Содержание

1 Введение

В данной статье мы рассмотрим модифицированный метод Лагерра, который обладает важным достоинством в виде локальной сходимости четвертого порядка [1].

Во втором разделе мы определяем формулу модифицированного метода Лагерра для обновления приближений корня для общего случая и для случая простых корней; на основе процедуры Бини подбираем начальные приближения, которые на практике почти всегда сходятся к корням многочлена; определяем процедуру вычисления поправочного члена с учетом машинной точности; описываем численно устойчивый метод для решения квадратного уравнения, возникающее в методе Лагерра; рассматриваем критерий остановки, гарантирующий, что итерации не прекратятся до тех пор, пока полученное значение не станет надежным для приближения корня.

В третьем разделе мы поэтапно описываем работу рассматриваемого алгоритма, приводим способ реализации с помощью псевдокода.

В четвертом разделе мы приводим способ имплементации данного алгоритма с помощью языка программирования C++, описываем реализованные функции и классы.

В пятом разделе мы демонстрируем результаты проведенных тестов для полиномов различных степеней.

2 Теоретическое обоснование

2.1 Вывод метода

Рассмотрим полином p от переменной z, определенный следующим образом:

$$p(z) = a_0 + a_1 z + \cdots + a^m z^m \tag{1}$$

где $a_0 a_m \neq 0$. Если $a_0 = 0$, полином делится на линейный коэффициет и метод применяется к полученному полиному. Обозначим $\{z_1, ..., z_m\}$ текущее приближение к корням $\{\zeta_1, ..., \zeta_m\}$ полинома p. Суть алгоритма заключается в обновлении каждого приближения z_j, пока оно не станет "достаточно близким" к корню ζ_j.

Метод Лагерра выполняет это обновление, решая квадратное уравнение, которое мы обозначим $Q_j(z) = 0$.Отметим, что если корни p вещественные, то оба решения $Q_j(z) = 0$ будут гарантированно ближе к ζ_j, чем текущее приближение z_j. Именно этим объясняется глобальная сходимость метода Лагерра, когда корни полинома вещественные.

Обозначим корни уравнения $Q_j(z)$ с помощью $\hat{z}_j$ и отметим, что:

$$\hat{z}_j = z_j - \frac{m}{G_j \pm \sqrt{(m-1)(mH_j - G_j^2)}} \tag{2}$$

где

$$G_j = \frac{p'(z_j)}{p(z_j)} = \sum_{i=1}^{m} \frac{1}{(z_i - \zeta_i)} \qquad H_j = -\left(\frac{p'(z_j)}{p(z_j)}\right)' = \sum_{i=1}^{m} \frac{1}{(z_i - \zeta_i)^2} \tag{3}$$

Обновление z_j определеятся ближайшим $\hat{z}_j$; то есть мы выбираем знак, который максимизирует знаменатель дроби в (2). Назовем это обновление *поправочным членом* z_j.

Для случая с простыми корнями, уравнение (3) записывается в виде невязки:

$$\hat{G}_j = \frac{p'(z_j)}{p(z_j)} - \sum_{\substack{i=1 \\ i \neq j}}^{m} \frac{1}{(z_j - z_i)} \qquad \hat{H}_j = -\left(\frac{p'(z_j)}{p(z_j)}\right)' - \sum_{\substack{i=1 \\ i \neq j}}^{m} \frac{1}{(z_j - z_i)^2} \tag{4}$$

Более того, это эквивалентно применению метода Лагерра к функции

$$f_j(z) = \frac{p(z)}{\prod_{\substack{i=1 \\ i \neq j}}^{m} (z_j - z_i)} \tag{5}$$

при вычислении поправочного члена z_j. Таким образом, мы создаем полюса во всех других корневых аппроксимациях и, следовательно, избегаем ненужной множественной сходимости к одному и тому же корню.

2.2 Начальная аппроксимация

Как и во всех итерационных методах, производительность модифицированного метода Лагерра существенно зависит от качества начальных приближений. Для их вычисления будет применяться процедура, предложенная Бини [2], которая выбирает комплексные числа вдоль окружностей подходящих радиусов, которые можно формализировать применив теорему Пелле [3].

Теорема 1. Пусть p - полином, определенный в (1). Для каждого k, такого что $a_k \neq 0$, рассмотрим уравнение:

$$|a_k|z^k = \sum_{\substack{i=0 \\ i\neq k}}^{m} |a_i|z^i \tag{6}$$

1. Если $k = 0$, существует одно положительное вещественное решение s_0, и p не имеет корней, которые по модулю меньше, чем s_0.

2. Если $0 < k < m$, то вещественных положительных решений либо нет, либо есть два вещественных положительных решения $t_k \leq s_k$. В последнем случае p не имеет корней в открытом кольце $A(t_k, s_k)$ и ровно k корней с модулями меньшими или равными t_k.

3. Если $k = m$, существует одно положительное решение t_m, и p не имеет корней с модулями больше, чем t_m.

Пусть $0 = k_1 < k_2 < ... < k_q = m$ - значения k, для которых в (6) существует позитивное(-ые) решение(-я), тогда пусть

$$s_0 = s_{k_1} \leq t_{k_2} \leq s_{k_2} \leq ... \leq t_{k_{q-1}} \leq t_{k_q} = t_m$$

- эти решения. Тогда, каждый полином в классе

$$\mathcal{P}(p) = \left\{ \sum_{i=0} b_i z^i : |b_i| = |a_i| \right\}$$

имеет $(k_{i+1} - k_i)$ нулей в замкнутом кольце $\bar{\mathcal{A}}(t_{k_i}, s_{k_i})$, для $i = 1, ..., q-1$, и ни одного нуля в открытом кольце $\mathcal{A}(t_{k_i}, s_{k_i})$, для $i = 1, ..., q$.

Хотя эти данные могут быть использованы для определения начальных приближений нашего итерационного метода, формирование положительных решений (6) требует решения нескольких полиномиальных уравнений. Поэтому мы ищем стратегию с наименьшей временной сложностью. Для этого определим:

$$\begin{aligned}
u_k &= \max_{i<k} \left| \frac{a_i}{a_k} \right|^{1/(k-i)}, k = 1, ..., m, \\
v_k &= \min_{\substack{i>k \\ a_i\neq 0}} \left| \frac{a_i}{a_k} \right|^{1/(k-i)}, k = 0, ..., m-1, \\
u_0 &= 1/(1 + \max_{i>0} \left| \frac{a_i}{a_0} \right|) \\
v_0 &= 1 + \max_{i<m} \left| \frac{a_i}{a_0} \right|.
\end{aligned} \tag{7}$$

Тогда, получаем, что положительные решения (6) удовлетворяют

$$u_{k_i} \leq t_{k_i} \leq s_{k_i} \leq v_{k_i}, i = 1, ..., q. \tag{8}$$

Рассмотрим множество $\mathcal{C} = \{(i, \log|a_i|), i = 0, 1, ..., m\}$ и обозначим

$$\gamma(\mathcal{C}) = \{(\hat{k}_i, \log|a_{\hat{k}_i}|), 0 = \hat{k}_1 < \hat{k}_2 < ... < \hat{k}_{\hat{q}} = m\}$$

верхняя огибающая выпуклой оболочки $\mathcal{C}$. Мы знаем, что вершины $\gamma(\mathcal{C})$ удовлетворяют условиям $u_l \leq v_l$ тогда и только тогда, когда $l \in \{\hat{k}_1, ..., \hat{k}_{\hat{q}}\}$. Следовательно, согласно (8), имеем $\{k_1, ..., k_q\} \subseteq \{\hat{k}_1, ..., \hat{k}_{\hat{q}}\}$, где содержащее множество может быть вычислено эффективно. Поскольку множество $\mathcal{C}$ упорядочено относительно первый координаты, целесообразно будет использовать алгоритм монотонный цепочки Эндрю для вычисления $\gamma(\mathcal{C})$, который в данном случае имеет временную сложность $O(m)$.

2.3 Алгоритм Эндрю

Алгоритм Эндрю - алгоритм построения выпуклой оболочки в двумерном пространстве, является модификацией алгоритма Грэхема.

Выпуклой оболочкой множества S называется наименьшее выпуклое множество (множество, в котором все точки отрезка, образуемого любыми двумя точками данного множества, также принадлежат данному множеству), содержащее S.

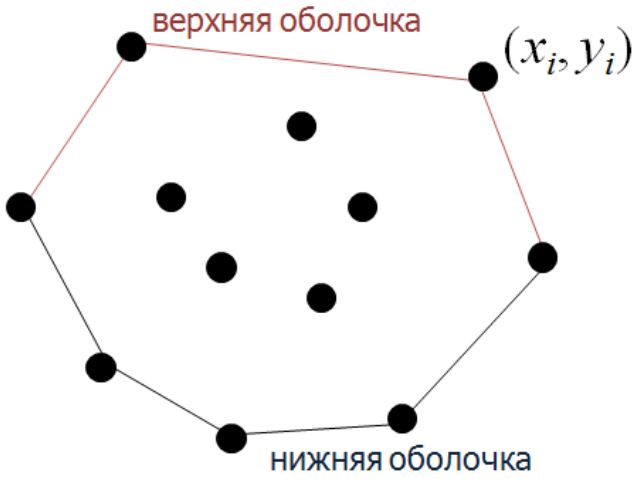

Рис. 1: Пример выпуклой оболочки для набора точек

Сначала алгоритм сортирует набор точек S по возрастанию координаты x, а затем y. Пусть минимальные и максимальные координаты x будут $x_{\min}$ и $x_{\max}$. Очевидно что у первой из точек $x = x_{\min}$, а у последней $x = x_{\max}$. Соединим эти две точки отрезком. Остальное множество точек разделяется на два, в зависимости от того, с какой стороны от этой прямой точки лежат. Таким образом, мы можем определить новую нижнюю и новую верхнюю линии. В совокупности эти линии и дают требуемую оболочку.

Для построения верхней оболочки точки множества S упорядочивается в соответствии с возрастанием абсциссы и после совершается работа полученных данных по алгоритму Грэхема (см. раздел №3). Для этого алгоритм Эндрю использует стек $x_0, x_1, \ldots, x_t$ для хранения текущей верхней оболочки. Точка x_t считается находящейся на вершине стека. После окончания работы алгоритма стек содержит верхнюю оболочку множества S.

2.4 Вычисление поправочного члена

В данном разделе мы рассматриваем численно устойчивый метод вычисления поправочного члена $\hat{z_j}$ (2), который используется для обновления приближения корня z_j.
Пусть $fl(p(\xi))$ представляет собой вычисленное значение функции p, оцененное в $\xi \in \mathbb{C}$ с использованием арифметики с плавающей точкой и машинной точностью μ. Тогда, используя метод Руффини-Хорнера получаем:

$$fl(p(\xi)) = \sum_{i=0}^{m} a_i(1+\epsilon_i)\xi^i \tag{9}$$

, где $|\epsilon_i| < ((2\sqrt{2}+1)i+1)\mu + O(\mu^2)$ Аналогичный результат имеет место быть для первой и второй производной
Метод Руффини-Хорнера склонен к переполнению при больших степенях полинома с положительными коэффициентами в точке ξ, где $|\xi| > 1$. По этой причине мы вводим обратный полином, определяемый как:

$$p_R(z) = a_0 z^m + ... + a_{m-1}z + a_m \tag{10}$$

Если $z \neq 0$, то $\rho = \frac{1}{z}$. Получаем соотношения:

$$\begin{aligned} \frac{p'(z)}{p(z)} &= \rho\left(m - \rho\frac{p'_R(\rho)}{p_R(\rho)}\right), \\ -\left(\frac{p'(z)}{p(z)}\right)' &= \rho^2\left(m - 2\rho\frac{p'_R(\rho)}{p_R(\rho)} - \rho^2\left(\frac{p'_R(\rho)}{p_R(\rho)}\right)'\right) \end{aligned} \tag{11}$$

Оценим временную сложность нашего алгоритма: начальные приближения в Алгоритме 1 вычисляются за время $O(m)$, затем каждое приближение обновляется за $O(m)$ следуя методу Руффини-Хорнера. Выходит, что Алгоритм 3 при фиксированном коэффициенте аппроксимирует все корни многочлена за $O(m^2)$

2.5 Критерий остановки

Пусть ξ - аппроксимация корня ζ, где все вычисления имеют машинную точность μ. В данном разделе мы вычислим обратную ошибку, которая измеряет то, насколько сильно нужно изменить коэффициенты многочлена, чтобы приближенный корень стал точным. Если обратная ошибка меньше или равна машинной точности, то алгоритм прекращает итерации и возвращает приближенный корень и его число обусловленности,где число обусловленности измеряет, насколько сильно изменение коэффициентов многочлена влияет на изменение корня. Если обратная ошибка больше машинной точности, то алгоритм продолжает итерации, пока не будет достигнут критерий остановки или не будет превышено максимальное число итераций.
Естественное определение обратной ошибки ξ выглядит следующим образом:

$$\eta(\xi) = min\{\epsilon : (p + \Delta p)(\xi) = 0, |\Delta a_i| \leq \epsilon |e_i|, i = 0, 1, ..., m\}, \tag{12}$$

где

$$\Delta p(z) = \Delta a_0 + \Delta a_1 z + ... + \Delta a_m z^m \tag{13}$$

Обратную ошибку можно определить как:

$$\eta(\xi) = \frac{p(\xi)}{\alpha(\xi)},\ \alpha(\xi) = \sum_{i=0}^{m} |e_i||\xi|^i \tag{14}$$

Исходя из анализа обратных ошибок правила Руффини-Хорнера (2.3), положим $e_i = ((2\sqrt{2} + 1)i + 1)a_i$. Тогда если $\eta(\xi) \leq \mu$, то ξ это корень $(p + \Delta p)$ и коэффициенты Δa_i не больше чем ϵ_i. Таким образом этот критерий остановки гарантирует, что итерации не прекратятся до тех пор, пока $fl(p(\xi)$ не перестанет быть надежным для приближения корня ξ.
Из секции вычисления поправочного члена, если $|\xi| > 1$, то мы работаем с обратным полиномом. Тогда обратная ошибка вычисляется как:

$$\eta(\xi) = \frac{|p_R(\rho)|}{\alpha_R(\rho)}, \alpha_R(\rho) = \sum_{i=0}^{m} |e_{m-i}||\rho|^i \tag{15}$$

Число обусловленности корня:

$$\kappa(\xi, \rho) = \frac{\alpha(\xi)}{|\xi||p'(\xi)|} \tag{16}$$

В итоге мы возвращаем значения κ и η, которые используются как критерий остановки и примерная чувствительность ζ к изменениям коэффициентов p. Произведение же $\eta(\xi)\kappa(\zeta, p)$ используется для получения ограничения первого порядка на ошибку прямого хода в приближении ξ.

3 Описание работы алгоритма

Пусть $p(z)$ - полином следующего вида:

$$p(z) = a_0 + a_1 z + \cdots + a_m z^m$$

Мы стремимися вычислить корни $p(z)$ и для этого необходимо выполнить следующие пункты:

1. Найти начальное приближение:

Algorithm 1: Вычисление начального приближения

Вычислить $\hat{k}_1, ..., \hat{k}_{\hat{q}}$ с помощью алгоритма монотонной цепи Эндрю
for $i = 1$ to $\hat{q} - 1$ **do**
    $n = \hat{k}_{i+1} - \hat{k}_i$
    $u_{k\hat{i+1}} = \left| \frac{a_{\hat{k}_i}}{a_{\hat{k}_{i+1}}} \right|^{1/n}$
    for $j = 1$ to n **do**
        $z_{\hat{k}_i + i} = u_{\hat{k}_{i+1}} e^{\left(\frac{2\pi}{n} j + \frac{2\pi}{m} + \sigma\right) i}$
    end for
end for

Для вычисления $\hat{k}_1, ..., \hat{k}_{\hat{q}}$ используется следующий алгоритм:

Algorithm 2: Алгоритм Эндрю для нахождения верхней оболочки

Функция определения направления поворота для трех точек:
function CROSS(Point p, Point q, Point r)
    val = (r.x - q.x) * (q.y - p.y) - (q.x - p.x) * (r.y - q.y)
    if val = 0 **then**
        return 0
    else if val > 0 **then**
        return 1
    else
        return 2
    end if
end function

$P \leftarrow$ список из m точек в двумерной плоскости
Отсортировать точки P по x-координате (в случае совпадения сортировка по y-координате)
for $i = m$ to 1 **do**
    p = P[i]
    L_upper $\leftarrow$ верхняя оболочка
    top возвращает верхнюю точку в стеке
    nextToTop возвращает вторую сверху точку в стеке
    while len(L_upper) >= 2 and cross(nextToTop(L_upper), top(L_upper), p) != 2 **do**
        L_upper.pop()
    end while
    L_upper.append(p)
end for

---

2. Обновлять начальное приближение, пока оно не станет "достаточно близко" к корням полинома, или пока не пройдет определенное число итераций.

Algorithm 3: Нахождение корней полинома

$(z_1, ..., z_m) \leftarrow$ Начальная аппроксимация полученная с помощью Алгоритма 1
while $i <$ itmax **do**
  for $j = 1$ to m **do**
    if z_j недостаточно близок к ζ_j **then**
      Вычислить корни $Q_j(\lambda)$ с помощью (2), используя (4)
      $z_j \leftarrow$ корень, который максимизирует знаменатель в (2)
    end if
  end for
  $i \leftarrow i + 1$
end while

4 Имплементация на C++

4.1 ExtendedFunctions.h

Описание:
Набор необходимых функций для реализации алгоритмов нахождения корней полиномов.
Функции:

- **anynotfinite (bool)** : проверка, является ли хотя бы одно число не конечным;
  Аргументы функции:
  - **T && ... t** : множество чисел (любое количество).

```
1 inline bool anynotfinite(T && ... t);
```

- **complexnotfinite (bool)** : проверка, содержит ли комплексное число значения NaN или Inf;
  Аргументы функции:
  - **a (complex<T>)** : комплексное число;
  - **big (T)** : максимальное значение для типа T.

```
1 bool complexnotfinite(complex<T> a, T big);
```

- **anycomplex (bool)** : проверка, что хотя бы одно комплесное число содержит мнимую часть;
  Аргументы функции:
  - **T && ... t** : множество чисел (любое количество).

```
1 inline bool anycomplex(T && ... t);
```

- **anycomplex (bool)** : проверка, что хотя бы одно комплесное число в векторе содержит мнимую часть;
  Аргументы функции:
  - **vec (vector<complex<T»)** : вектор комплексных чисел.

```
1 inline bool anycomplex(vector<complex<T>> vec);
```

- **sign (int)** : определение знака числа
  Аргументы функции:
  - **val (number)** : заданное значение;

  Возвращаемое значение: если заданное значение положительно, функция возвращает 1, если отрицательно, возвращает -1, если значение равно 0, возвращает 0.

```
1 inline int sign(number val);
```

- **fms (number)** : операция "fused multiply-subtract"(FMS), которая вычисляет разность произведения первых двух чисел и других двух чисел;
  Аргументы функции:

- **a (number)** : первое число;
- **b (number)** : второе число;
- **c (number)** : третье число;
- **d (number)** : четвертое число;

Возвращаемое значение: число $a \cdot b - d \cdot c$.

```
1 inline number fms(number a, number b, number c, number d);
```

- **fms (complex<number>)** : операция "fused multiply-subtract"(FMS) для комплесных чисел, которая вычисляет разность произведения первых двух чисел и других двух чисел;
  Аргументы функции:
  - **a (std::complex<number>)** : первое комплесное число;
  - **b (std::complex<number>)** : второе комплесное число;
  - **c (std::complex<number>)** : третье комплесное число;
  - **d (std::complex<number>)** : четвертое комплесное число;

  Возвращаемое значение: комплесное число $a \cdot b - d \cdot c$.

```
1 inline complex<number> fms(std::complex<number> a,
2                            std::complex<number> b,
3                            std::complex<number> c,
4                            std::complex<number> d);
```

- **fma (complex<number>)** : операция "fused multiply-add"(FMA) для комплесных чисел, которая вычисляет разность произведения двух чисел и третьего числа;
  Аргументы функции:
  - **a (std::complex<number>)** : первое комплесное число;
  - **b (std::complex<number>)** : второе комплесное число;
  - **c (std::complex<number>)** : третье комплесное число;

  Возвращаемое значение: комплесное число $a \cdot b - c$.

```
1 inline complex<number> fma(std::complex<number> a,
2                            std::complex<number> b,
3                            std::complex<number> c);
```

- **printVec (void)** : вывод вектора чисел в консоль;
  Аргументы функции:
  - **vec (vector<number>)** : вектор чисел;

```
1 inline void printVec(vector<number> vec);
```

- **castVec (vector<number>)** : преобразование вектора с типом T в вектор с типом number;
  Аргументы функции:
  - **vec (vector<T>)** : вектор чисел;

Возвращаемое значение: вектор чисел типа number.

```
1 inline vector<number> castVec(vector<T> vec);
```

4.2 Класс BaseSolver

Описание класса:

Абстрактный базовый класс для нахождения корней полиномов.

Методы класса:

- **operator() (void)** : нахождение корней полинома;

  Аргументы метода:

  - **coeff (std::vector<T>&)** : вектор, содержащий коэффициенты полинома;
  - **roots (std::vector<std::complex<T>&)** : вектор для хранения корней полинома;
  - **conv (std::vector<int>&)** : вектор для хранения статуса сходимости каждого корня;
  - **itmax (int)** : максимально допустимое количество итераций.

```
1 virtual void operator()(std::vector<T>& coeff,
2                         std::vector<std::complex<T>>& roots,
3                         std::vector<int>& conv,
4                         int itmax) = 0;
```

4.3 Класс ModifiedLaguerre18

Описание класса:

Класс, реализующий модифицированный алгоритм Лагерра для поиска корней полинома, основанный на статье [4].

Атрибуты класса:

- **eps (T)** : машинная точность для типа T;
- **big (T)** : максимальное значение для типа T;
- **small (T)** : минимальное положительное значение для типа T;
- **PI (T)** : значение числа Пи для типа T;
- **pi2 (T)** : удвоенное значение числа Пи для типа T.

Методы класса:

- **ModifiedLaguerre18()** : конструктор класса **ModifiedLaguerre18**;
- **operator() (void)** : нахождение корней полинома с использованием модифицированного метода Лагерра;
  Аргументы метода:
  - **poly (std::vector<T>&)** : вектор, содержащий коэффициенты полинома;
  - **roots (std::vector<std::complex<T>&)** : вектор для хранения корней полинома;

- **conv (std::vector<int>&)** : вектор для хранения статуса сходимости каждого корня;
- **itmax (int)** : максимально допустимое количество итераций.

```
1 void operator()(std::vector<T>& poly,
2                 std::vector<std::complex<T>>& roots,
3                 std::vector<int>& conv, int itmax);
```

- **cross (T)** : нахождение величины кросс-произведения (векторного произведения) двух векторов;
  Аргументы метода:
  - **h (std::vector<int>&))** : вектор целых чисел, представляющих точки;
  - **a (std::vector<T>&)** : вектор значений типа T, представляющих координаты точек;
  - **c (int)** : индекс текущей точки;
  - **i (int)** : Индекс следующей точки.

  Возвращаемое значение: величина кросс-произведения заданных векторов (T)

```
1 inline T cross(std::vector<int>& h,
2                std::vector<T>& a,
3                int c,
4                int i);
```

- **conv_hull (void)** : нахождение выпуклой оболочки заданного набора точек;
  Аргументы метода:
  - **n (int)** : количество точек;
  - **a (std::vector<T>&)** : вектор значений типа T, представляющих координаты точек;
  - **h (std::vector<int>&)** : вектор для хранения выпуклой оболочки;
  - **c (int&)** : количество точек в выпуклой оболочке.

```
1 inline void conv_hull(int n,
2                       std::vector<T>& a,
3                       std::vector<int>& h,
4                       int& c);
```

- **estimates (void)** : оценка корней полинома;
  Аргументы метода:
  - **alpha (vector<T>&)** : вектор модулей коэффициентов полинома;
  - **deg (int)** : степень полинома;
  - **conv (std::vector<int>&)** : вектор для хранения статуса сходимости каждого корня;
  - **nz (int&)** : количество нулей.

```
inline void estimates(vector<T>& alpha,
                      int deg,
                      std::vector<std::complex<T>>& roots,
                      std::vector<int>& conv,
                      int& nz);
```

- **rcheck_lag (void)** : коррекция метода Лагерра, вычисляет обратную ошибку и проверяет условие сходимости, если модуль корня больше 1;
  Аргументы метода:
  - **p (std::vector<T>&))** : вектор коэффициентов полинома;
  - **alpha (vector<T>&)** : вектор модулей коэффициентов полинома;
  - **deg (int)** : степень полинома;
  - **b (complex<T>&)** : выходное комплексное число, представляющее корень;
  - **c (complex<T>&)** : выходное комплексное число, представляющее коррекцию;
  - **z (complex<T>)** : комплексное число, используемое для коррекции;
  - **r (T)** : значение, используемое для коррекции;
  - **conv (int&)** : выходное значение, указывающее статус сходимости;
  - **berr (T&)** : выходная обратная ошибка;
  - **cond (T&)** : выходное число условия.

```
inline void rcheck_lag(std::vector<T>& p,
                       vector<T>& alpha,
                       int deg, complex<T>& b,
                       complex<T>& c,
                       complex<T> z,
                       T r,
                       int& conv,
                       T& berr,
                       T& cond);
```

- **check_lag (void)** : коррекция метода Лагерра, вычисляет обратную ошибку и проверяет условие сходимости, если модуль корня меньше 1;
  Аргументы метода:
  - **p (std::vector<T>&))** : вектор коэффициентов полинома;
  - **alpha (vector<T>&)** : вектор модулей коэффициентов полинома;
  - **deg (int)** : степень полинома;
  - **b (complex<T>&)** : выходное комплексное число, представляющее корень;
  - **c (complex<T>&)** : выходное комплексное число, представляющее коррекцию;
  - **z (complex<T>)** : комплексное число, используемое для коррекции;
  - **r (T)** : значение, используемое для коррекции;
  - **conv (int&)** : выходное значение, указывающее статус сходимости;
  - **berr (T&)** : выходная обратная ошибка;

– **cond (T&)** : выходное число условия.

```
inline void check_lag(std::vector<T>& p,
                      vector<T>& alpha,
                      int deg, complex<T>& b,
                      complex<T>& c,
                      complex<T> z,
                      T r,
                      int& conv,
                      T& berr,
                      T& cond);
```

- **modify_lag (void)** : модифицированная коррекция метода Лагерра на основе метода Аберта;
  Аргументы метода:
  - **deg (int)** : степень полинома;
  - **b (complex<T>&)** : выходное комплексное число, представляющее корень;
  - **c (complex<T>&)** : выходное комплексное число, представляющее коррекцию;
  - **z (complex<T>)** : комплексное число, используемое для коррекции;
  - **j (int)** : индекс корня для модификации;
  - **roots (std::vector<std::complex<T»&)** : вектор оцененных корней.

```
void modify_lag(int deg,
            std::complex<T>& b,
            std::complex<T>& c,
            std::complex<T> z,
            int j,
            std::vector<std::complex<T>>& roots);
```

5 Тестирование

Во время тестирования для каждой степени полинома случайным образом генерировалось 10000 экспериментальных полиномов. Для данных типа float рассматривались только полиномы 5-ой степени и ниже, так как при более высоких степенях происходила потеря точности коэффициентов и, как следствие, нахождение неверных корней.

5.1 Результаты тестов для обычных корней (float)

Степень полинома	**Худшая абсолютная погрешность**	**Худшая относительная погрешность**
3	0.000259757	0.000380564
4	0.00363469	0.0047001
5	0.0111473	0.0133276

5.2 Результаты тестов для обычных корней (double)

Степень полинома	**Худшая абсолютная погрешность**	**Худшая относительная погрешность**	**Диапазон корней**
5	0.0000004997	0.0000067689	[-1, 1]
10	0.0000005910	0.0001649274	[-1, 1]
20	0.0008829461	0.0000007432	[-1, 1]
50	0.01132795287	0.0756564119	[-1, 1]
100	0.0528136577	0.0538916389	[-1, 1]
200	0.0930998252	0.0534754074	[-1, 1]
500	3.5390033426	0.5439278577	[-10, 10]
1000	67.4228850912	0.804489235	[-100, 100]
2000	78.1241004113	0.85875	[-100, 100]
5000	38.5979819615	0.876262012	[-200, 200]
10000	59.8129892316	1.2473010413	[-200, 200]

5.3 Результаты тестов для кластеризованных корней (float)

Степень полинома	**Худшая абсолютная погрешность**	**Худшая относительная погрешность**	**Диапазон корней**	**Минимальная разница между корнями**
3	0.00162196	0.0311852	[-1, 1]	1e-5
4	0.0155236	0.0176819	[-1, 1]	1e-5
5	0.00782436	0.0113724	[-1, 1]	1e-5

5.4 Результаты тестов для кластеризованных корней (double)

Степень полинома	**Худшая абсолютная погрешность**	**Худшая относительная погрешность**	**Диапазон корней**	**Максимальная разница между корнями**
5	0.0003858524	0.0008009150	[-1, 1]	1e-5
10	0.002589944	0.0064761389	[-1, 1]	1e-5
20	0.0015857997	0.007432	[-1, 1]	1e-5
50	0.2114560494	0.1142833161	[-1, 1]	1e-5
100	0.3114560494	0.6281456165	[-2, 2]	1e-5
200	5.5913219644	0.1824598485	[-50, 50]	0.1
500	127.5988266699	0.9970928925	[-100, 100]	0.1
1000	404.5289142849	2.804489235	[-500, 500]	0.1
2000	430.6675218712	3.012456	[-500, 500]	0.1
5000	455.5979819615	3.20685	[-500, 500]	0.1
10000	489.8129892316]	3.45474	[-500, 500]	0.1

6 Список литературы

1. Petkovic, M.S., Ilic, S., Trickovic, S.: A family of simultaneous zero finding methods. Comput. Math. Appl. 34(10), 49–59 (1997)

2. Bini, D.A.: Numerical computation of polynomial zeros by means of Aberth's method. Numer. Algor. 13, 179–200 (1996)

3. Pellet, A.E.: Sur un mode de separation des racines des equations et la formule de lagrange. Bull. Sci. Math. 5(2), 393–395 (1881)

4. Thomas R. Cameron : An effective implementation of a modified Laguerre method for the roots of a polynomial. Numer. Algor. 82, 1065-1084 (2018)