**Преобразование Чирнгауза для решения** **полиномиального уравнения   
третьей степени**

Преобразование Чирнгауза – преобразование, переводящее многочлен с корнями в многочлен с корнями , где – также многочлен. Коэффициенты могут быть выражены через коэффициенты и , что может быть использовано для решения уравнений третьей и четвёртой степеней и упрощения общего вида уравнений более высоких степеней.

Преобразование Чирнгауза – метод для нахождения корней канонической формы кубического уравнения над полем комплексных чисел.

Любое уравнение общего вида при помощи замены переменной

может быть приведено к указанной выше канонической форме с коэффициентами

**Решение кубических уравнений с помощью преобразования Чирнгауза**

Пусть дано кубическое уравнение в упрощённом (приведённом) виде:

Тогда преобразование Чирнгауза переводит его в квадратное уравнение:

Осуществляя различные подстановки, Чирнгауз пришёл к выражению, зависящему от и и необходимому для вычисления коэффициента *a*:

Если положить, что коэффициент перед *y* равен нулю, получаем квадратное уравнение относительно *a*. Далее [**любой из корней**](#корни) *a*1,2 приводит к равенству

Отсюда находятся три значения для *y*. Чирнгауз подставил найденные значения в и нашёл решения относительно *x*. Однако он не указал, что, вообще говоря, получается ровно 6 значений для *x*. Путём исключения *y* из и было получено уравнение, которое объясняет появление шести корней для *x*:

После нахождения всех шести корней необходимо проверить на соответствие исходному уравнению каждый корень и отбросить посторонние.

**Описание кода программы**

Программа полностью адаптирована под многочлены, содержащие комплексные коэффициенты. В целях повышения точности вычислений при тестировании преимущественно используется тип данных *double* (хотя резкое падение точности [вычислений] при использовании *float* не наблюдалось). В начале представлена реализация вспомогательных функций для упрощения дальнейшей работы с основной функцией – *Tschirnhaus\_transformation*. Все функции описаны в шаблонах.

**Функция для нахождения всех значений кубического корня**

По ходу программы не раз приходится извлекать кубический корень из некоторого числа, причём необходимо получить все три значения. Для этого описана специальная функция, вычисляющая аргумент и модуль комплексного числа и переводящая это число в тригонометрическую форму, которая позволяет найти все значения кубического корня. Функция принимает комплексное число и возвращает вектор комплексных чисел.

template<typename T>

vector<complex<T>> cubic\_roots(complex<T> z)

{

complex<double> one\_third = { 1.0 / 3.0, 0 };

complex<double> phi = arg(z); //аргумент комплексного числа

complex<double> module = abs(z); //модуль комплексного числа

complex<double> cubic\_root\_module = pow(module, one\_third); //кубический корень из модуля комплексного числа

vector<complex<T>> roots;

for (int n = 0; n <= 2; n++) //вычисление значений корня

{

double p = 2 \* M\_PI \* n;

double alfa = real((phi + complex<double>{(p), 0})\* one\_third);

complex<double> value = cubic\_root\_module \* complex<double>{ cos(alfa), sin(alfa) };

roots.push\_back(value);

}

return roots;

}

**Листинг 1.** Нахождение значений кубического корня

В цикле переменная *p* отвечает за аргумент косинуса и синуса (без деления на 3), *alfa* отвечает за этот же аргумент, разделенный на 3. Переменная *value* – каждое из значений корня – записывается в вектор *roots*.

**Функция для нахождения корней квадратного уравнения**

Не менее часто возникает потребность находить корни квадратного уравнения   
 (старший коэффициент *a* = 1). Описанная ниже функция получает на вход вектор комплексных чисел (два значения), который представляет коэффициенты *b* и *с*, и возвращает вектор комплексных чисел – два корня уравнения. Корни отыскиваются через дискриминант.

template<typename T>

vector<complex<T>> roots\_of\_square\_poly(vector<complex<T>> s)

{

complex<double> one\_second = { 0.5, 0 };

complex<double> D = s[0] \* s[0] - complex<double>{ 4.0, 0 } \* s[1]; //дискриминант

complex<double> rootD = sqrt(D); //корень из дискриминанта

complex<T> x\_1 = (-s[0] + rootD) \* one\_second;

complex<T> x\_2 = (-s[0] - rootD) \* one\_second;

return { x\_1, x\_2 };

}

**Листинг 2.** Нахождение корней квадратного уравнения

**Функция для приведения общего вида многочлена 3-ей степени к упрощённому виду**

Так как преобразование Чирнгауза предполагает работу с трёхчленом , необходимо привести исходный многочлен к упрощённому виду. На этом этапе важно понимать, что после приведения уравнение решается именно **относительно новой переменной *t***, полученной путём некоторой замены, описанной ниже.

Функция принимает вектор комплексных чисел, представляющий коэффициенты многочлена 3-ей степени, и возвращает вектор комплексных чисел, представляющий новые коэффициенты.

template<typename T>

vector<complex<T>> canonical\_reduction(vector<complex<T>> P)

{

complex<double> one\_third = { 1.0 / 3.0, 0 };

complex<double> one\_twentyseventh = { 1.0 / 27.0, 0 };

complex<double> d = P[2], c = P[1], b = P[0]; //коэффициенты

complex<T> p = (complex<double>{ 3, 0 } \* c - b \* b)\* one\_third; //новые коэффициенты

complex<T> q = (complex<double>{27, 0} \* d - complex<double>{9, 0} \* b \* c + complex<double>{2, 0} \* b \* b \* b) \* one\_twentyseventh;

return { p,q };

}

**Листинг 3.** Приведение к упрощённому виду

Кубическое уравнение общего вида может быть приведено к каноническому виду путём деления на *a* (в нашем случае *a* = 1) и замены переменной . В результате получается упрощённый вид уравнения . Здесь

Функция вычисляет и возвращает значения для *p* и *q*.

**Основная функция – преобразование Чирнгауза**

После описания вспомогательных функций представляется возможным описание основной функции, осуществляющей преобразование Чирнгауза и поиск корней полиномиального уравнения 3-ей степени . Функция получает на вход вектор комплексных чисел – коэффициенты многочлена в исходном уравнении – и возвращает также вектор комплексных чисел, несущий три корня уравнения. В функции описаны в том числе частные случаи.

template<typename T>

vector<complex<T>> Tschirnhaus\_transformation(vector<complex<T>> F)

{

if ((F[1] == complex<double>{0, 0}) && (F[2] == complex<double>{0, 0})) //быстрый подсчёт корней при c = d = 0

{

return { complex<T>{0, 0},complex<T>{0, 0},-F[0] };

}

**Листинг 4.** Нулевое значение двух коэффициентов

В том случае, если *c* = *d* = 0, преобразование Чирнгауза становится неэффективным из-за низкой точности вычислений ввиду наличия кратного нулевого корня. Более целесообразным будет простое вынесение за скобку и нахождение третьего корня, равного.

Далее вводятся переменные, необходимые для оптимизации программы, например, *one\_third* и *one\_twentyseventh*. Здесь же происходит приведение уравнения к упрощённому виду с помощью функции *canonical\_reduction*. Новые значения записываются в переменные *p* и *q*.

После преобразования был получен трёхчлен . Однако, если *p* крайне мало, возникает проблема – деление на близкое к нулю значение. В таком случае можно не обрабатывать исключение, а просто найти корни = 0, применяя описанную в самом начале кода функцию.

complex<double> reverse\_p = complex<double>{ 1,0 } / p;

if ((isinf(real(reverse\_p)) || isnan(real(reverse\_p))) && (isinf(imag(reverse\_p)) || isnan(imag(reverse\_p)))) //проверка на ноль в знаменателе [если да - поиск корней для t^3 + q = 0]

{

//throw division\_by\_zero();

complex<double> y\_cubic\_0 = -q;

vector<complex<T>> x0 = cubic\_roots(y\_cubic\_0);

for (int l = 0; l <= 2; l++)

{

x0[l] = x0[l] - F[0] \* one\_third;

}

return x0;

}

**Листинг 5.** Случай при *p* → 0

Затем вычисляются значения и находятся коэффициенты *find\_a* для квадратного уравнения относительно *a* (потребность в таком вычислении описана в теоретической части). С помощью вспомогательной функции для поиска корней квадратного уравнения отыскиваются корни *a* (*a* – вектор комплексных чисел), а потом вычисляется коэффициент (не путать с из исходного уравнения), записанный в переменную *b*. В переменную *y\_cubic\_1* записывается значение куба коэффициента *y*, после чего вычисляются значения кубического корня, которые записываются в переменную *y1*.

Теперь нужно проверить, не обращаются ли все три значения *y* в ноль (это возможно при ). В случае, если куб *y* нулевой и, соответственно, все значения корня нулевые, получим . Находим корни *main\_roots0* последнего уравнения и добавляем один корень, равный (это проверяется делением «столбиком» полученного выше на ). Таким образом получим корни исходного полиномиального уравнения 3-ей степени.

vector<complex<double>> y1 = cubic\_roots(y\_cubic\_1); //проверка кубических корней из y1 на нулевое значение

if (y1 == vector<complex<double>>{0, 0, 0})

{

vector<complex<double>> sq\_poly\_coefs0 = { a[0], b };

vector<complex<T>> main\_roots0 = roots\_of\_square\_poly(sq\_poly\_coefs0);

main\_roots0.push\_back(a[0]);

return main\_roots0;

}

**Листинг 6.** Проверка коэффициента *y* на нулевое значение

В случае, если куб *y* не обращается в 0 и, соответственно, не все значения кубического корня *y1* нулевые, вычисляем шесть корней для исходного уравнения (по два на каждое значение *y1*), предварительно введя переменную *eps* = , которая нужна для дальнейшего «отбрасывания» посторонних корней путём проверки на сходимость к нулю.

* Стоит заметить, что такое значение для *eps* не всегда позволяет отобрать истинные корни, поэтому в дальнейшем будет проведено исследование взаимосвязи *eps* и коэффициентов исходного уравнения для выявления наиболее оптимального способа задания значения *eps*.

six\_roots.push\_back(main\_roots1[0] - F[0] \* one\_third);

six\_roots.push\_back(main\_roots1[1] - F[0] \* one\_third);

**Листинг 7.** Нахождение корней исходного уравнения с применением   
обратной замены

После вычисления шести корней вводим переменную *final\_roots*, в которую будут записаны истинные корни уравнения. В цикле проверяем корни и избавляемся от посторонних. В конце возвращаем вектор комплексных чисел – это и есть значение функции.

vector<complex<T>> final\_roots;

complex<double> d0 = F[2], c0 = F[1], b0 = F[0];

for (int j = 0; j <= 5; j++) //отбрасывание посторонних корней

{

complex<double> value = six\_roots[j];

complex<double> x\_square = value \* value;

complex<double> x\_cubic = value \* x\_square;

double re\_root = abs(real(x\_cubic + b0 \* x\_square + c0 \* value + d0));

double im\_root = abs(imag(x\_cubic + b0 \* x\_square + c0 \* value + d0));

if (re\_root <= eps && im\_root <= eps)

{

final\_roots.push\_back(value);

}

}

return final\_roots;

}

**Листинг 8.** Нахождение истинных корней

Вводим ещё одну функцию для вывода, применяя цикл. На этом код программы завершается.

**Заключение**

Следует доработать и оптимизировать метод подбора значения переменной *eps* для «отбрасывания» посторонних корней. Для этого необходимо выявить взаимосвязь переменной *eps* с коэффициентами исходного уравнения (в особенности – со свободным коэффициентом *d*)

Описанный код программы оптимизирован по времени. На этапе тестирования не было выявлено ошибок в вычислениях – функция *Tschirnhaus\_transformation* успешно завершила вычисление корней 140 уравнений (7 различных вариантов выбора ненулевых коэффициентов, содержащих по 20 тестов).

**Источники**

1. <https://www.researchgate.net/publication/268992597_Polynomial_transformations_of_Tschirnhaus_Bring_and_Jarrard>
2. <https://en.wikipedia.org/wiki/Cubic_equation>
3. <https://en.cppreference.com/w/>
4. E. Tschirnhaus, A method for removing all intermediate terms from a given equation, THIS BULLETIN, Vol 37(1), No. 143, (2003) 1–3. [Original: Methodus auferendi omnes terminos intermedios ex data equatione, ActaEruditorium,2(1683), 204-207]