sqrt(radicandQ);

// Вычисляем итоговые расчетные коэффициенты p_n и q_n (n = 1, 2)
fp_t p1 = fma(ONE_HALF, a, -radicalP); // a / 2 - radicalP
fp_t p2 = fma(ONE_HALF, a, radicalP); // a / 2 + radicalP
fp_t q1 = fms(ONE_HALF, u, -sigma, radicalQ); // u / 2 + sigma * radicalQ
fp_t q2 = fms(ONE_HALF, u, sigma, radicalQ); // u / 2 - sigma * radicalQ

// Вычисляем радиканы будущего решения
fp_t radicand = fma(ONE_QUARTER * p1, p1, -q1); // p1^2 / 4 - q1
fp_t radicand_ = fma(ONE_QUARTER * p2, p2, -q2); // p2^2 / 4 - q2

// Если полученный радиканд (radicand) > 0 либо близок к 0, то уравнение имеет
два либо больше вещественных решений
if (radicand >= 0)
{
    fp_t radical = sqrt(radicand);

    roots =
    {
        fma(-ONE_HALF, p1, radical), // -p1 / 2 + radical
        fma(-ONE_HALF, p1, -radical) // -p1 / 2 - radical
    };

    numberOfRoots += 2;
}
else if (abs(radicand) <= EPS)
{
    fp_t root = -p1 * ONE_HALF;

    roots =
    {
        root,
        root
    };

    numberOfRoots += 2;
}

// Если полученный радиканд (radicand_) > 0 либо близок к 0, то уравнение имеет
два либо больше вещественных решений
if (radicand_ >= 0)
{
    fp_t radical = sqrt(radicand_);
}

```

```

        roots[numberOfRoots] = fma(-ONE_HALF, p2, radical); // -p2 / 2 + radical
        roots[numberOfRoots + 1] = fma(-ONE_HALF, p2, -radical); // -p2 / 2 - radical

        numberOfRoots += 2;
    }
    else if (abs(radicand_) <= EPS)
    {
        fp_t root = -p2 * ONE_HALF;

        roots[numberOfRoots] = root;
        roots[numberOfRoots + 1] = root;

        numberOfRoots += 2;
    }

    return numberOfRoots;
}

/*
Имплементация метода решения уравнения четвертой степени – Euler's Method
Информация о методе – https://quarticequations.com/Quartic2.pdf (стр. 6)
Работу выполнил – Погосов Даниэль (https://github.com/DarklleS)
*/
template<typename fp_t>
unsigned int euler(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>& roots)
{
    // Нормировка коэффициентов
    if (isZero(n) || isinf(a /= n))
        return solveCubic(a, b, c, d, roots);
    if (isinf(b /= n))
        return 0;
    if (isinf(c /= n))
        return 0;
    if (isinf(d /= n))
        return 0;

    // Объявление констант
    static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
    static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
    static constexpr fp_t EPS = numeric_limits<fp_t>::epsilon();

    // Количество вещественных корней
    unsigned numberOfRoots = 0;

    // Вычисляем расчетные коэффициенты
    fp_t C = a * ONE_QUARTER;
    fp_t a_ = fma(static_cast<fp_t>(-6.0L) * C, C, b); // b - 6 * C^2
}

```

```

        fp_t b_ = fma(fms(static_cast<fp_t>(8.0L) * c, c, static_cast<fp_t>(2.0L), b),
c, c); // c - 2 * b * c + 8 * c^3
        fp_t c_ = fma(fma(static_cast<fp_t>(-3.0L) * c, c, b), c, -c), c, d); // d
- c * c + b * c^2 - 3 * c^4

        // Решаем резольвентное кубическое уравнение вида: r^3 + a_ / 2 * r^2 + (a_^2 -
4 * c_) / 16 * r - b_^2 / 64 = 0
        vector<complex<fp_t>> cubicRoots(3);
        unsigned numberOfCubicRoots = solveCubic(static_cast<fp_t>(1.0L), a_ *
ONE_HALF, fms(static_cast<fp_t>(0.0625L) * a_, a_, ONE_QUARTER, c_), -b_ * b_ *
static_cast<fp_t>(0.015625L), cubicRoots);

        // Определяем знак радикандов
        fp_t sigma = b_ > 0 ? static_cast<fp_t>(1.0L) : static_cast<fp_t>(-1.0L);

        // Объявление вычислительный переменных для поиска радикандов
        fp_t r;
        fp_t x2, x3, y;
        fp_t subradicand;
        fp_t radicand, radicand_;

        // Вычисление радикандов. Разбираем 2 случая дабы оптимизировать вычисления
        // - Если решение резольвентного кубического уравнения имеет единственный
вещественный корень
        if (numberOfCubicRoots == 1)
{
    r = cubicRoots[0].real();
    if (r < 0)
        return 0;

    x2 = cubicRoots[1].real();
    x3 = x2;
    y = cubicRoots[1].imag();

    subradicand = fms(x2, x3, -y, y); // x2 * x3 - y^2
    // Если отрицательно, то имеем комплексные радиканды, отсюда уравнение
будет иметь исключительно комплексное решение
    if (subradicand < 0)
        return 0;

    fp_t subRadical = sqrt(subradicand);

    radicand = fms(static_cast<fp_t>(2.0L), x2, static_cast<fp_t>(2.0L) *
sigma, subRadical); // 2 * x2 - 2 * sigma * subRadical
    radicand_ = fms(static_cast<fp_t>(2.0L), x2, static_cast<fp_t>(-2.0L) *
sigma, subRadical); // 2 * x2 + 2 * sigma * subRadical
}
// - Иначе все корни резольвентного кубического уравнения вещественные
else
{

```

```

r = cubicRoots[2].real();
if (r < 0)
    return 0;

x2 = cubicRoots[0].real();
x3 = cubicRoots[1].real();
y = static_cast<fp_t>(0.0L);

subradicand = x2 * x3;
// Если отрицательно, то имеем комплексные радиканды, отсюда уравнение
будет иметь исключительно комплексное решение
if (subradicand < 0)
    return 0;

fp_t subRadical = sqrt(subradicand);

radicand = fma(static_cast<fp_t>(-2.0L) * sigma, subRadical, x2) + x3; // -
2 * sigma * subRadical + x2 + x3
    radicand_ = fma(static_cast<fp_t>(2.0L) * sigma, subRadical, x2) + x3; // 2
* sigma * subRadical + x2 + x3
}

fp_t rootPart = sqrt(r) - C;

// Если полученный радиканд (radicand) > 0 либо близок к 0, то уравнение имеет
два либо больше вещественных решений
if (radicand >= 0)
{
    fp_t radical = sqrt(radicand);

    roots =
    {
        rootPart + radical,
        rootPart - radical
    };

    numberOfRoots += 2;
}
else if (abs(radicand) <= EPS)
{
    roots =
    {
        rootPart,
        rootPart
    };

    numberOfRoots += 2;
}

rootPart = -sqrt(r) - C;

```

```

    // Если полученный радиканд (radicand_) > 0 либо близок к 0, то уравнение имеет
    // два либо больше вещественных решений
    if (radicand_ >= 0)
    {
        fp_t radical = sqrt(radicand_);

        roots[numberOfRoots] = rootPart + radical;
        roots[numberOfRoots + 1] = rootPart - radical;

        numberOfRoots += 2;
    }
    else if (abs(radicand_) <= EPS)
    {
        roots[numberOfRoots] = rootPart;
        roots[numberOfRoots + 1] = rootPart;

        numberOfRoots += 2;
    }

    return numberOfRoots;
}

/*
    Имплементация метода решения уравнения четвертой степени – Van der Waerden's
Method
    Информация о методе – https://quarticequations.com/Quartic2.pdf (стр. 7)
    Работу выполнил – Погосов Даниэль (https://github.com/Darklles)
*/
template<typename fp_t>
unsigned int vanDerWaerden(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>&
roots)
{
    // Нормировка коэффициентов
    if (isZero(n) || isinf(a /= n))
        return solveCubic(a, b, c, d, roots);
    if (isinf(b /= n))
        return 0;
    if (isinf(c /= n))
        return 0;
    if (isinf(d /= n))
        return 0;

    // Объявление констант
    static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
    static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
    static constexpr fp_t EPS = numeric_limits<fp_t>::epsilon();

    // Количество вещественных корней
    unsigned numberOfRoots = 0;

```

```

// Вычисляем расчетные коэффициенты
fp_t C = a * ONE_QUARTER;
fp_t a_ = fma(static_cast<fp_t>(-6.0L) * C, C, b); // b - 6 * C^2
fp_t b_ = fms(static_cast<fp_t>(8.0L) * C, C, static_cast<fp_t>(2.0L), b),
C, c); // c - 2 * b * C + 8 * C^3
fp_t c_ = fma(fma(static_cast<fp_t>(-3.0L) * C, C, b), C, -c), C, d); // d
- c * C + b * C^2 - 3 * C^4

// Решаем резольвентное кубическое уравнение вида: theta^3 - 2 * a_ * theta^2 +
(a_^2 - 4 * c_) * theta + b_ ^2 = 0
vector<complex<fp_t>> cubicRoots(3);
unsigned numberOfCubicRoots = solveCubic(static_cast<fp_t>(1.0L),
static_cast<fp_t>(-2.0L) * a_, fms(a_, a_, static_cast<fp_t>(4.0L), c_), b_ * b_,
cubicRoots);

// Определяем знак радикандов
fp_t sigma = b_ > 0 ? static_cast<fp_t>(1.0L) : static_cast<fp_t>(-1.0L);

// Объявление вычислительный переменных для поиска радикандов
fp_t theta;
fp_t x2, x3, y;
fp_t subradicand;
fp_t radicand, radicand_;

// Вычисление радикандов. Разбираем 2 случая дабы оптимизировать вычисления
// - Если решение резольвентного кубического уравнения имеет единственный
вещественный корень
if (numberOfCubicRoots == 1)
{
    theta = cubicRoots[0].real();
    if (theta > 0)
        return 0;

    x2 = cubicRoots[1].real();
    x3 = x2;
    y = cubicRoots[1].imag();

    subradicand = fms(x2, x3, -y, y); // x2 * x3 - y^2
    // Если отрицательно, то имеем комплексные радиканды, отсюда уравнение
будет иметь исключительно комплексное решение
    if (subradicand < 0)
        return 0;

    fp_t subRadical = sqrt(subradicand);

    radicand = fms(static_cast<fp_t>(-2.0L), x2, static_cast<fp_t>(2.0L) *
sigma, subRadical); // -2 * x2 - 2 * sigma * subRadical
    radicand_ = fms(static_cast<fp_t>(-2.0L), x2, static_cast<fp_t>(-2.0L) *
sigma, subRadical); // -2 * x2 + 2 * sigma * subRadical

```

```

}

// - Иначе все корни резольвентного кубического уравнения вещественные
else
{
    theta = cubicRoots[0].real();
    if (theta > 0)
        return 0;

    x2 = cubicRoots[1].real();
    x3 = cubicRoots[2].real();
    y = static_cast<fp_t>(0.0L);

    subradicand = x2 * x3;
    // Если отрицательно, то имеем комплексные радиканды, отсюда уравнение
    // будет иметь исключительно комплексное решение
    if (subradicand < 0)
        return 0;

    fp_t subRadical = sqrt(subradicand);

    radicand = fma(static_cast<fp_t>(-2.0L) * sigma, subRadical, -x2) - x3; // -2 * sigma * subRadical - x2 - x3
    radicand_ = fma(static_cast<fp_t>(2.0L) * sigma, subRadical, -x2) - x3; // 2 * sigma * subRadical - x2 - x3
}

fp_t sqrtTheta = sqrt(-theta);

fp_t rootPart = fma(ONE_HALF, sqrtTheta, -C);

// Если полученный радиканд (radicand) > 0 либо близок к 0, то уравнение имеет
// два либо больше вещественных решений
if (radicand >= 0)
{
    fp_t radical = sqrt(radicand);

    roots =
    {
        fma(ONE_HALF, radical, rootPart), // radical / 2 + rootPart
        fma(-ONE_HALF, radical, rootPart) // -radical / 2 + rootPart
    };

    numberOfRoots += 2;
}
else if (abs(radicand) <= EPS)
{
    roots =
    {
        rootPart,
        rootPart
}

```

```

        };

        numberOfRoots += 2;
    }

    rootPart = fma(-ONE_HALF, sqrtTheta, -C);

    // Если полученный радиканд (radicand_) > 0 либо близок к 0, то уравнение имеет
    // два либо больше вещественных решений
    if (radicand_ >= 0)
    {
        fp_t radical = sqrt(radicand_);

        roots[numberOfRoots] = fma(ONE_HALF, radical, rootPart); // radical / 2 +
rootPart
        roots[numberOfRoots + 1] = fma(-ONE_HALF, radical, rootPart); // -radical /
2 + rootPart

        numberOfRoots += 2;
    }
    else if (abs(radicand_) <= EPS)
    {
        roots[numberOfRoots] = rootPart;
        roots[numberOfRoots + 1] = rootPart;

        numberOfRoots += 2;
    }
}

return numberOfRoots;
}

/*
Имплементация метода решения уравнения четвертой степени – Tschirnhaus's Method
Информация о методе – https://dialnet.unirioja.es/descarga/articulo/6523979.pdf
Работу выполнил – Погосов Даниэль (https://github.com/Darklles)
*/
template<typename fp_t>
unsigned int tschirnhaus(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>&
roots)
{
    // Нормировка коэффициентов
    if (isZero(n))
        return solveCubic(a, b, c, d, roots);
    if (isinf(a /= n) || isinf(b /= n) || isinf(c /= n) || isinf(d /= n))
        return 0;

    // Объявление констант
    static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
    static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
    static constexpr fp_t EPS = numeric_limits<fp_t>::epsilon();
}

```

```

        static const complex<fp_t> TWO_C(static_cast<fp_t>(2.0L),
static_cast<fp_t>(0.0L));
        static const complex<fp_t> ONE_HALF_C(static_cast<fp_t>(0.5L),
static_cast<fp_t>(0.0L));

        // Вычисляем расчетные коэффициенты
        fp_t C = a * ONE_QUARTER;
        fp_t a_ = fma(static_cast<fp_t>(-6.0L) * C, C, b); // b - 6 * C^2
        fp_t b_ = fma(fms(static_cast<fp_t>(8.0L) * C, C, static_cast<fp_t>(2.0L), b),
C, c); // c - 2 * b * C + 8 * C^3
        fp_t c_ = fma(fma(static_cast<fp_t>(-3.0L) * C, C, b), C, -c), C, d); // d
- c * C + b * C^2 - 3 * C^4

        // Счетчик количества вещественных корней
unsigned numberOfRoots = 0;

        // Вычисляем коэффициенты для бикубического резольвентного уравнения
        fp_t A_ = a_ * ONE_HALF;
        fp_t B_ = -c_;
        fp_t C_ = fms(b_, b_, static_cast<fp_t>(4.0L) * a_, c_) *
static_cast<fp_t>(0.125L);

        // Решаем резольвентное бикубическое уравнение вида: f^6 + a_/2 * f^4 - c_ *
f^2 + (b_^2 - 4 * a_ * c_) / 8 = 0
        vector<complex<fp_t>> cubicRoots(3);
        unsigned numberOfCubicRoots = solveCubic(static_cast<fp_t>(1.0L), A_, B_, C_,
cubicRoots);

        // Определяем решение (наибольшее) бикубического уравнения
        complex<fp_t> ff = cubicRoots[numberOfCubicRoots - 1];
        complex<fp_t> f = sqrt(ff);

        if (isnan(ff.real()) || isnan(ff.imag()))
            return 0;

        // Определение вспомогательных коэффициентов
        complex<fp_t> k = fmac(TWO_C, ff, complex<fp_t>(a_, static_cast<fp_t>(0.0L)));
// 2 * f^2 + a_
        complex<fp_t> sqrtK = sqrt(-k);

        // Вычисляем коэффициенты для квадратных вспомогательных уравнений
        complex<fp_t> A = fmac(TWO_C, f, -sqrtK); // 2 * f - sqrtK
        complex<fp_t> B = fmse(f, k, ONE_HALF_C, complex<fp_t>(-b_,
static_cast<fp_t>(0.0L))) * sqrtK / k; // (f * k - b_ / 2) * sqrtK / (2 * k)

        if (isnan(B.real()))
{
    return 0;
}

```

```

        // Решаем резольвентное бикубическое уравнение вида: d^2 + (2 * f - sqrt(-k)) *
f - sqrt(-k) * ((2 * f * k + b) / (2 * k)) = 0
        vector<complex<fp_t>> quadraticRoots(2);
        unsigned numberQuadraticRoots =
solveQuadratic(complex<fp_t>(static_cast<fp_t>(1.0L), static_cast<fp_t>(0.0L)), A,
-B, quadraticRoots);

        vector<complex<fp_t>> rootsComplex =
{
    quadraticRoots[0] + f - C,
    quadraticRoots[1] + f - C
};

// Отбрасываем комплексные решения
selectRealRoots(rootsComplex, numberRoots, roots);

// Переопределяем коэффициент при d для нового квадратного уравнения
A = fmac(TWO_C, f, sqrtK);

// Решаем резольвентное бикубическое уравнение вида: d^2 + (2 * f + sqrt(-k)) *
f + sqrt(-k) * ((2 * f * k + b) / (2 * k)) = 0
        numberQuadraticRoots = solveQuadratic(complex<fp_t>(static_cast<fp_t>(1.0L),
static_cast<fp_t>(0.0L)), A, B, quadraticRoots);

        rootsComplex =
{
    quadraticRoots[0] + f - C,
    quadraticRoots[1] + f - C
};

// Отбрасываем комплексные решения
selectRealRoots(rootsComplex, numberRoots, roots);

        return numberRoots;
}

/*
Имплементация метода решения уравнения четвертой степени –
A_Note_on_the_Solution_of_Quartic_Equations-Salzer-1960
Информация о методе – https://www.ams.org/journals/mcom/1960-14-071/S0025-5718-1960-0117882-6/S0025-5718-1960-0117882-6.pdf
Работу выполнил – Погосов Даниэль (https://github.com/Darklles)
*/
template<typename fp_t>
unsigned int salzer(fp_t N, fp_t A, fp_t B, fp_t C, fp_t D, vector<fp_t>& roots)
{
    // Нормировка исходных коэффициентов, где N - старший коэффициент уравнения
    if (isZero(N) || isinf(A /= N))
        return solveCubic(A, B, C, D, roots);
    if (isinf(B /= N))

```

```

        return 0;
    if (isinf(C /= N))
        return 0;
    if (isinf(D /= N))
        return 0;

    // Количество вещественных корней
    unsigned int numberOfRoots = 0;

    // Вычисление коэффициентов кубического уравнения
    fp_t a = -B;
    fp_t b = fms(A, C, static_cast<fp_t>(4), D); // A * C - 4 * D
    fp_t c = fms(D, fms(static_cast<fp_t>(4), B, A, A), C, C); // D * (4 * B - A^2)
    - C^2
    vector<fp_t> cubicRoots(3);

    // Решение резольвентного кубического уравнения вида: x^3 + a * x^2 + b * x + c
    = 0
    solveCubic(static_cast<fp_t>(1), a, b, c, cubicRoots);

    // Вещественный корень кубического уравнения
    fp_t x = cubicRoots[0];

    // Вычисление начальных расчетных коэффициентов
    fp_t m;
    fp_t n;
    fp_t mm = fma(static_cast<fp_t>(0.25L) * A, A, -B) + x; // A^2 / 4 - B + c

    if (mm > 0)
    {
        m = sqrt(mm);
        n = static_cast<fp_t>(0.25L) * fms(A, x, static_cast<fp_t>(2), C) / m; // (A * x - 2 * C) / (2 * m)
    }
    else if (isZero(mm))
    {
        m = 0;
        n = sqrt(fma(static_cast<fp_t>(0.25L) * x, x, -D)); // sqrt(x^2 / 4 - D)
    }
    else // m - комплексное, следовательно уравнение не будет иметь вещественных
корней
    {
        return 0;
    }

    // Вычисление расчетных коэффициентов
    fp_t alpha = fma(static_cast<fp_t>(0.5L) * A, A, -x) - B; // A^2 / 2 - x - B
    fp_t beta = fms(static_cast<fp_t>(4), n, A, m); // 4 * n - A * m
    fp_t gamma = alpha + beta;
    fp_t delta = alpha - beta;

```

```

    if (gamma >= 0) // Если gamma >= 0, то уравнение имеет два либо больше
    вещественных корней
    {
        gamma = sqrt(gamma);

        roots[0] = fma(static_cast<fp_t>(0.5L), gamma, fms(static_cast<fp_t>(0.5L),
m, static_cast<fp_t>(0.25L), A)); // gamma / 2 + (m / 2 - A / 4)
        roots[1] = fma(static_cast<fp_t>(-0.5L), gamma,
fms(static_cast<fp_t>(0.5L), m, static_cast<fp_t>(0.25L), A)); // -gamma / 2 + (m /
2 - A / 4)

        numberOfRoots += 2;
    }
    if (delta >= 0) // Если delta >= 0, то уравнение имеет два либо больше
    вещественных корней
    {
        delta = sqrt(delta);

        roots[numberOfRoots] = fma(static_cast<fp_t>(0.5L), delta,
fms(static_cast<fp_t>(-0.5L), m, static_cast<fp_t>(0.25L), A)); // delta / 2 + (-m
/ 2 - A / 4)
        roots[numberOfRoots + 1] = fma(static_cast<fp_t>(-0.5L), delta,
fms(static_cast<fp_t>(-0.5L), m, static_cast<fp_t>(0.25L), A)); // -delta / 2 + (-m
/ 2 - A / 4)

        numberOfRoots += 2;
    }

    return numberOfRoots;
}

/*
    Имплементация метода решения уравнения четвертой степени – Mansfield Merriman

    Информация о методе – https://www.jstor.org/stable/2369666
    Работу выполнил – Погосов Даниэль (https://github.com/DarklileS)
*/
template<typename fp_t>
unsigned int merriman(fp_t n, fp_t a_, fp_t b_, fp_t c_, fp_t d_, vector<fp_t>&
roots)
{
    // Нормировка коэффициентов
    if (isZero(n) || isinf(a_ /= n))
        return solveCubic(a_, b_, c_, d_, roots);
    if (isinf(b_ /= n))
        return 0;
    if (isinf(c_ /= n))
        return 0;
    if (isinf(d_ /= n))
        return 0;
}

```

```

// Объявление констант
static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
static const fp_t ONE_QUARTER = static_cast<fp_t>(0.25L);
static const fp_t NINE_QUARTER = static_cast<fp_t>(2.25L);
static const fp_t ONE_SIXTH = static_cast<fp_t>(1.0L / 6.0L);
static const fp_t ONE_THIRD = static_cast<fp_t>(1.0L / 3.0L);
static const fp_t FOUR_THIRD = static_cast<fp_t>(4.0L / 3.0L);
static const fp_t TWO = static_cast<fp_t>(2.0L);
static const fp_t THREE = static_cast<fp_t>(3.0L);
static const fp_t SIX = static_cast<fp_t>(6.0L);
static const fp_t NINE = static_cast<fp_t>(9.0L);
static const fp_t EPS = static_cast<fp_t>(1e-6);

// Пересчёт переменных
fp_t a = a_ * ONE_QUARTER;
fp_t b = b_ * ONE_SIXTH;
fp_t c = c_ * ONE_QUARTER;
fp_t d = d_;

// Количество вещественных корней
unsigned numberOfRoots = 0;
//vector<fp_t> roots(4);

// Вычисляем расчетные коэффициенты
fp_t m = fma(b * b, b, fms(d, fma(a, a, -b), -c, fma(-2 * a, b, c))); // a^2 *
d + b^3 + c^2 - 2abc - bd
fp_t nn = pow(fma(b, b, fms(ONE_THIRD, d, FOUR_THIRD * a, c)), THREE); // (b^2
+ 1/3d - 4/3ac)^3

// переменная для разветвления логики m^2 - n: приводит к разным u,v,w
fp_t radical = fma(m, m, -nn);
// переменная для разветвления логики 2a^3 - 3ab + c: определяет знаки в корнях
fp_t cond_coef = fms(TWO, pow(a, THREE), THREE, a * b) + c; // 2 * a^3 - 3 * a
* b + c

if (radical > 0) {
    fp_t s1 = m + sqrt(radical);
    fp_t s = ONE_HALF * pow(s1, ONE_THIRD); // 1/2 (m + sqrt(m^2 - n))^1/3

    fp_t j = (m - sqrt(radical) < 0) ? -1 : 1; // определяем знак для t1
    fp_t t1 = j * (m - sqrt(radical));

    fp_t t = j * ONE_HALF * pow(t1, ONE_THIRD); // 1/2 (m - sqrt(m^2 - n))^1/3

    // Вычисляем коэффициенты решения
    fp_t u = fma(a, a, -b) + s + t; // a^2 - b + s + t
    fp_t v = fms(TWO, pow(a, TWO), TWO, b) - s - t; // 2a^2 - 2b - s - t
    fp_t w = fms(THREE, pow(s, TWO), SIX * s, t) +
        fma(v, v, THREE * pow(t, TWO)); // v^2 + 3s^2 - 6st + 3t^2
}

```

```

// вычисляем слагаемые исходного решения
fp_t roots_term_1 = sqrt(u);
fp_t roots_term_2 = sqrt(v + sqrt(w));
fp_t roots_term_3 = sqrt(v - sqrt(w));
// все корни комплексные
if (isnan(roots_term_1) || (isnan(roots_term_2) && isnan(roots_term_3))) {
    return 0;
}
// вещественные корни, 2 корня кратные
else if (isnan(roots_term_3)) {
    if (cond_coef < 0) {
        roots[0] = -a + roots_term_1 + roots_term_2;
        roots[1] = -a + roots_term_1 - roots_term_2;
        roots[2] = -a - roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
        roots[3] = -a - roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
    }
    else {
        roots[0] = -a - roots_term_1 - roots_term_2;
        roots[1] = -a - roots_term_1 + roots_term_2;
        roots[2] = -a + roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
        roots[3] = -a + roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
    }
}
// вещественные корни, 2 корня кратные
else if (isnan(roots_term_2)) {
    if (cond_coef < 0) {
        roots[0] = -a + roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
        roots[1] = -a + roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
        roots[2] = -a - roots_term_1 + roots_term_3;
        roots[3] = -a - roots_term_1 - roots_term_3;
    }
    else {
        roots[0] = -a - roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
        roots[1] = -a - roots_term_1; //УБРАЛИ МНИМУЮ ЧАСТЬ
        roots[2] = -a + roots_term_1 - roots_term_3;
        roots[3] = -a + roots_term_1 + roots_term_3;
    }
}
// вещественные корни, 4 различных корня
else {
    if (cond_coef < 0) {
        roots[0] = -a + roots_term_1 + roots_term_2;
        roots[1] = -a + roots_term_1 - roots_term_2;
        roots[2] = -a - roots_term_1 + roots_term_3;
        roots[3] = -a - roots_term_1 - roots_term_3;
    }
    else {
        roots[0] = -a - roots_term_1 - roots_term_2;
        roots[1] = -a - roots_term_1 + roots_term_2;
    }
}

```

```

        roots[2] = -a + roots_term_1 - roots_term_3;
        roots[3] = -a + roots_term_1 + roots_term_3;
    }
}

numberOfRoots = 4;
}

// если мнимая часть слишком мала (на уровне погрешности fp_t)
else if (abs(radical) <= EPS) {
    // вычисляем расчетные коэффициенты для исходного решения
    fp_t u = fma(a, a, -b) + pow(nn, ONE_SIXTH); // a^2 - b + n^1/6
    fp_t v = fms(TWO, pow(a, TWO), TWO, b) - pow(nn, ONE_SIXTH); // 2a^2 - 2b -
n^1/6

    if (isZero(v)) {
        // вычисляем слагаемое исходного решения
        fp_t roots_term = sqrt(fms(THREE, pow(a, TWO), THREE, b)); // sqrt(3 *
a^2 - 3 * b)

        if (cond_coef < 0) {
            roots[0] = -a + roots_term;
            roots[1] = roots[0];
            roots[2] = -a - roots_term;
            roots[3] = roots[2];
        }
        else {
            roots[0] = -a - roots_term;
            roots[1] = roots[0];
            roots[2] = -a + roots_term;
            roots[3] = roots[2];
        }
        numberOfRoots = 4;
    }
    else {
        // вычисляем слагаемые исходного решения
        fp_t roots_term_1 = sqrt(u);
        fp_t roots_term_2 = sqrt(TWO * v);

        if (cond_coef < 0) {
            roots[0] = -a + roots_term_1 + roots_term_2;
            roots[1] = -a + roots_term_1 - roots_term_2;
            roots[2] = -a - roots_term_1;
            roots[3] = roots[2];
        }
        else {
            roots[0] = -a - roots_term_1 - roots_term_2;
            roots[1] = -a - roots_term_1 + roots_term_2;
            roots[2] = -a + roots_term_1;
            roots[3] = roots[2];
        }
    }
}

```

```

        numberOfRoots = 4;
    }
}
else if (isZero(m)) {
    // вычисляем расчетные коэффициенты для исходного решения
    fp_t u = fma(a, a, -b); // a^2 - b
    fp_t v = fms(TWO, pow(a, TWO), TWO, b); // 2a^2 - 2b
    fp_t w = fms(v, v, THREE, pow(b, TWO)) - d; // v^2 - 3b^2 - d

    // вычисляем слагаемые исходного решения
    fp_t roots_term_1 = sqrt(u);
    fp_t roots_term_2 = sqrt(v + sqrt(w));
    fp_t roots_term_3 = sqrt(v - sqrt(w));

    if (isnan(roots_term_1) || (isnan(roots_term_2) && isnan(roots_term_3))) {
        return 0;
    }
    else if (isnan(roots_term_2)) {
        if (cond_coef < 0) {
            roots[0] = -a - roots_term_1;
            roots[1] = -a - roots_term_1;
            roots[2] = -a + roots_term_1 + roots_term_3;
            roots[3] = -a + roots_term_1 - roots_term_3;
        }
        else {
            roots[0] = -a + roots_term_1;
            roots[1] = -a + roots_term_1;
            roots[2] = -a - roots_term_1 - roots_term_3;
            roots[3] = -a - roots_term_1 + roots_term_3;
        }
    }
    else if (isnan(roots_term_3)) {
        if (cond_coef < 0) {
            roots[0] = -a - roots_term_1 + roots_term_2;
            roots[1] = -a - roots_term_1 - roots_term_2;
            roots[2] = -a + roots_term_1; //y
            roots[3] = -a + roots_term_1;
        }
        else {
            roots[0] = -a + roots_term_1 - roots_term_2;
            roots[1] = -a + roots_term_1 + roots_term_2;
            roots[2] = -a - roots_term_1;
            roots[3] = -a - roots_term_1;
        }
    }
    numberOfRoots = 4;
}
// не попали ни в 1 кейс -> не удалось найти корни аналитически, возвращаем 0
else {

```

```

        return 0;
    }

    return numberofRoots;
}

/*
Имплементация метода решения уравнения четвёртой степени – Solution of quartic
equations by William Squire
Информация о методе –
https://www.tandfonline.com/doi/abs/10.1080/0020739790100223
Работу выполнил – Погосов Даниэль (https://github.com/DarklileS)
*/
template <typename fp_t>
unsigned int squire(fp_t a, fp_t b, fp_t c, fp_t d, fp_t e, std::vector<fp_t>&
roots)
{
    // Нормировка коэффициентов
    if (isZero(a) || isinf(b /= a))
        return solveCubic(b, c, d, e, roots);
    if (isinf(c /= a))
        return 0;
    if (isinf(d /= a))
        return 0;
    if (isinf(e /= a))
        return 0;
    // переопределяем исходные коэффициенты
    fp_t a0 = e, a1 = d, a2 = c, a3 = b;
    unsigned int numberofRoots = 0;

    //if (abs(a0) > 1 or abs(a1) > 1 or abs(a2) > 1 or abs(a3) > 1) return 0;

    fp_t p; // переменная полиноме 6 степени
    fp_t bd = 0; // заданная граница приближения
    fp_t coef[6];

    // Вычисление коэффициентов полинома 6 степени функции S(p)
    coef[0] = pow(a0, 3);
    if (coef[0] < bd) bd = coef[0];
    coef[1] = -a2 * a0 * a0;
    if (coef[1] < bd) bd = coef[1];
    coef[2] = a0 * fma(a1, a3, -a0); // a0*(a1*a3-a0)
    if (coef[2] < bd) bd = coef[2];
    coef[3] = fma(-a0, a3 * a3, fms<fp_t>(2 * a0, a2, a1, a1)); // 2*a0*a2-a1^2 +-
a0*a3^2
    if (coef[3] < bd) bd = coef[3];
    coef[4] = fma(a1, a3, -a0); // a1*a3-a0
    if (coef[4] < bd) bd = coef[4];
    coef[5] = -a2;
    if (coef[5] < bd) bd = coef[5];
}

```

```

    // Функция S(p) =
    p^6+coef[5]*p^5+coef[5]*p^4+coef[4]*p^5+coef[3]*p^3+coef[2]*p^2+coef[1]*p+coef[0]
    auto fun = +[](fp_t p, fp_t* coef) {
        return static_cast<fp_t> (fma(fma(fma(fma(
            coef[5] + p, p, coef[4]), p, coef[3]), p, coef[2]), p, coef[1]), p,
    coef[0]));
    };

    // переопределяем границу приближения
    bd = 1 - bd;

    fp_t eps1 = eps_temp<fp_t>;

    if (a0 > 0) {
        // определяем решение уравнения 6-й степени, если a0 > 0 и abs(fun(p,
        coef)) < eps1
        p = sqrt(a0);
        // подставляем решение p в уравнений 6-й степени и сравниваем с величиной
        близкой к 0
        if (abs(fun(p, coef)) < eps1)
        {
            // находим корни исходного уравнения
            return findRoots(a3, a2, p, roots);
        }

        // находим корень методом бисекции (a0 > 0)
        p = refine<fp_t>(p, bd, coef, fun);
    }
    else p = refine<fp_t>(0, bd, coef, fun); // находим корень методом бисекции (a0
    <= 0)

    fp_t den = fma(p, p, -a0); // p*p-a0
    fp_t eps2 = eps_temp<fp_t>;

    if (abs(den) < eps2)
    {
        // находим корень методом бисекции
        return findRoots(a3, a2, p, roots);
    }

    // Нахождение коэффициентов при квадратных уравнениях
    fp_t p1 = a0 / p;
    fp_t q = p * fma(a3, p, -a1) / den; // p * (a3*p - a1) / den
    fp_t q1 = fms(a1, p, a3, a0) / den; // (a1*p - a3*a0) / den

    vector<fp_t> quadraticRoots(2);
    unsigned int nRoots = 0;

    // Нахождение корней квадратных уравнений

```

```

nRoots = solveQuadratic<fp_t>(1, q, p, quadraticRoots);
roots[numberofRoots] = quadraticRoots[0] * (nRoots > 0);
numberofRoots += (nRoots > 0);
roots[numberofRoots] = quadraticRoots[1] * (nRoots > 0);
numberofRoots += (nRoots > 0);

nRoots = solveQuadratic<fp_t>(1, q1, p1, quadraticRoots);
roots[numberofRoots] = quadraticRoots[0] * (nRoots > 0);
numberofRoots += (nRoots > 0);
roots[numberofRoots] = quadraticRoots[1] * (nRoots > 0);
numberofRoots += (nRoots > 0);

return numberofRoots;
}

/*
Имплементация метода решения уравнения четвертой степени – FQS method
Информация о методе –
https://www.academia.edu/es/27122162/The\_fast\_quartic\_solver
Работу выполнил – Погосов Даниэль (https://github.com/DarklileS)
*/
template<typename fp_t>
unsigned int fqs(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>& roots) {

    // Нормировка коэффициентов
    if (isZero(n) || isinf(a /= n))
        return solveCubic(a, b, c, d, roots);
    if (isinf(b /= n))
        return 0;
    if (isinf(c /= n))
        return 0;
    if (isinf(d /= n))
        return 0;

    vector<complex<fp_t>> roots1(2);
    vector<complex<fp_t>> roots2(2);

    unsigned numberofRoots = 0;

    // Объявление констант
    static const fp_t ONE_HALF = static_cast<fp_t>(0.5L);
    static const fp_t ONE_6TH = static_cast<fp_t>(1.0L / 6.0L);

    //Case 1. Два двойных корня / Один четвертичный корень(4 - кратный корень).

    // Определяем коэффициенты вспомогательного уравнения
    fp_t alpha_ = a * ONE_HALF;
    fp_t beta_ = fma(-alpha_, alpha_, b) * ONE_HALF; // (b - alpha_^2) / 2

    // Определяющие коэффициенты
}

```

```

        fp_t E1 = fma(static_cast<fp_t>(-2.0L) * alpha_, beta_, c); // c - 2 * alpha_ *
beta
        fp_t E2 = fma(-beta_, beta_, d); // d - beta_^2

        if (isZero(E1) and isZero(E2)) {
            // Поиск корней вспомогательного уравнения
            solveQuadratic<fp_t>(static_cast<fp_t>(1.0L), alpha_, beta_, roots1);

            // Определим действительные корни
            if (isZero(roots1[0].imag())) {
                roots[numberOfRoots] = roots1[0].real();
                roots[numberOfRoots + 1] = roots1[0].real();
                numberOfRoots += 2;
            }
            if (isZero(roots1[1].imag())) {
                roots[numberOfRoots] = roots1[1].real();
                roots[numberOfRoots + 1] = roots1[1].real();
                numberOfRoots += 2;
            }
        }

        return numberOfRoots;
    }

    // Case 2. Один тройной корень и один простой корень.

    // Определяем коэффициенты вспомогательного уравнения
    alpha_ = a * ONE_HALF; beta_ = b * ONE_6TH;

    // Поиск корней вспомогательного уравнения
    vector<fp_t> roots_(2);
    solveQuadratic<fp_t>(static_cast<fp_t>(1.0L), alpha_, beta_, roots_);

    // Определяем корни исходного уравнения
    fp_t x1 = roots_[0], x2 = -fma(static_cast<fp_t>(3.0L), x1, a); // x2 = -3 * x1
- a
    // Определяющие коэффициенты
    E1 = fma(x1 * x1, fma(static_cast<fp_t>(3.0L), x2, x1), c); // c + x1 * x1 *
(x1 + 3 * x2);
    E2 = fma(-x1 * x1, x1 * x2, d); // d - x^3 * x2

    if (isZero(E1) and isZero(E2)) {
        roots = { x1, x2, x2, x2 }; return numberOfRoots += 4;
    }

    // Определяем корни исходного уравнения
    x1 = roots_[1]; x2 = -fma(static_cast<fp_t>(3.0L), x1, a); // -3 * x1 - a
    // Определяющие коэффициенты
    E1 = fma(x1 * x1, fma(static_cast<fp_t>(3.0L), x2, x1), c); // c + x1 * x1 *
(x1 + 3 * x2);
    E2 = fma(-x1 * x1, x1 * x2, d); // d - x1^3 * x2

```

```

    if (isZero(E1) and isZero(E2)) {
        roots = { x1, x2, x2, x2 }; return numberOfRoots += 4;
    }

    //Case 3. Один двойной корень и два простых корня / 4 простых корня.
    vector<fp_t> coeffs(4);
    // Определяем коэффициенты вспомогательных уравнений с помощью итеративного
    метода
    FQScoeffs(n, a, b, c, d, coeffs);
    fp_t alpha = coeffs[0], beta = coeffs[1], gamma = coeffs[2], delta = coeffs[3];

    // Поиск корней вспомогательного уравнения
    solveQuadratic<fp_t>(static_cast<fp_t>(1.0L), alpha, beta, roots1);
    solveQuadratic<fp_t>(static_cast<fp_t>(1.0L), gamma, delta, roots2);

    //Определим действительные корни
    if (isZero(roots1[0].imag())) {
        roots[numberOfRoots] = roots1[0].real();    numberOfRoots++;
    }
    if (isZero(roots1[1].imag())) {
        roots[numberOfRoots] = roots1[1].real();    numberOfRoots++;
    }
    if (isZero(roots2[0].imag())) {
        roots[numberOfRoots] = roots2[0].real();    numberOfRoots++;
    }
    if (isZero(roots2[1].imag())) {
        roots[numberOfRoots] = roots2[1].real();    numberOfRoots++;
    }

    return numberOfRoots;
}

/*
Имплементация метода решения уравнения четвертой степени – Ungar Method
Информация о методе – https://core.ac.uk/download/pdf/82351384.pdf

Работу выполнил – Погосов Даниэль (https://github.com/DarklIeS)
*/
template<typename fp_t>
unsigned int ungar(fp_t n, fp_t a, fp_t b, fp_t c, fp_t d, vector<fp_t>& roots)
{
    unsigned int numberOfRoots = 0;

    // MARK: - Объявление констант

    static const fp_t EIGHT_THIRDS = static_cast<fp_t>(8.0L / 3.0L);
    static const fp_t ONE_FOURTH = static_cast<fp_t>(0.25L);

```

```

        static const complex<fp_t> COMPLEX_ONE_FOURTH = complex<fp_t>(ONE_FOURTH,
static_cast<fp_t>(0.0L));
        static const complex<fp_t> COMPLEX_FOUR_THIRDS =
complex<fp_t>(static_cast<fp_t>(4.0L / 3.0L), static_cast<fp_t>(0.0L));
        static const complex<fp_t> COMPLEX_HALF =
complex<fp_t>(static_cast<fp_t>(0.5L), static_cast<fp_t>(0.0L));

        static const complex<fp_t> q0 = complex<fp_t>(static_cast<fp_t>(1.0L),
static_cast<fp_t>(0.0L));
        static const complex<fp_t> q1 = complex<fp_t>(static_cast<fp_t>(-0.5L),
static_cast<fp_t>(0.5L) * sqrt(static_cast<fp_t>(3.0L)));
        static const complex<fp_t> q2 = complex<fp_t>(static_cast<fp_t>(-0.5L),
static_cast<fp_t>(-0.5L) * sqrt(static_cast<fp_t>(3.0L)));

// MARK: - Вычисляем расчетные коэффициенты P, Q, R, alpha0, betta0, gamma0

// P
fp_t P_helper = fms(static_cast<fp_t>(4.0L), -b, -a, a); // -4b + a^2
fp_t P = fms(static_cast<fp_t>(8.0L), c, -a, P_helper); // 8c + a*P_helper

// Q
fp_t Q_helper = fms(static_cast<fp_t>(4.0L), d, a, c); // 4d - ac
fp_t Q = fms(static_cast<fp_t>(3.0L), Q_helper, -b, b); // 3Q_helper + b^2

// R
fp_t R_helper0 = fms(static_cast<fp_t>(3.0L) * d, a, b, c); // 3ad-bc
fp_t R_helper1 = fms(R_helper0, a, c * c, -static_cast<fp_t>(3.0L)); // R_helper0*a + 3c^2
fp_t R_helper2 = fms(b, b, static_cast<fp_t>(36.0L), d); // b^2 - 36d
fp_t R = fms(static_cast<fp_t>(2.0L) * R_helper2, b, -R_helper1,
static_cast<fp_t>(9.0L)); // 2 * R_helper2 * b + 9 * R_helper1

fp_t alpha0 = fms(a, a, EIGHT_THIRDS, b); // a^2 - 8 * b / 3

fp_t betta_gamma_helper = fma(ONE_FOURTH, R * R, -pow(Q,
static_cast<fp_t>(3.0L))); // R^2 / 4 - Q^3

complex<fp_t> sqrt_betta_gamma_helper = (betta_gamma_helper > 0) ?
sqrt(betta_gamma_helper) : complex<fp_t>(0, sqrt(abs(betta_gamma_helper)));
complex<fp_t> betta_helper_1 = fmac(COMPLEX_HALF, complex<fp_t>(R, 0),
sqrt_betta_gamma_helper); // R/2 + sqrt_betta_gamma_helper
complex<fp_t> gamma_helper_1 = fmac(COMPLEX_HALF, complex<fp_t>(R, 0), -
sqrt_betta_gamma_helper); // R/2 - sqrt_betta_gamma_helper

// MARK: Находим все значения кубического корня (R + sqrt(R^2 - 4Q^3)) / 2. В
статье ошибка (см. файл с опечатками)
vector<complex<fp_t>> betta0CubeRoots = cubeRoot(betta_helper_1);
vector<complex<fp_t>> gamma0CubeRoots = cubeRoot(gamma_helper_1);

```

```

// MARK: - Согласно теореме 5.1 считаем количество комплексных и действительных
корней.
// 1) R*R - 4*pow(Q,3) > 0 --- > 2 действит. + 2 комплексных корня
// 2) R*R - 4*pow(Q,3) = 0 --- > 2 действит. + (2 комплексных <=> T >= 0 для
всех cuberoots(betta_helper_1))
// 3) R*R - 4*pow(Q,3) < 0 --- > 3.1) 4 действит. <=> T >= 0 для всех
cuberoots(betta_helper_1)
int cnt_negative_T = 0;
int cnt_pozitive_T = 0;

std::vector<fp_t> T;

for (int i = 0; i < 3; ++i)
{
    fp_t _3a1_8b = fms(static_cast<fp_t>(3.0L) * a, a, static_cast<fp_t>(8.0L),
b); // 3*a^2 - 8*b
    fp_t t = fma(static_cast<fp_t>(8.0L), real(betta0CubeRoots[i]), _3a1_8b);
// 8 * Re(betta0CubeRoots[i]) + _3a1_8b
    // MARK: Используем round т.к никогда не получаем чистый 0. (Проверка с eps
не проходит. При вычитании двух одинаковых чисел получали не 0, но при этом
результат больше eps)
    if (round(abs(static_cast<fp_t>(8.0L) * real(betta0CubeRoots[i]) +
_3a1_8b)) <= numeric_limits<fp_t>::epsilon()) { t = 0; }
    T.push_back(t);
    if (T[i] < 0) cnt_negative_T++;
    if (T[i] >= 0) cnt_pozitive_T++;
}

// MARK: Используем round т.к никогда не получаем чистый 0. (Проверка с eps не
проходит)

// Исправление
numberOfRoots = (round(abs(betta_gamma_helper)) <=
numeric_limits<fp_t>::epsilon() && cnt_pozitive_T == 3) ? 4 :
(betta_gamma_helper > 0) ? 2 : (cnt_pozitive_T == 3) ? 4 : 0;

// Проверка условия 4.10
int beta0_I = 0;
int gamma0_J = 0;
check4_10Condition(betta0CubeRoots, gamma0CubeRoots, Q, beta0_I, gamma0_J);

complex<fp_t> alpha0_complex = complex<fp_t>(alpha0, static_cast<fp_t>(0.0L));
// +-sqrt(4/3*(q0*betta0CubeRoots[beta0_I] +
q0*gamma0CubeRoots[gamma0_J])+alpha0_complex)
vector<complex<fp_t>> bettaSqrts = { sqrt(fmac(COMPLEX_FOUR_THIRDS, fmsc(q0,
betta0CubeRoots[beta0_I], -q0, gamma0CubeRoots[gamma0_J]), alpha0_complex)),
-sqrt(fmac(COMPLEX_FOUR_THIRDS, fmsc(q0,
betta0CubeRoots[beta0_I], -q0, gamma0CubeRoots[gamma0_J]), alpha0_complex)) };
// +-sqrt(4/3*(q1*betta0CubeRoots[beta0_I] +
q2*gamma0CubeRoots[gamma0_J])+alpha0_complex)

```

```

    vector<complex<fp_t>> gammaSqrts = { sqrt(fmac(COMPLEX_FOUR_THIRDS, fmsc(q1,
        betta0CubeRoots[beta0_I], -q2, gamma0CubeRoots[gamma0_J]), alpha0_complex)),
        -sqrt(fmac(COMPLEX_FOUR_THIRDS, fmsc(q1,
        betta0CubeRoots[beta0_I], -q2, gamma0CubeRoots[gamma0_J]), alpha0_complex)) };
    // +-sqrt(4/3*(q2*betta0CubeRoots[beta0_I] +
    q1*gamma0CubeRoots[gamma0_J])+alpha0_complex)
    vector<complex<fp_t>> deltaSqrts = { sqrt(fmac(COMPLEX_FOUR_THIRDS, fmsc(q2,
        betta0CubeRoots[beta0_I], -q1, gamma0CubeRoots[gamma0_J]), alpha0_complex)),
        -sqrt(fmac(COMPLEX_FOUR_THIRDS, fmsc(q2,
        betta0CubeRoots[beta0_I], -q1, gamma0CubeRoots[gamma0_J]), alpha0_complex)) };
    // Проверка условия 4.11
    int bettaI = 0;
    int gammaJ = 0;
    int deltaK = 0;
    check4_11Condition(bettaSqrts, gammaSqrts, deltaSqrts, P, bettaI, gammaJ,
    deltaK);

    // MARK: - Находим корни

    fp_t alpha = -ONE_FOURTH * a;
    complex<fp_t> alpha_complex = complex<fp_t>(alpha, static_cast<fp_t>(0.0L));
    // (deltaSqrts[deltaK]/4 + (gammaSqrts[gammaJ]/4+(bettaSqrts[bettaI]/4 +
    alpha_complex)))
    complex<fp_t> w0 = fmac(COMPLEX_ONE_FOURTH, deltaSqrts[deltaK],
        fmac(COMPLEX_ONE_FOURTH, gammaSqrts[gammaJ], fmac(COMPLEX_ONE_FOURTH,
        bettaSqrts[bettaI], alpha_complex)));
    // (-deltaSqrts[deltaK]/4 + (-gammaSqrts[gammaJ]/4+(bettaSqrts[bettaI]/4 +
    alpha_complex)))
    complex<fp_t> w1 = fmac(-COMPLEX_ONE_FOURTH, deltaSqrts[deltaK], fmac(-
        COMPLEX_ONE_FOURTH, gammaSqrts[gammaJ], fmac(COMPLEX_ONE_FOURTH,
        bettaSqrts[bettaI], alpha_complex)));
    // (-deltaSqrts[deltaK]/4 + (gammaSqrts[gammaJ]/4+(-bettaSqrts[bettaI]/4 +
    alpha_complex)))
    complex<fp_t> w2 = fmac(-COMPLEX_ONE_FOURTH, deltaSqrts[deltaK],
        fmac(COMPLEX_ONE_FOURTH, gammaSqrts[gammaJ], fmac(-COMPLEX_ONE_FOURTH,
        bettaSqrts[bettaI], alpha_complex)));
    // (deltaSqrts[deltaK]/4 + (-gammaSqrts[gammaJ]/4+(-bettaSqrts[bettaI]/4 +
    alpha_complex)))
    complex<fp_t> w3 = fmac(COMPLEX_ONE_FOURTH, deltaSqrts[deltaK], fmac(-
        COMPLEX_ONE_FOURTH, gammaSqrts[gammaJ], fmac(-COMPLEX_ONE_FOURTH,
        bettaSqrts[bettaI], alpha_complex)));

    if (numberOfRoots == 4)
    {
        roots[0] = w0.real();
        roots[1] = w1.real();
        roots[2] = w2.real();
        roots[3] = w3.real();
    }
    else if (numberOfRoots == 2) {

```

```

        if (!isComplex(w0)) roots[0] = w0.real();
        if (!isComplex(w1)) roots[1] = w1.real();
        if (!isComplex(w2)) roots[2] = w2.real();
        if (!isComplex(w3)) roots[3] = w3.real();
    }
    return numberOfRoots;
}

/*
    Метод проведения оценки точность имплементированных методов
*/
template<typename fp_t>
void testQuarticPolynomial(int testCount, long double maxDistance, int P1, int P2)
{
    unsigned P = 4; // Степень исходного полинома
    fp_t low = -1, high = 1; // Интервал на котором заданы корни полинома
    fp_t absMaxError, relMaxError; // Абсолютная и относительная погрешность по
итогам пройденного теста
    fp_t absMaxErrorTotal = -1, relMaxErrorTotal = -1; // Итоговая максимальная
абсолютная и относительная погрешность по итогам всех тестов
    long double absErrorAvg = 0, relErrorAvg = 0; // Средняя абсолютная и
относительная погрешность по итогам всех тестов
    unsigned numberOfFoundRoots; // Количество найденных корней
    unsigned cantFind = 0; // Счетчик количества ситуаций, когда методу не удалось
найти корни (numberOfFoundRoots == 0)
    vector<fp_t> coefficients(P + 1); // Вектор коэффициентов полинома
    unsigned count = 0; // Счетчик количества ситуаций, когда относительная
погрешность больше определенного числа (relMaxError > n)
    int countExcessRoots = 0;
    int countLostRoots = 0;
    vector<fp_t> errors;
    for (size_t i = 0; i < testCount; ++i)
    {
        vector<fp_t> foundRoots(P);
        vector<fp_t> trueRoots(P);
        int excessRoots = 0;
        int lostRoots = 0;

        generate_polynomial<fp_t>(P, 0, P1, P2, static_cast<fp_t>(maxDistance),
low, high, trueRoots, coefficients);

        numberOfFoundRoots = ferrari<fp_t>(coefficients[4], coefficients[3],
coefficients[2], coefficients[1], coefficients[0], foundRoots);

        if (numberOfFoundRoots > 0)
        {
            compare_roots<fp_t>(numberOfFoundRoots, P, foundRoots, trueRoots,
absMaxError, relMaxError, excessRoots, lostRoots, errors);
        }
    }
}

```

```

        absMaxErrorTotal = absMaxError > absMaxErrorTotal ? absMaxError :
absMaxErrorTotal;
        absErrorAvg += absMaxError;

        relMaxErrorTotal = relMaxError > relMaxErrorTotal ? relMaxError :
relMaxErrorTotal;
        relErrorAvg += relMaxError;

        countExcessRoots += excessRoots;
        countLostRoots += lostRoots;

        count += relMaxError > 0.95 ? 1 : 0;
    }
    else
    {
        countLostRoots += 4;
        cantFind += 1;
    }
}

absErrorAvg /= (testCount - cantFind);
relErrorAvg /= (testCount - cantFind);
sort(errors.begin(), errors.end());
string label("");
long double quantile = 0;

if (P1 == 0 && P2 == 0)
{
    label = string("NON-CLUSTER ");
    quantile = 0.01;
}
else if (P1 == 0)
{
    label = string("MULTIPLE ");
    quantile = 0.1;
}
else if (P2 == 0)
{
    label = string("CLUSTER ");
    quantile = 0.1;
}

fp_t quantileMaxErr = errors[errors.size() - errors.size() * quantile];

if (PRINT)
{
    cout << label << "QUARTIC TEST RESULTS" << endl;
    cout << "===== " << endl;
    cout << "Max distance: " << maxDistance << endl;
    cout << "Total count of tests: " << testCount << endl;
}

```

```

        cout << "Couldn't find roots: " << cantFind << " times " << endl;
        cout << "-----" << endl;
        cout << "Average absolute error: " << absErrorAvg << endl;
        cout << "Total maximum absolute error: " << absMaxErrorTotal << endl;
        cout << "Average relative error: " << relErrorAvg << endl;
        cout << "Total maximum relative error: " << relMaxErrorTotal << endl;
        cout << "-----" << endl;
        cout << "Total count of lost roots: " << countLostRoots << endl;
        cout << "Total count of excess roots: " << countExcessRoots << endl;
        cout << "-----" << endl;
        cout << "relMaxError > 0.95: " << count << " times" << endl;
        cout << "Quantile max error: " << quantileMaxErr << endl;
        cout << "===== " << endl;
    }
}

```

2. Исходный код имплементации алгоритмов генерации коэффициентов полинома и оценки абсолютной и относительной погрешности (файл excerpt.cpp)

```

// Creates a test polynomial, both in the form of roots, e.g. (x-roots[0])*(x-
// roots[1])*(quadratic polynomial with no real roots)
// and represented by its coefficients, e.g.
// (coefficients[4]=1)*x^4 + coefficients[3]*x^3 + coefficients[2]*x^2 +
coefficients[1]*x + coefficients[0].
// The highest-degree coefficient always equals 1. The function returns the actual
number of different real roots placed into the vector
// (roots) (complex roots are not placed there). Negative return values may mean
internal implementation error
template<typename fp_t>
int generate_polynomial(
    unsigned P, // polynomial degree
    unsigned N_pairs_of_complex_roots, // how many pairs of complex conjugate roots
to introduce
    unsigned N_clustered_roots, // how many clustered roots to introduce; all the
clustered roots are real
    unsigned N_multiple_roots, // how many multiple roots to introduce; all
multiple roots are real
    fp_t max_distance_between_clustered_roots, // maximal distance between the
closest of the clustered roots
    fp_t root_sweep_low,
    fp_t root_sweep_high, // low and high boundaries of real roots; imaginary parts
of complex conjugate roots are in the same range
    std::vector<fp_t>& roots, // storage where to put the roots; size should exceed
P-1
    std::vector<fp_t>& coefficients) // storage where to put the coefficients; size
should exceed P
{

```

```

    int n_simple_roots = P - 2 * N_pairs_of_complex_roots - N_clustered_roots -
N_multiple_roots;
    assert(N_clustered_roots != 1);
    assert(N_multiple_roots != 1);
    assert(n_simple_roots >= 0);
    assert(P > 0 && P <= 4);
    assert(max_distance_between_clustered_roots > static_cast<fp_t>(0.0L));
    assert(root_sweep_high - root_sweep_low > 2 * P *
max_distance_between_clustered_roots);

    unsigned long long seed =
        std::chrono::system_clock::now().time_since_epoch().count() + std::rand();
// counts milliseconds
    std::mt19937_64 rng(seed); // randomize seed from the clock
    std::uniform_real_distribution<fp_t> rnr(root_sweep_low,
        root_sweep_high); // uniform random data generator for single roots
    std::uniform_real_distribution<fp_t> rnc(static_cast<fp_t>(0.0L),
        max_distance_between_clustered_roots); // uniform random data generator for
root clusters
    fp_t re, im, u, v;
    auto root_mid_sweep = root_sweep_low + 0.5 * (root_sweep_high -
root_sweep_low);
    long double RE, IM, U, V, TMP; // high-precision counterparts of re, im, u, v

    coefficients[P] = static_cast<fp_t>(1.0L); // invariant
    switch (P) {
    case 0:
        coefficients[0] = rnr(rng);
        return 0;
    case 1:
        coefficients[0] = -(roots[0] = rnr(rng));
        return 1;
    case 2: {
        if (N_pairs_of_complex_roots == 1) // no real roots
        {
            re = rnr(rng);
            while ((im = rnr(rng)) == static_cast<fp_t>(0.0L)) {}
            RE = re;
            IM = im;
            coefficients[1] = static_cast<fp_t>(-2.0L * RE); // -2*re
            coefficients[0] = static_cast<fp_t>(pr_product_difference(RE, RE, -IM,
IM)); // re*re+im*im
            return 0;
        }
        else if (N_clustered_roots == 2) // 2 close but distinct roots
        {
            roots[0] = re = rnr(rng);
            while ((im = rnc(rng)) == static_cast<fp_t>(0.0L)) {}
            roots[1] = im = (re >= root_mid_sweep ? re - im : re + im);
        }
    }
}

```

```

        else if (N_multiple_roots == 2) // double root counted as a single root
    {
        roots[1] = roots[0] = im = re = rnr(rng);
    }
    else // 2 distinct single roots
    {
        roots[0] = re = rnr(rng);
        while ((im = rnr(rng)) == re) {}
        roots[1] = im = rnr(rng);
    }
    RE = re;
    IM = im;

    coefficients[1] = static_cast<fp_t>(-RE - IM);
    coefficients[0] = static_cast<fp_t>(RE * IM);
    return 2; // return ((re!=im) ? 2 : 1);
} // P=2
case 3: {
    if (N_pairs_of_complex_roots == 1) // one real root
    {
        re = rnr(rng);
        while ((im = rnr(rng)) == static_cast<fp_t>(0.0L)) {}
        roots[0] = u = rnr(rng);
        RE = re;
        IM = im;
        U = u;

        IM = pr_product_difference(RE, RE, -IM, IM); // re*re+im*im
        RE *= -2.0L; // irreducible quadratic polynomial is (x^2 + re*x + im);
        multiply it by (x-u)
        coefficients[0] = static_cast<fp_t>(-IM * U);
        coefficients[2] = static_cast<fp_t>(RE - U);
        coefficients[1] = static_cast<fp_t>(std::fma(-RE, U, IM)); // im-re*u;
        return 1;
    }
    else if (N_clustered_roots == 3) // 3 clustered distinct roots
    {
        roots[0] = re = rnr(rng);
        while ((im = rnc(rng)) == static_cast<fp_t>(0.0L)) {}
        while ((u = rnc(rng)) == static_cast<fp_t>(0.0L)) {}
        roots[1] = im = (re > root_mid_sweep ? roots[2] = u = (re - im - u), re
- im : roots[2] = u = (re + im +
            u), re +
            im);
    }
    else if (N_clustered_roots == 2) // 2 clustered roots, 1 single root; all
distinct
    {
        roots[0] = re = rnr(rng);
        while ((im = rnc(rng)) == static_cast<fp_t>(0.0L)) {}

```

```

        roots[1] = im = (re > root_mid_sweep ? re - im : re + im);
        do { roots[2] = u = rnr(rng); } while (u == re || u == roots[1]);
    }
    else if (N_multiple_roots == 3) // triple root counted as a single root
    {
        roots[2] = roots[1] = roots[0] = u = im = re = rnr(rng);
    }
    else if (N_multiple_roots == 2) // double root and 1 single root; totally 2
roots
    {
        roots[1] = roots[0] = im = re = rnr(rng);
        while ((roots[2] = u = rnr(rng)) == re) {}
    }
    else // 3 distinct single roots
    {
        roots[0] = re = rnr(rng);
        while ((roots[1] = im = rnr(rng)) == re) {}
        do { roots[2] = u = rnr(rng); } while (u == re || u == im);
    }
    RE = re;
    IM = im;
    U = u;
    coefficients[2] = static_cast<fp_t>(-RE - IM - U);
    coefficients[0] = static_cast<fp_t>(-RE * IM * U);
    V = pr_product_difference(RE, IM, -RE, U);
    coefficients[1] = static_cast<fp_t>(std::fma(IM, U, V)); // re*im+re*u+im*u=im*u+(re*im-(-re*u));
    // if (re!=im && im!=u && u!=re) return 3;
    // if (re==im && im==u) return 1;
    // return 2;
    return 3;
} // P=3
case 4: // DEN DEBUG: check it carefully and perform calculation of
coefficients in long double
{
    if (N_pairs_of_complex_roots == 2) // no real roots
    {
        re = rnr(rng);
        while (std::abs(im = rnr(rng)) < std::abs(re)) {}
        RE = re;
        IM = im;
        IM = pr_product_difference(RE, RE, -IM, IM); // RE*RE+IM*IM
        RE *= -2.0L; // irreducible quadratic polynomial is (x^2 + re*x + im)
        u = rnr(rng);
        while (std::abs(v = rnr(rng)) < std::abs(u)) {}
        U = u;
        V = v;
        V = pr_product_difference(U, U, -V, V); // U*U+V*V
        U *= -2.0L; // irreducible quadratic polynomial is (x^2 + u*x + v)
        // multiply both irreducible quadratics
    }
}

```

```

        coefficients[0] = static_cast<fp_t>(IM * V);
        coefficients[1] = static_cast<fp_t>(pr_product_difference(RE, V, -IM,
U)); // RE*V+IM*U;
        coefficients[2] = static_cast<fp_t>(std::fma(RE, U, IM + V)); //
IM+RE*U+V
        coefficients[3] = static_cast<fp_t>(RE + U);
        return 0;
    }
    else if (N_pairs_of_complex_roots == 1) // two real roots
    {
        re = rnr(rng);
        while (std::abs(im = rnr(rng)) < std::abs(re)) {}
        RE = re;
        IM = im;
        IM = pr_product_difference(RE, RE, -IM, IM); // RE*RE+IM*IM
        RE *= -2.0L; // irreducible quadratic polynomial is (x^2 + re*x + im);
multiply it by the rest
        // 2 real roots follow
        if (N_clustered_roots == 2) // 2 clustered roots
        {
            roots[0] = u = rnr(rng);
            v = rnc(rng);
            roots[1] = v = (u > root_mid_sweep ? u - v : u + v);
        }
        else if (N_multiple_roots == 2) // 2 multiple roots
        {
            roots[1] = roots[0] = u = v = rnr(rng);
        }
        else // 2 distinct roots
        {
            roots[0] = u = rnr(rng);
            roots[1] = v = rnr(rng);
        }
        U = u;
        V = v;
        TMP = -U - V;
        V *= U;
        U = TMP; // two-real-root quadratic polynomial is (x^2 + u*x + v)
// multiply irreducible and reducible quadratics
        coefficients[0] = static_cast<fp_t>(IM * V);
        coefficients[1] = static_cast<fp_t>(pr_product_difference(RE, V, -IM,
U)); // RE*V+IM*U
        coefficients[2] = static_cast<fp_t>(std::fma(RE, U, IM + V)); //
IM+RE*U+V
        coefficients[3] = static_cast<fp_t>(RE + U);
        return 2;
    }
    else if (N_clustered_roots == 4) // 4 clustered roots
    {
        roots[0] = re = rnr(rng);

```

```

        im = rnc(rng);
        u = rnc(rng);
        v = rnc(rng);
        roots[1] = im = (re > root_mid_sweep ? (roots[3] = v = (re - im - u -
v), roots[2] = u = (re - im - u),
re - im) :
(roots[3] = v = (re + im + u + v), roots[2] = u = (re + im + u), re
+ im));
    }
    else if (N_clustered_roots == 3) // 3 clustered roots and 1 single root
    {
        roots[0] = re = rnr(rng);
        im = rnc(rng);
        u = rnc(rng);
        roots[1] = im = (re > root_mid_sweep ? (roots[2] = u = (re - im - u),
re - im) : (roots[2] = u = (re +
im +
u),
re + im));
        roots[3] = v = rnr(rng); // a single root
    }
    else if (N_clustered_roots == 2) // 2 clustered roots
    {
        roots[0] = re = rnr(rng);
        im = rnc(rng);
        roots[1] = im = (re > root_mid_sweep ? re - im : re + im);
        if (N_multiple_roots == 2) // 2 multiple roots
        {
            roots[3] = roots[2] = v = u = rnr(rng);
        }
        else // 2 single roots
        {
            roots[2] = u = rnr(rng);
            roots[3] = v = rnr(rng);
        }
    }
    else if (N_multiple_roots == 4) // 4 multiple roots
    {
        roots[3] = roots[2] = roots[1] = roots[0] = v = u = im = re = rnr(rng);
    }
    else if (N_multiple_roots == 3) // 3 multiple roots and 1 single root
    {
        roots[2] = roots[1] = roots[0] = u = im = re = rnr(rng);
        roots[3] = v = rnr(rng);
    }
    else if (N_multiple_roots == 2) // 2 multiple roots and 2 single roots
    {
        roots[1] = roots[0] = im = re = rnr(rng);
        roots[2] = u = rnr(rng);
        roots[3] = v = rnr(rng);
    }
}

```

```

    }
    else // 4 distinct single roots
    {
        roots[0] = re = rnr(rng);
        roots[1] = im = rnr(rng);
        roots[2] = u = rnr(rng);
        roots[3] = v = rnr(rng);
    }
    // compute coefficients from 4 roots: re, im, u, v
    RE = re;
    IM = im;
    U = u;
    V = v;
    TMP = -RE - IM;
    IM *= RE;
    RE = TMP; // now we have the 1.st quadratic polynomial: x^2 + x*re + im
    TMP = -U - V;
    V *= U;
    U = TMP; // now we have the 2.nd quadratic polynomial: x^2 + x*u + v
    coefficients[0] = static_cast<fp_t>(IM * V);
    coefficients[1] = static_cast<fp_t>(pr_product_difference(RE, V, -IM, U));
// RE*V+IM*U
    coefficients[2] = static_cast<fp_t>(std::fma(RE, U, IM + V)); // IM+RE*U+V
    coefficients[3] = static_cast<fp_t>(RE + U);
    return 4;
} // P=4
default:
    return -1;
} // switch (P)
return -1; // unreachable, means a flaw in control here
}

// Compares two vectors of roots; root orderings play no role. For each entry in
// (roots_ground_truth),
// the closest entry in (roots_to_check) is found and corresponding distance found.
// Among such distances
// the largest will be stored to (max_deviation)
template<typename fp_t>
int compare_roots(
    unsigned N_roots_to_check, // number of roots in (roots_to_check)
    unsigned N_roots_ground_truth, // number of roots in (roots_ground_truth)
    std::vector<fp_t>& roots_to_check, // one should take into account only first
    (N_roots_to_check) roots here
    std::vector<fp_t>& roots_ground_truth, // one should take into account only
    first (N_roots_ground_truth) roots here
    fp_t& max_absolute_error, // here the greatest among the smallest deviations of
    the roots in (roots_to_check) and (roots_ground_truth)
    // will be placed

```

```

    fp_t& max_relative_error, // here the greatest relative error among all the
    roots found will be placed
    int& excessRoots, // number of excess roots
    int& lostRoots, // number of lost roots
    std::vector<fp_t>& errors) // vector of all max relative errors
{
    max_absolute_error = static_cast<fp_t>(0);
    max_relative_error = static_cast<fp_t>(0);

    for (int i = 0; i < N_roots_to_check; i++) {
        if (std::isnan(roots_to_check[i]))
            return PR_AT_LEAST_ONE_ROOT_IS_NAN;
        //Since we can't compare return errors as zero for better comparing
        compatibility
    }

    int compareNumOfRoots = N_roots_ground_truth - N_roots_to_check;

    if (compareNumOfRoots > 0)
    {
        excessRoots = 0;
        lostRoots = compareNumOfRoots;
    }
    else if (compareNumOfRoots < 0)
    {
        excessRoots = -compareNumOfRoots;
        lostRoots = 0;
    }
    else
    {
        excessRoots = 0;
        lostRoots = 0;
    }

    fp_t deviation, absolute_error_max = static_cast<fp_t>(0.0L),
relative_error_max = static_cast<fp_t>(0.0L);
    fp_t deviation1 = static_cast<fp_t>(0.0L);
    fp_t deviation2 = static_cast<fp_t>(0.0L);
    auto rv = (N_roots_to_check < N_roots_ground_truth) ? PR_AT_LEAST_ONE_ROOT_LOST
    :
        ((N_roots_to_check > N_roots_ground_truth) ? PR_AT_LEAST_ONE_ROOT_IS_FAKE : PR_NUMBERS_OF_ROOTS_EQUAL);
    // find the largest distance between the closest pairs of roots: one - from
    ground truth, one - from found ones

    if (N_roots_to_check <= 0) {
        throw std::out_of_range("N_roots_to_check should be greater than zero");
    }

    for (int i = 0; i < N_roots_to_check; ++i) {

```

```

// find the closest found root to the given ground truth root
fp_t deviation_min_for_this_root = std::numeric_limits<fp_t>::infinity();
auto i_closest_root = -1, j_closest_root = -1;
//std::cout<<"rootsToCheck:<<N_roots_to_check;
for (int j = 0; j < N_roots_ground_truth; ++j) {

    deviation = std::abs(roots_to_check[i] - roots_ground_truth[j]);
    //deviation2 = std::abs(-roots_to_check[i] - roots_ground_truth[j]);
    //deviation = deviation1 < deviation2 ? deviation1 : deviation2;
    //roots_to_check[i] = deviation1 < deviation2 ? roots_to_check[i] : -
roots_to_check[i];
    //std::cout<<"dev:"<<deviation;
    deviation_min_for_this_root =
        deviation < deviation_min_for_this_root ? i_closest_root = i,
j_closest_root = j, deviation
        : deviation_min_for_this_root;
}

if (i_closest_root == -1 or j_closest_root == -1) {
    throw std::out_of_range("closest root not found");
}
// assert(i_closest_root != -1 and j_closest_root != -1);
//auto relative_error_for_this_root = static_cast<fp_t>(2.0L) *
deviation_min_for_this_root /
    //                                         (std::abs(roots_ground_truth[
i_closest_root]) +
    //                                         std::abs(roots_to_check[j_cl
osest_root]));
    auto relative_error_for_this_root = (deviation_min_for_this_root +
std::numeric_limits<fp_t>::epsilon()) /
        (std::max(std::abs(roots_ground_truth[j_closest_root]),
            std::abs(roots_to_check[i_closest_root])) +
std::numeric_limits<fp_t>::epsilon());

    absolute_error_max =
        deviation_min_for_this_root > absolute_error_max ?
deviation_min_for_this_root : absolute_error_max;
    relative_error_max =
        relative_error_for_this_root > relative_error_max ?
relative_error_for_this_root : relative_error_max;

    if (relative_error_for_this_root > 3.99)
    {
        std::cout << relative_error_for_this_root << std::endl <<
            deviation_min_for_this_root + std::numeric_limits<fp_t>::epsilon()
<< std::endl <<
            (std::max(std::abs(roots_ground_truth[j_closest_root]),
std::abs(roots_to_check[i_closest_root])) + std::numeric_limits<fp_t>::epsilon())
<< std::endl <<

```

```

        roots_ground_truth[j_closest_root] << " " <<
roots_to_check[i_closest_root] << std::endl <<
    (std::max(std::abs(roots_ground_truth[j_closest_root]),
std::abs(roots_to_check[i_closest_root])) + std::numeric_limits<fp_t>::epsilon()) -
(deviation_min_for_this_root + std::numeric_limits<fp_t>::epsilon()) << std::endl
<< std::endl;
    }

}

errors.push_back(relative_error_max);
max_absolute_error = absolute_error_max;
max_relative_error = relative_error_max;

return rv;
}

```

3. Исходный код имплементации алгоритмов аналитической оценки погрешностей (файл framework.ipynb)

```

# Метод упрощения выражения (epsilon порядка > 1 -> 0)
def simplify_eps(expr):
    expr = expand(expr)

    if not isinstance(expr, Expr):
        return expr

    # Проверяем содержит ли выражение перемножение epsilon-содержащих элементов
    if isinstance(expr, Mul):
        args = []
        for arg in expr.args:
            # Если аргумент - степень
            if arg.is_Pow:
                # Разбиваем его на основание и показатель степени
                base, exp = arg.as_base_exp()
                # Если основание содержит подстроку 'epsilon_' и показатель степени
                больше 1
                if str(base).startswith('epsilon_') and exp > 1:
                    # Заменяем аргумент на 0
                    arg = 0
                else:
                    # Рекурсивно упрощаем основание и показатель степени
                    arg = simplify_eps(base)**simplify_eps(exp)
            else:
                # Рекурсивно упрощаем аргумент
                arg = simplify_eps(arg)

            args.append(arg)

        new_args = []

```

```

        for i, arg1 in enumerate(args):
            for j in range(i + 1, len(args)):
                arg2 = args[j]
                # Если оба аргумента содержат подстроку 'epsilon_' (epsilon_1 *
epsilon_2), то обращаем в 0
                if str(arg1).startswith('epsilon_') and
str(arg2).startswith('epsilon_'):
                    # Добавляем 0 в новый список аргументов и прекращаем дальнейшую
проверку
                    new_args.append(0)
                    break
                else:
                    new_args.append(arg1)
        return Mul(*new_args)

    # Проверяем содержит ли выражение возведение в степень epsilon-содержащего
элемента
    if isinstance(expr, Pow):
        # Разбиваем выражение на основание и показатель степени
        base, exp = expr.as_base_exp()
        # Если основание содержит подстроку 'epsilon_' и показатель степени больше 1
        if str(base).startswith('epsilon_') and exp > 1:
            # Обращаем слагаемое в 0
            return 0
        else:
            # Рекурсивно упрощаем основание
            base = simplify_eps(base)
            # Рекурсивно упрощаем показатель степени
            exp = simplify_eps(exp)

        return Pow(base, exp)

    if isinstance(expr, Add):
        args = []
        for arg in expr.args:
            # Рекурсивно упрощаем аргумент
            arg = simplify_eps(arg)
            args.append(arg)
        return Add(*args)

    return expr

# Метод определения наихудшего случая
def abs_coeff_eps(expr):
    # Группируем коэффициенты относительно каждого epsilon-содержащего элемента
    expr = expr.expand()
    expr = expr.collect(epsilons)

    # Убираем все epsilon (epsilon_temp = 0)
    new_expr = expr.subs({s: 0 for s in epsilons})

```

```

# Создаем словарь всех epsilon-содержащих элементов
eps_syms = [symbol for symbol in epsilons if str(symbol) in [str(s) for s in expr.free_symbols]]

for eps_sym in eps_syms:
    eps_sym = sp.Symbol(str(eps_sym))
    # Получаем коэффициент при eps_sym
    coeff = expr.coeff(eps_sym)
    # Задаем модуль полученного коэффициента
    abs_coeff = Abs(coeff)
    # Добавляем перемножение eps_sym и модуля коэффициента, как слагаемое
    # нового выражения
    new_expr += abs_coeff*eps_sym

# Обработка случаев когда eps_syms = sqrt(epsilon_temp)
for i in range(len(eps_syms)):
    eps_syms[i] = sp.Symbol(str(eps_syms[i]))**0.5

for eps_sym in eps_syms:
    # Получаем коэффициент при eps_sym
    coeff = expr.coeff(eps_sym)
    # Задаем модуль полученного коэффициента
    abs_coeff = Abs(coeff)
    # Добавляем перемножение eps_sym и модуля коэффициента, как слагаемое
    # нового выражения
    new_expr += abs_coeff*eps_sym

# Замена всех epsilon-содержащих элементов единой константой epsilon
new_expr = new_expr.subs({s: epsilon for s in epsilons})

return new_expr

# Метод поиска наихудшей относительной погрешности каждого полинома
def find_max_error(roots, trueRoots, coeffs, N):
    rel_err_arr = [] # Массив относительной погрешности
    new_coeffs = [] # Массив отсеянных коэффициентов
    n_found = 0 # Количество найденных корней
    n_true = 4 # Количество истинных корней
    lossRoots = 0 # Количество потерянных корней

    for i in range(N):
        max_rel_err = -1 # Максимальная относительная ошибка
        n_found = 0 # Обновляем счетчик количества найденных корней

        for j in range(4):
            # Если удалось определить корень, увеличиваем счетчик найденных корней
            if(roots[i][j] != np.inf):
                n_found += 1
            # Иначе, увеличиваем счетчик потерянных корней

```

```

        else:
            lossRoots += 1

        for j in range(n_found):
            j_close = -1 # Индекс ближайшего найденного корня
            k_close = -1 # Индекс ближайшего истинного корня
            deviation_min_for_this_root = np.inf # Минимальная абсолютная
погрешность для данного корня

            for k in range(n_true):
                # Вычисляем абсолютную погрешность
                deviation = abs(roots[i][j]-trueRoots[i][k])

                # Если найденная величина минимальная, но обновляем индексы и мин.
абс. погрешность
                if deviation < deviation_min_for_this_root:
                    j_close = j
                    k_close = k
                    deviation_min_for_this_root = deviation

            # Не удалось определить индексы - выводим ошибку
            if j_close == -1 or k_close == -1:
                print("ERROR")

            # Определяем истинный корень
            tr = sp.Float(trueRoots[i][k_close], acc)
            # Определяем найденый корень
            r = sp.Float(roots[i][j_close], acc)

            # Вычисляем абсолютную погрешность
            abs_err = abs(tr - r)
            # Вычисляем относительную погрешность
            rel_err = abs_err / (abs(tr) + EPS_FLOAT)
            # Определяем наибольшую величину относительной погрешности
            max_rel_err = rel_err if max_rel_err < rel_err else max_rel_err

# Обрабатываем случаи, когда работаем с величиной близкой к 0
powerToZero = 0.375 * acc
closeToZero = mpmath.mpf(10**(-powerToZero))

# Если не удалось найти корени, то выводим ошибку
if closeToZero > max_rel_err or n_found == 0:
    print(COEFFS[i])
    print(i)
else:
    # Добавляем коэффициенты новый список
    new_coeffs.append(COEFFS[i])
    # Вычисляем логарифмированную величину максимальной относительной
погрешности
    rel_err_arr.append(mpmath.log(max_rel_err))

```

```

print(lossRoots)

return rel_err_arr, new_coeffs

# Метод вывода графика зависимостей двух коэффициентов и логарифмированной величины
# относительной погрешности
def plot_3d_surface(x, y, error, xLabel, yLabel, rad, angx, angy, path, file_name):
    x = np.array(x)
    y = np.array(y)
    error = np.array(error)
    print("Aza")
    # Определение окрестности
    radius = rad

    # Создание новых массивов для точек с максимальным значением error в
    окрестности
    x_new = []
    y_new = []
    error_new = []

    # Вычисление евклидовой матрицы расстояний между точками
    dist_matrix = cdist(np.column_stack((x, y)), np.column_stack((x, y)))

    for i in range(len(x)):
        x_i = x[i]
        y_i = y[i]
        error_i = error[i]

        # Поиск индексов точек в окрестности
        indices = np.where(dist_matrix[i] <= radius)[0]

        # Поиск максимального значения error в окрестности
        max_error = np.max(error[indices])

        # Добавление точки с максимальным значением error в новые массивы
        if error_i == max_error:
            x_new.append(x_i)
            y_new.append(y_i)
            error_new.append(max_error)

    x_new = np.asarray(x_new, dtype=np.float64)
    y_new = np.asarray(y_new, dtype=np.float64)
    error_new = np.asarray(error_new, dtype=np.float64)

    # Создание экземпляра Rbf
    interp = Rbf(x_new, y_new, error_new, function='linear', smooth = 0)

    # Создание сетки точек для интерполяции
    xi = np.linspace(min(x_new), max(x_new), 1000)

```

```

yi = np.linspace(min(y_new), max(y_new), 1000)

xi, yi = np.meshgrid(xi, yi)

# Интерполяция высоты поверхности с использованием Rbf
zi = interp(xi, yi)
print(len(zi))
# Создание цветовой сетки
colors = ['#02aab0', '#00ca10', '#c6ca00', '#ca6c00', '#ca0000']
cmap = mcolors.LinearSegmentedColormap.from_list('custom_cmap', colors)
# Создание трехмерного графика
fig = plt.figure(dpi=180)
ax = fig.add_subplot(111, projection='3d')
ax.view_init(elev=angy, azim=angx)

ax.plot_surface(xi, yi, zi, cmap=cmap, alpha = 1, linewidth=0.1, edgecolors =
(0, 0, 0, 0.5))

ax.tick_params(axis='x', labelsize=16)
ax.tick_params(axis='y', labelsize=16)
ax.tick_params(axis='z', labelsize=16)

# Настройка осей
ax.set_xlabel(xLabel, fontsize=22)
ax.set_ylabel(yLabel, fontsize=22)
ax.set_zlabel('$\ln(error)$', fontsize=22)

current_directory = os.getcwd()

# Создание пути для сохранения файла
save_folder = path
os.makedirs(os.path.join(current_directory, save_folder), exist_ok=True)

fig.set_size_inches(5.25, 5.25)
save_file_path = os.path.join(current_directory, save_folder, file_name)

fig.savefig(save_file_path)

plt.show()

# Выкладки для метода Феррари
def Ferrari(A3, A2, A1, A0, delta = sp.Symbol('delta'), m = sp.Symbol('m')):
    C = mul(A3, ONE_QUARTER)
    b2 = sub(A2, mul(mul(SIX, C), C))
    b1 = add(sub(A1, mul(mul(TWO, A2), C)), mul(mul(mul(EIGHT, C), C), C))
    b0 = sub(add(sub(A0, mul(A1, C)), mul(mul(A2, C), C)), mul(mul(mul(mul(THREE,
C), C), C), C))

    m = m * (1 + delta)
    if 'epsilon' in str(delta):

```

```

        coeff, exp = delta.as_base_exp()
    if exp < 1:
        epsilons.append(coeff)
    else:
        delta = simplify_eps(delta)
        if(delta != 0):
            epsilons.append(delta)

display(m)

sigma = sign(b1)
if(b1 == 0):
    sigma = -ONE

display(mul(b2, m))
display(abs_coeff_eps(mul(b2, m)).subs({s: sp.Symbol('epsilon') for s in
epsilons}))

R = mul(sigma, Sqrt(sub(add(add(mul(m, m), mul(b2, m)), mul(ONE_QUARTER,
mul(b2, b2))), b0)))

mHalf = mul(m, ONE_HALF)
sqrtMHalf = Sqrt(mHalf)
b2Half = mul(b2, ONE_HALF)

radicandPart = sub(-mHalf, b2Half)
radical1 = Sqrt(sub(radicandPart, R))
radical2 = Sqrt(add(radicandPart, R))

rootPart1 = sub(sqrtMHalf, C)
rootPart2 = sub(-sqrtMHalf, C)

Z1 = add(rootPart1, radical1)
Z2 = sub(rootPart1, radical1)
Z3 = add(rootPart2, radical2)
Z4 = sub(rootPart2, radical2)

return Z1, Z2, Z3, Z4

# Выкладки для метода Декарта
def Descartes (A3, A2, A1, A0, delta = sp.Symbol('delta'), yy = sp.Symbol('y')**2):
    C = mul(A3, ONE_QUARTER)
    b2 = sub(A2, mul(mul(SIX, C), C))
    b1 = add(sub(A1, mul(mul(TWO, A2), C)), mul(mul(mul(EIGHT, C), C), C))
    b0 = sub(add(sub(A0, mul(A1, C)), mul(mul(A2, C), C)), mul(mul(mul(mul(THREE,
C), C), C), C))

    yy = yy * (1 + delta)
    if 'epsilon' in str(delta):
        coeff, exp = delta.as_base_exp()

```

```

    if exp < 1:
        epsilons.append(coeff)
    else:
        delta = simplify_eps(delta)
        if(delta != 0):
            epsilons.append(delta)

    sigma = sign(b1)
    if(b1 == 0):
        sigma = -1

    R = mul(sigma, Sqrt(sub(add(add(mul(mul(yy, yy), ONE_QUARTER), mul(mul(b2, ONE_HALF), yy)), mul(mul(b2, b2), ONE_QUARTER)), b0)))

    y = Sqrt(yy)

    yHalf = mul(y, ONE_HALF)
    b2Half = mul(b2, ONE_HALF)
    yyQuarter = mul(-yy, ONE_QUARTER)

    radicandPart = sub(yyQuarter, b2Half)
    radical1 = Sqrt(sub(radicandPart, R))
    radical2 = Sqrt(add(radicandPart, R))

    rootPart1 = sub(yHalf, C)
    rootPart2 = -add(yHalf, C)

    Z1 = add(rootPart1, radical1)
    Z2 = sub(rootPart1, radical1)
    Z3 = add(rootPart2, radical2)
    Z4 = sub(rootPart2, radical2)

    return Z1, Z2, Z3, Z4
# Выкладки для метода NBS
def NBS(A3, A2, A1, A0, delta = sp.Symbol('delta'), u = sp.Symbol('u')):
    u = u * (1 + delta)
    if 'epsilon' in str(delta):
        coeff, exp = delta.as_base_exp()
        if exp < 1:
            epsilons.append(coeff)
        else:
            delta = simplify_eps(delta)
            if(delta != 0):
                epsilons.append(delta)

    sigma = sign(A1 - A3 * u * ONE_HALF)
    if(A1 - A3 * u * ONE_HALF == 0):
        sigma = -1

    A3Half = mul(A3, ONE_HALF)

```

```

AA3Quarter = mul(mul(A3, A3), ONE_QUARTER)
radicalP = Sqrt(sub(add(AA3Quarter, u), A2))

uHalf = mul(u, ONE_HALF)
uuQuarter = mul(mul(u, u), ONE_QUARTER)
radicalQ = mul(sigma, Sqrt(sub(uuQuarter, A0)))

p1 = sub(A3Half, radicalP)
p2 = add(A3Half, radicalP)
q1 = add(uHalf, radicalQ)
q2 = sub(uHalf, radicalQ)

p1Half = mul(p1, -ONE_HALF)
pp1Quarter = mul(mul(p1, p1), ONE_QUARTER)
p2Half = mul(p2, -ONE_HALF)
pp2Quarter = mul(mul(p2, p2), ONE_QUARTER)

radical1 = Sqrt(sub(pp1Quarter, q1))
radical2 = Sqrt(sub(pp2Quarter, q2))

Z1 = add(p1Half, radical1)
Z2 = sub(p1Half, radical1)
Z3 = add(p2Half, radical2)
Z4 = sub(p2Half, radical2)

return Z1, Z2, Z3, Z4

# Выкладки для метода Эйлера
def Euler (A3, A2, A1, A0, delta):
    C = mul(A3, ONE_QUARTER)
    b2 = sub(A2, mul(mul(SIX, C), C))
    b1 = add(sub(A1, mul(mul(TWO, A2), C)), mul(mul(mul(EIGHT, C), C), C))
    b0 = sub(add(sub(A0, mul(A1, C)), mul(mul(A2, C), C)), mul(mul(mul(mul(THREE,
C), C), C), C))

    for i in range(4):
        if 'epsilon' in str(delta[i]):
            coeff, exp = delta[i].as_base_exp()
        if exp < 1:
            epsilons.append(coeff)
        else:
            delta[i] = simplify_eps(delta[i])
            if(delta[i] != 0):
                epsilons.append(delta[i])

    r1 = sp.Symbol('r_1') * (1 + delta[0])
    x2 = sp.Symbol('x_2') * (1 + delta[1])
    x3 = sp.Symbol('x_3') * (1 + delta[2])
    y2 = sp.Symbol('y_2') * (1 + delta[3])

```

```

sigma = sign(b1)
if(b1 == 0):
    sigma = -1

subradical = mul(TWO, mul(sigma, Sqrt(add(mul(x2, x3), mul(y2, y2)))))
sumX = add(x2, x3)
sqrtR1 = Sqrt(r1)

radical1 = Sqrt(sub(sumX, subradical))
radical2 = Sqrt(add(sumX, subradical))

Z1 = sub(add(sqrtR1, radical1), C)
Z2 = sub(sub(sqrtR1, radical1), C)
Z3 = sub(add(-sqrtR1, radical2), C)
Z4 = sub(sub(-sqrtR1, radical2), C)

return Z1, Z2, Z3, Z4

# Выкладки для метода Ван дер Вардена
def VanDerWaerden(A3, A2, A1, A0, delta):
    C = mul(A3, ONE_QUARTER)
    b2 = sub(A2, mul(mul(SIX, C), C))
    b1 = add(sub(A1, mul(mul(TWO, A2), C)), mul(mul(mul(EIGHT, C), C), C))
    b0 = sub(add(sub(A0, mul(A1, C)), mul(mul(A2, C), C)), mul(mul(mul(mul(THREE,
C), C), C), C))

    for i in range(4):
        if 'epsilon' in str(delta[i]):
            coeff, exp = delta[i].as_base_exp()
        if exp < 1:
            epsilons.append(coeff)
        else:
            delta[i] = simplify_eps(delta[i])
            if(delta[i] != 0):
                epsilons.append(delta[i])

    theta1 = sp.Symbol('theta_1') * (1 + delta[0])
    x2 = sp.Symbol('theta_x_2') * (1 + delta[1])
    x3 = sp.Symbol('theta_x_3') * (1 + delta[2])
    y2 = sp.Symbol('theta_y_2') * (1 + delta[3])

    sigma = sign(b1)
    if(b1 == 0):
        sigma = -1

    subradical = mul(TWO, mul(sigma, Sqrt(add(mul(x2, x3), mul(y2, y2)))))
    subX = sub(-x2, x3)
    sqrtTheta1 = Sqrt(-theta1)

    radical1 = Sqrt(sub(subX, subradical))

```

```
radical2 = Sqrt(add(subX, subradical))

Z1 = sub(mul(ONE_HALF, add(sqrtTheta1, radical1)), C)
Z2 = sub(mul(ONE_HALF, sub(sqrtTheta1, radical1)), C)
Z3 = sub(mul(ONE_HALF, add(-sqrtTheta1, radical2)), C)
Z4 = sub(mul(ONE_HALF, sub(-sqrtTheta1, radical2)), C)

return Z1, Z2, Z3, Z4
```