Теоретическая часть

Алгоритм Дженкинса – Трауба для полиномиальных нулей — это быстрый глобально сходящийся итерационный метод. Были предложены два варианта: один для общих многочленов с комплексными коэффициентами, широко известный как алгоритм «CPOLY», и более сложный вариант для частного случая многочленов с действительными коэффициентами, широко известный как алгоритм «RPOLY».

Для многочлена P,

$$P(z) = \sum_{i=0}^{n} a_i z^{n-i}, \quad a_0 = 1, \ a_n \neq 0$$

с комплексными коэффициентами вычисляет приближения к n нулям $\alpha_1, \alpha_2, \ldots, \alpha_n$ от P (z), по одному с примерно возрастающим порядком величины. После вычисления каждого корня его линейный множитель удаляется из многочлена. Использование такой дефляции гарантирует, что каждый корень вычисляется только один раз и что все корни найдены.

Вариант с действительными коэффициентами следует тому же шаблону, но вычисляет два корня одновременно: либо два действительных корня, либо пару сопряженных комплексных корней. Реальный вариант может быть быстрее (в 4 раза), чем сложный вариант.

Алгоритм Дженкинса – Трауба вычисляет все корни многочлена с комплексными коэффициентами. Алгоритм начинается с проверки полинома на наличие очень больших или очень маленьких корней. При необходимости коэффициенты масштабируются путем изменения масштаба переменной. В алгоритме правильные корни находятся один за другим и, как правило, в увеличивающемся размере. После того, как каждый корень найден, полином дефлируется путем деления соответствующего линейного множителя.

1. Процедура поиска корня

Начиная с текущего многочлена P (X) степени n, нужно вычислить наименьший корень P (x). С этой целью строится последовательность так называемых H-полиномов. Все эти многочлены имеют степень n - 1 и должны сходиться к P (X), содержащему все оставшиеся корни. Последовательность полиномов H встречается в двух вариантах: ненормализованный вариант, который позволяет легко получить теоретическое понимание, и нормализованный вариант полиномов $\overline{H}$, который сохраняет коэффициенты в численно разумном диапазоне.

Построение многочленов H $\left(H^{(\lambda)}(z)\right)_{\lambda=0,1,2,\ldots}$ зависит от последовательности комплексных чисел $(s_\lambda)_{\lambda=0,1,2,\ldots}$, называемых сдвигами. Сами эти сдвиги зависят, по крайней мере, на третьем этапе, от предыдущих многочленов H. Многочлены H определяются как решение неявной рекурсии $H^{(0)}(z) = P'(z)$ и $(X - s_\lambda) \cdot H^{(\lambda+1)}(X) \equiv H^{\lambda}(X)$

Прямым решением этого неявного уравнения является

$$H^{(\lambda+1)}(X) = \frac{1}{X - s_\lambda} \cdot \left(H^{(\lambda)}(X) - \frac{H^{(\lambda)}(s_\lambda)}{P(s_\lambda)} P(X) \right)$$

где полиномиальное деление точное.

Алгоритмически можно использовать, например, схему Хорнера или правило Руффини для оценки многочленов в s_λ и одновременно получить частные. С полученными частными p (X) и h (X) в качестве промежуточных результатов следующий полином H получается как

$$\left.\begin{array}{r} P(X) = p(X)\cdot(X - s_\lambda) + P(s_\lambda) \\ H^\lambda(X) = h(X)\cdot(X - s_\lambda) + H^{(\lambda)}(s_\lambda) \end{array}\right\} \Longrightarrow H^{(\lambda+1)}(z) = h(z) - \frac{H^{(\lambda)}(s_\lambda)}{P(s_\lambda)}p(z)$$

Поскольку коэффициент наивысшей степени получается из P (X), старший коэффициент $H^{(\lambda+1)}(X)$ равен $-\frac{H^{(\lambda)}(s_\lambda)}{P(s_\lambda)}$. Если это разделить, нормализованный многочлен H равен

$$\overline{H}^{(\lambda+1)}(X) = \frac{1}{X - s_\lambda}\cdot\left(P(X) - \frac{P(s_\lambda)}{H^{(\lambda)}(s_\lambda)}H^{(\lambda)}(X)\right) = \frac{1}{X - s_\lambda}\cdot\left(P(X) - \frac{P(s_\lambda)}{\overline{H}^{(\lambda)}(s_\lambda)}\overline{H}^{(\lambda)}(X)\right).$$

Этап первый: процесс без сдвига

Для $\lambda = 0,1,\ldots,M-1$ установить $s_\lambda = 0$. Обычно M = 5 выбирается для многочленов умеренных степеней вплоть до n = 50. Этот этап не является необходимым только из теоретических соображений, но полезен на практике. Он подчеркивает в полиномах H кофактор (линейного множителя) наименьшего корня.

Этап второй: процесс фиксированного сдвига

Сдвиг для этого этапа определяется как некоторая точка, близкая к наименьшему корню многочлена. Он квазислучайно расположен на окружности с внутренним радиусом корня, который, в свою очередь, оценивается как положительное решение уравнения

$$R^n + |a_{n-1}|R^{n-1} + \cdots + |a_1|R = |a_0|.$$

Поскольку левая часть является выпуклой функцией и монотонно возрастает от нуля до бесконечности, это уравнение легко решить, например, с помощью метода Ньютона.

Теперь выберите $s = R\cdot\exp(i\phi_{random})$ на окружности этого радиуса. Последовательность многочленов $H^{(\lambda+1)}(z), \lambda = M, M+1, \ldots, L-1$, создается с фиксированным значением сдвига $s_\lambda = s$. Во время этой итерации текущее приближение для корня

$$t_\lambda = s - \frac{P(s)}{\overline{H}^{(\lambda)}(s)}$$

отслеживается. Второй этап завершается успешно, если условия

$|t_{\lambda+1} - t_\lambda| < \frac{1}{2}|t_\lambda|$ и $|t_\lambda - t_{\lambda-1}| < \frac{1}{2}|t_{\lambda-1}|$

одновременно выполняются. Если после некоторого количества итераций успеха не было, пробуется другая случайная точка на круге. Обычно используется 9 итераций для полиномов средней степени со стратегией удвоения в случае множественных отказов.

Этап третий: процесс переменного сдвига

$H^{(\lambda+1)}(X)$ теперь генерируются с использованием сдвигов переменных s_λ, $\lambda = L, L+1, \ldots$, которые генерируются

$$s_L = t_L = s - \frac{P(s)}{\overline{H}^{(\lambda)}(s)}$$

является последней оценкой корня второго этапа и

$$s_{\lambda+1} = s - \frac{P(s)}{\overline{H}^{(\lambda)}(s)}, \quad \lambda = L, L+1, \ldots,$$

где $\overline{H}^{(\lambda+1)}(z)$ - нормированный многочлен H, то есть $H^{(\lambda)}(z)$, деленный на его старший коэффициент.

Если размер шага на данном этапе недостаточно быстро падает до нуля, второй шаг перезапускается с использованием другой случайной точки. Если это не удается после небольшого количества перезапусков, количество шагов на втором этапе удваивается.

2. Сходимость

Можно показать, что при условии, что L выбрано достаточно большим, s_λ всегда сходится к корню P.

Алгоритм сходится для любого распределения корней, но может не найти все корни многочлена. Более того, сходимость немного быстрее, чем квадратичная сходимость итерации Ньютона – Рафсона, однако она использует как минимум вдвое больше операций на шаг.

3. Для работы с вещественными коэффициентами

Алгоритм Дженкинса – Трауба, описанный выше, работает для многочленов с комплексными коэффициентами. Те же авторы создали трехэтапный алгоритм для многочленов с действительными коэффициентами - трехэтапный алгоритм для вещественных многочленов с использованием квадратичной итерации. Алгоритм находит линейный или квадратичный фактор, полностью работающий в реальной арифметике. Если комплексный и реальный алгоритмы применяются к одному и тому же действительному многочлену, реальный алгоритм будет примерно в четыре раза быстрее. Реальный алгоритм всегда сходится, и скорость сходимости выше второго порядка.

Рассмотрим трехэтапный алгоритм Дженкинса-Трауба. На этапе 1 мы определяем

$$H^{(0)}(z) = P'(z); H^{(k+1)}(z) = \frac{1}{z}\left[H^{(k)}(z) - \frac{H^{(k)}(0)}{P(0)}P(z)\right](k = 0,1,2,\ldots,M$$

На втором этапе фактор $\frac{1}{z}$ заменяется на $\frac{1}{z-s}$ для фиксированного s, а на третьем этапе $\frac{1}{z-s_k}$, где s_k пересчитывается на каждой итерации. Затем s_k приближается к значению корня. Немного другой алгоритм дан для вещественных многочленов.

Мы вводим новый трехэтапный процесс вычисления нулей многочлена с вещественными коэффициентами. Алгоритм находит либо линейный, либо квадратичный множитель, полностью работая в реальной арифметике. На третьем этапе алгоритм использует одну из двух итераций с переменным сдвигом, соответствующих линейному или квадратичному случаю. Итерация для линейного множителя является действительной арифметической версией третьего этапа алгоритма для комплексных коэффициентов. Новая итерация со сдвигом переменной подходит для квадратичных множителей.

Результаты

Программа основана на трехэтапном алгоритме, описанном Дженкинсом и Траубом [2].

Итерации третьего этапа завершаются, когда выполняется критерий остановки, основанный на анализе ошибок округления.

Программа была протестирована на большом количестве полиномов, некоторые были выбраны для проверки слабых мест, общих для процедур поиска нуля, другие тесты проводились с использованием фреймворка Google Test [3]. Были проведены тесты на отдельно взятых частях алгоритма Дженкинса-Трауба. Тесты были написаны командой самостоятельно. Функция по вычислению корней выдает правильные корни: они совпадают с результатами вычислений в matlab (см. ниже).

```
18: Test command: /home/god/programing/jenkins_traub/tests/impl_1/build/tests_impl1 "--gtest_filter=Polynomial.HardPolynomial1" "--gtest_also_run_disabled_tests"
18: Working Directory: /home/god/programing/jenkins_traub/tests/impl_1/build
18: Test timeout computed to be: 10000000
18: Running main() from /home/god/programing/jenkins_traub/tests/impl_1/build/_deps/googletest-src/googletest/src/gtest_main.cc
18: Note: Google Test filter = Polynomial.HardPolynomial1
18: [==========] Running 1 test from 1 test suite.
18: [----------] Global test environment set-up.
18: [----------] 1 test from Polynomial
18: [ RUN      ] Polynomial.HardPolynomial1
18:
18:
18: roots_re:  1.14
18: -2.21
18: -0.32
18: -0.32
18: -1.66
18: -1.66
18:  3.33
18: -2.37
18: -3.84
18:  4.57
18: roots_im:     0
18:     0
18:  2.34
18: -2.34
18:  1.44
18: -1.44
18:     0
18:     0
18:     0
18:     0
18: [       OK ] Polynomial.HardPolynomial1 (1 ms)
18: [----------] 1 test from Polynomial (1 ms total)
18:
18: [----------] Global test environment tear-down
18: [==========] 1 test from 1 test suite ran. (1 ms total)
18: [  PASSED  ] 1 test.
```

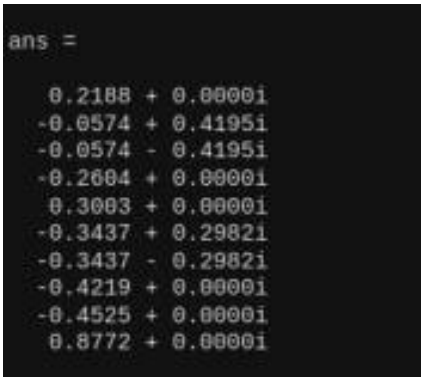
Пример кода одного из тестов, используя Google Test [3]. В нем мы создаем полином из 10 членов: $5.576312106019016 * x^{10} + 18.62488243410351 * x^9 + \cdots$. Далее мы находим корни, используя *FindPolynomialRootsJenkinsTraub(), out* - аргументы - *roots_re, roots_im* (далее выводим их и сравниваем с выводом из matlab-а).

```
TEST(Polynomial, HardPolynomial1)
{
  Eigen::VectorXd polynomial(11);
  Eigen::VectorXd roots_re(10);
  Eigen::VectorXd roots_im(10);

  polynomial(10) = -52412.8655144021;
  polynomial(9) = -28342.548095425875;
  polynomial(8) = 20409.84088622263;
  polynomial(7) = 25844.743360023815;
  polynomial(6) = 11474.831044766257;
  polynomial(5) = 1909.2968243041091;
  polynomial(4) = -692.3579951742573;
  polynomial(3) = -562.5089223272787;
  polynomial(2) = -105.89974320540716;
  polynomial(1) = 18.62488243410351;
  polynomial(0) = 5.576312106019016;

  EXPECT_TRUE(FindPolynomialRootsJenkinsTraub(polynomial, &roots_re, &roots_im));
  std::cout << "\n\n";
  std::cout << "roots_re: " << roots_re << "\n";
  std::cout << "roots_im: " << roots_im << "\n";
}
```

Вывод из matlab-а для заданного многочлена:



Для получения результатов из matlab-а необходимо зайти на https://www.mathworks.com/matlabcentral/fileexchange/50462-polynomial-roots-with-jenkins-traub-algorithm [4] и выбрать *Open in MATLAB Online*, скомпилировать программу *make_polyroots.m* [1], нажать на этот файл и выбрать *Run* в панеле сверху, после чего появится файл *polyroots.mex64* в левой панеле. Далее вводим в терминале *polyroots()*, подставляя значения в () в виде массива: []. Например, для вычисления корней уравнения $2x^2 + 3x$ нужно запустить так: *polyroots([2,3])* и нажать *Enter*. В выходных данных видим массив из корней, где первое слагаемое - действительный корень, а второе - мнимый, со знаком *i*.

Так как при нахождении корней могут быть погрешности и неточности, для сравнения с действительными корнями был использован ε. Если корень находится ближе, чем *(корень - ε)*, то корень найден правильно. В тестах за ε была взята величина 1e-10 (e^{-10}).

Одним из недостатков метода является то, что его трудно применить к задаче выделения корней в интересующей области, задаче, для решения которой алгоритмы разбиения особенно хорошо подходят.

Этот метод, как правило, непригоден для использования многочленов степени n > 50 и проблем сходимости вокруг множественных корней и кластеров корней. Тем не менее, его стабильность и скорость сходимости делают его одним из основных методов для нахождения корней многочленов.

Источники

1. Github: https://github.com/Sunaked/jenkins_traub
2. https://wiki5.ru/wiki/Jenkins%E2%80%93Traub_algorithm#What_gives_the_algorithm_its_power
3. https://github.com/google/googletest
4. https://www.mathworks.com/matlabcentral/fileexchange/50462-polynomial-roots-with-jenkins-traub-algorithm
5. https://github.com/jervisfm/JenkinsTraub
6. https://www.mathworks.com/matlabcentral/fileexchange/50462-polynomial-roots-with-jenkins-traub-algorithm?s_tid=prof_contriblnk
7. http://www.hvks.com/Numerical/Downloads/cpoly_ver_1.cpp
8. https://github.com/dkogan/PDL/tree/master/Basic/Math
9. http://www.crbond.com/downloads/rpoly.cpp
10. https://www.akiti.ca/rpoly_ak1_cpp.html
11. https://github.com/sweeneychris/RpolyPlusPlus
12. https://math.stackexchange.com/questions/1384741/how-to-scale-polynomial-coefficients-for-root-finding-algorithms