Введение

Уже много лет рынок информационных технологий удерживает огромные капиталы, причём со временем наблюдается их стремительный рост, что говорит о востребованности на рынке качественных программных продуктов. Абсолютно очевидно, что хорошо сделанные программы требуются абсолютно везде (где готовы платить), начиная с базы данных местной поликлиники, продолжая реализацией математических моделей в московских НИИ, кончая фильтрами в Instagram и спецэффектами в кино. За многими из этих программ стоит математика, причём достаточно сложная и требующая интенсивных вычислений, занимающих порой до 80% времени работы приложений. Я считаю, что такая ситуация затормаживает наш прогресс и отдаляет то прекрасное за счёт сверхумных технологий будущее, которое можно увидеть в фильмах. Частичное решение этой проблемы – оптимизация вычислительных алгоритмов, которая остаётся актуальной ещё с тех времён, когда компьютеры назывались электронно-вычислительными машинами (ЭВМ); это верно по той причине, что ресурсы компьютеров остаются ограниченными, особенно это касается мощности процессоров, которая не может превысить отметку в несколько гигагерц[1]; да, создание многоядерных процессоров намного лучше, чем ничего, но многоядерность позволяет уменьшить время работы программ (при умелом использовании) лишь в несколько раз, чего недостаточно, вдобавок это бывает дорого. И первое, что нужно сделать для ускорения программы, – оптимизировать алгоритмы.

Почему я написал эту книгу

Когда я писал диплом, мне на своём опыте пришлось прочувствовать, как, вроде бы, не такая уж сложная программа работала почти весь день на моём компьютере с 4-ядерным процессором и делала лишь две трети того, что требовалось. Тогда же я писал красивое приложение, решающее сложную волновую задачу, но вся его красота портилась тем фактом, что на более-менее точные вычисления каждый раз уходило около 10-ти минут работы. Я понимал, что оптимизировать надо конкретно реализацию математических алгоритмов, но, в принципе, нигде не мог найти литературу на эту тему; везде я наталкивался только на книги, где описывались одни и те же алгоритмы сортировки, поиска в ширину и т. п. (впрочем, их знать очень полезно, но это не совсем то), да не встречал ничего касательно именно улучшения алгоритмов, связанных, к тому же, с высшей математикой. Иногда попадались неплохие книжки, но они либо содержали мало полезной информации, либо были уже очень старые для применения на практике в наши дни (описанные там методы оптимизации учитывали архитектуру старых компьютеров и компиляторов языков Fortran, АЛГОЛ, C и сейчас не имеют силы, поскольку даже компиляторы стараются оптимизировать программу[2]), а обычно – и то и другое. Спустя время я сам решил написать такую книгу, основываясь на собранных знания и собственном опыте, чтобы другие не повторяли моих ошибок.

Примеры

За некоторыми исключениями я отталкиваюсь от принципа, что **в научных приложениях скорость работы программы на порядок важнее затрат памяти**, поэтому основная часть методов оптимизации рассчитана на ускорение; очень хорошо (и так бывает нередко), если специальный алгоритм не только ускоряет программу, но и снижает её затраты памяти, но так бывает не всегда. Скоростью следует жертвовать лишь в тех случаях, когда программа "сжирает"очень много памяти в местах, занимающих малое время от работы всего приложения.

[1] На сайте https://www.iguides.ru/main/other/pochemu_chastoty_protsessorov_ne_rastut_vyshe_neskolkikh_gigagerts/ можно узнать почему

[2] В статье https://en.wikipedia.org/wiki/Fortran можно увидеть пример, где скорость работы программы на фортране сильно зависит от того, в каком порядке обходятся измерения массива. Такие тонкости оптимизации относятся к конкретному языку и быстро становятся недействительными, если язык развивается

Книга состоит из набора принципов оптимизации, почти все из них подкрепляются примерами использования, из реальных приложений или выдуманными, также я старался проводить тестовые примеры на скорость работы: если не оговорено обратного, в тестовых примерах происходит сравнение разных алгоритмов при разных размерностях задачи, время работы есть усреднённое время при минимум десяти испытаниях. В книге я старался описывать правила оптимизации в как можно более общем виде, без привязки к конкретному языку программирования. Примеры использования этих принципов написаны либо на C#, либо на D, очень редко на F#, но должны быть понятны, полагаю, всем, кто имел дело с такими языками, как C/C++/Java/JavaScript/Go/Python/Fortran и другими языками более-менее общего назначения (без уклона в функциональное программирование, базы данных и разметку).

Дополнительные замечания

Сам я раньше писал и на QBasic (в школе), и на Pascal и C++ (на первых курсах универа), но по объективным причинам отдал предпочтение C#, хотя что-то могу написать и на F#, и на D. Примеры приводятся конкретно на C#/D/F#, потому что и мне так удобней, и языки по структуре реализации разные, что позволяет сравнивать одни и те же вещи в разных условиях. Но примеры я постарался писать так, чтобы даже не сразу можно было понять, написаны они на C++/D/C# или Java. А вообще я рекомендую учить сразу несколько языков программирования, хотя 1 следует знать, конечно, почти полностью и следует посвятить ему несколько лет отдельно. Со временем мне стало ясно, что с языками нужно быть толерантным, поскольку другой язык какие-то вещи опишет лучше и быстрее: например, сама эта книга написана языком разметки LaTeX, поскольку в MS Word слишком тяжело использовать математические выражения, дополнительно следя за отступами и пр.; а многие приводимые графики сделаны с помощью пакетов языка R, поскольку чистый C# плохо поддерживает некоторые виды математических графиков.

Также я рекомендую отдельно получить более чем базовые знания по алгоритмам и численным методам, потому что многие из них всё равно придётся реализовывать самостоятельно. К тому же, полезно знать, что, например, сортировку можно выполнить за время $O(n \log n)$, чтобы прикидывать, насколько ваша реализация оптимальна.

Чтобы укрепить полученные знания практикой рекомендую время от времени решать задачи на http://codeforces.com, где все задачи имеют адекватные требования по времени и памяти.

1 Оптимизация алгоритмов

Моя история программирования начинается с того дня, когда в книге [2], которую я из скуки стал читать в трамвае, чтобы не тратить время впустую, встретилась следующая цитата:

> Основная побудительная причина изучения алгоритмов состоит в том, что это позволяет обеспечить огромную экономию ресурсов, вплоть до получения решений задач, которые в противном случае были бы невозможны. В приложениях, в которых обрабатываются миллионы объектов, часто оказывается возможным ускорить работу программы в миллионы раз, используя хорошо разработанный алгоритм ... Для сравнения, вложение дополнительных денег или времени для приобретения и установки нового компьютера потенциально позволяет ускорить работу программы всего в 10-100 раз. Тщательная разработка алгоритма — исключительно эффективная часть процесса решения сложной задачи в любой области применения.

Тогда я понял, как важно уметь писать не просто работающие программы, но работающие достаточно быстро. Возможно, многие программисты далеко не сразу понимают, что же такого может делать программа, выполняющаяся видимое время, но я учился на матфаке и со временем стал сталкиваться с задачами, в которых множество раз требуется решать системы линейных алгебраических

уравнений, элементами которых являются интегралы от функций, которые сами могут задаваться через интегралы, при этом задача решается в нескольких измерениях и т. д. (и почти все задачи прикладной математики насколько требовательны к вычислениям), а программы даже при хорошей оптимизации работали несколько минут. Позднее я узнал, как легко в .NET-языках проводить распараллеливание, но, даже если не брать во внимание некоторые нюансы, распараллеленная программа была быстрее однопоточной всего лишь в несколько раз, что со временем переставало воодушевлять. По этой причине в научных вычислениях требуется в первую очередь использовать оптимизацию алгоритмов, к тому же, – немаловажно, – в качестве языков программирования не нужно использовать "медленные"(JavaScript,Python); но при этом не нужно и входить в крайность и писать на фортране, потому что это "язык научных вычислений, в учёных кругах пишут на нём": на древних языках писать полезные и современные программы слишком сложно, а иногда и невозможно, а всякие "нововведения"типа ленивых вычислений и параллельных циклов, просто украденные у современных языков, – это попытки удлинить жизнь смертельно больного, которые просто не вписываются в язык, придуманный 60 лет назад; лишь несколько языков, пережив перезапуск, продолжают продуктивно по современным меркам использоваться (например, Visual Basic), другие же (тот же Object Pascal) ненавидимы огромным количеством людей и продолжают преподаваться только потому, что сами преподаватели уже слишком старые и консервативные, чтобы переучиться.

Итак, для работы программа нуждается в **системных ресурсах**; под системными ресурсами подразумеваются *память* и *время*. Если говорить грубо, для экономии времени нужно меньше вычислять, для экономии памяти – создавать меньше объектов и поменьше вызывать рекурсивные функции, если только вы работаете не с функциональным языком программирования, где рекурсии оптимизированы. Во многих простых случаях выигрыш в памяти и выигрыш в скорости взаимодополняемы (например, если не создавать массив много раз, не потратится и время на его создание), но иногда приходится крупно жертвовать либо тем, либо другим, но тогда и выигрыш будет крупным.

Любому разработчику необходимо хотя бы иметь представление о наиболее оптимальных вариантах часто использующихся конструкций. Многие разработчики усваивают принципы оптимизации с годами опыта, но вам не обязательно повторять чужие ошибки.

1.1 Общие правила оптимизации, дающие выигрыш и в памяти и в скорости

1.1.1 Замена условных выражений на эквивалентные

Логическое выражение может иметь только значения true и false (если оно выполняется без ошибок), поэтому если, например, два объекта отличаются друг от друга, то в логическом выражении не имеет значения, насколько сильно они отличаются, поэтому не имеет смысла считать характеристики этих объектов через трудоёмкие выражения, если можно заменить эти выражения эквивалентными.

Например, для действительных векторов размерности n справедливы следующие нормы:

$$||x||_1 = \sum_{i=1}^{n} |x_i| \tag{1}$$

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i \cdot x_i}, \tag{2}$$

$$||x||_3 = \max_{1 \le i \le n} |x_i|, \tag{3}$$

которые (впрочем, как и любые две нормы в конечномерном пространстве пространстве) эквивалентны между собой, а из этого следует, что если для некоторых векторов x, y верно $||x||_2 \le ||y||_2$, то

верно и $||x||_3 \leq \alpha||y||_3$ [3], как и наоборот. Однако с вычислительной точки зрения вычисление нормы $||\cdot||_2$ требует ровно n произведений, $n-1$ сложений и 1 вычисление корня (всего $2n$ операций), покуда вычисление $||\cdot||_3$ требует **максимум** n вычислений модуля и $n-1$ сравнений, то есть максимум $2n-1$ операций и n операций в среднем[4]. В принципе, $2n-1$ операций требует и вычисление $||\cdot||_1$, но тут существует и большая разница в точности, особенно при больших n: если численно выражение $||\cdot||_3$ будет иметь одно и то же значение при любом расположении компонент в векторе, то $||\cdot||_1$ и $||\cdot||_2$ всегда зависят от порядка по причине ошибок округления при суммировании, причём ошибки могут быть значительными, когда вектор содержит сильно отличающиеся по модулю элементы. По этой причине два действительных вектора лучше сравнивать по норме $||\cdot||_3$. Это правило ещё более верно, когда векторы комплексные, поскольку вычисление модуля комплексного числа ($|x+iy| = \sqrt{x^2+y^2}$) требует 4 операции, а произведение комплексных чисел ($(a,b)\cdot(c,b) = (ac-bd, ad+bc)$) – 6 операций (не забудьте, что в комплексном случае должно быть взято комплексное сопряжение). Аналогичные и более выигрышные правила будут действовать и для матриц. Для двух функций $f, g \in C[a,b]$, если они не имеют сильных скачков и если задача не требует конкретной нормы, не рекомендуется считать равномерную норму по 100+ точкам (хоть она и существует в пространстве $C[a,b]$), если вполне точно можно вычислить среднеквадратичную норму по формула Гаусса-Кронрода (15-61 точек).

Другой крайне показательный пример был описан в книге [1] (раздел 9.2). Для каждой из 20000 точек требовалось найти расстояние до множества из 5000 точек, причём все точки находились на сфере и имели сферические координаты; расстояние между точками считалось по формулам римановой геометрии, то есть:

$$\rho(a,b) = \arccos\left(\sin\phi_a \sin\phi_b + \cos\phi_a \cos\phi_b \cos(\lambda_a - \lambda_b)\right),$$

но когда заметили, что это расстояние эквивалентно $(a_x-b_x)^2 + (a_y-b_y)^2 + (a_z-b_z)^2$ и поменяли соответствующие выражения, программа заработала быстрее в 300 раз.

Замечания. Возможно, замена нормы $||\cdot||_2$ на норму $||\cdot||_3$ кажется несколько надуманной, так как, по сути, они требуются почти одинакового числа операций, но этот пример использовался лишь как идея. С другой стороны, при сравнении нормы вектора с 0 (что делается очень часто), использование максимума по элементам намного разумнее их суммирования.

1.2 Ускорение в ущерб памяти

1.2.1 Расширение структуры данных ради упрощения вычислений

Часто бывает полезным вместо прямого использования объекта создать некоторый класс, ему эквивалентный. При этом в самом классе будут находиться другие структуры данных, из-за чего мы проигрываем в памяти, однако упрощаем задачу сопровождения кода и можем его ускорить. В одном из предыдущих разделов описывался пример, когда сферическое расстояние заменили евклидовым и сильно выиграли в скорости, однако для этого пришлось к точке с координатами ϕ, λ добавить координаты x, y, z, что примерно в 2,5 раза увеличило затраты на память.

Другой пример – реализация полиномов как массивов с некоторыми свойствами. Полиномы – особые функции, которые нужны во многих задачах и которые наиболее хорошо исследованы. Полиномы используются при интерполяции, аппроксимации, и часто тестирование методов численной математики начинают на полиномах. И чтобы работа с полиномами была максимально быстрой и точной, их нужно реализовать как векторы (массивы) с некоторым поведением; в таком случае не только быстрее считается значение полинома в точке, но также вполне нетрудно выполнять дифференцирование, интегрирование, умножение и деление полиномов почти с максимальной для машин точностью.

[3] Это следует из выражения $\alpha_1||x||_1 \leq ||x||_2 \leq ||y||_2 \leq \alpha_2||y||_1$, которое следует из определения эквивалентности $\alpha_1||x||_1 \leq ||x||_2 \leq \alpha_2||x||_1$, но при этом важно учитывать, что сам коэффициент $\alpha = \frac{\alpha_2}{\alpha_1}$ может быть и меньше 1, поэтому $||x||_1 \leq ||y||_1$ не значит $||x||_2 \leq ||y||_2$, так что при выборе метрики нельзя забывать про контекст задачи

[4] Если вектор отчасти отсортирован

Возьмём полином $p(x) = (x-1)(x-2)(x+1)(x+2)(x+0.5)(x-0.1) = x^6 + 0.4x^5 - 5.05x^4 - 2x^3 + 4.25x^2 + 1.6x - 0.2$ с известными корнями. Его можно задать как обычную функцию таким образом:

```
F#
let p (x:double)=x**6.0+0.4*x**5.0-5.05*x**4.0-2.0*x**3.0+4.25*x**2.0+1.6*x-0.2
```

В таком случае она будет считаться не очень быстро и не очень точно из-за прямых возведений x в степени. Можно исправить эту ситуацию, заменив выражение типа $c_0x^n + c_1x^{n-1} + \cdots + c_{n-1}x + c_n$ схемой Горнера $(\ldots((c_0x + c_1)x + c_2)x\ldots)x + c_0$, однако так придётся поступать для каждого полинома. Разумнее оформить полиномы классом и вставить в класс все нужные методы, тогда полином отождествится с набором своих коэффициентов, но это случится в ущерб памяти. Первая полезная вещь, которую я сделал на C# – написал такой класс. Результаты его работы в сравнении с описанной функцией на F#:

1.2.2 Мемоизация

Замечание. Не следует мемоизировать всё подряд слишком обобщённо! Мемоизация каких-то очень часто используемых функций может сжирать слишком много памяти и образовывать настолько большой словарь, что поиск по словарю может оказаться в разы медленнее прямого вычисления функции. Сохраняйте значений только тех функций, которые считаются достаточно долго, либо сохраняйте очень осторожно. Поясню на примере.

В одном приложении мне нужно было несколько сотен раз считать следующий интеграл:

$$\mathbf{u}(x,y,z) = \frac{1}{2\pi}\iint_{-\infty}^{\infty} K(\alpha_1,\alpha_2,z)Q(\alpha_1,\alpha_2)e^{-i(\alpha_1 x+\alpha_2 y)}d\alpha_1 d\alpha_2 = \\ = \frac{1}{2\pi}\int_{\Gamma^+}(K*J)(\alpha,x,y,z)Q(\alpha)\alpha d\alpha, \quad (4)$$

где Γ^+ – положительная действительная полуось с отклонением в комплексную плоскость на одном участке. Этот интеграл требовал как минимум нескольких сотен подсчётов такой матрицы:

$$(K * J)(\alpha,x,y,z) = \begin{pmatrix} i\left(M\frac{\partial^2 J_0(\alpha r)}{\partial x^2} + N\frac{\partial^2 J_0(\alpha r)}{\partial y^2}\right) & i(M-N)\frac{\partial^2 J_0(\alpha r)}{\partial x\partial y} & -P\frac{\partial J_0(\alpha r)}{\partial x} \\ i(M-N)\frac{\partial^2 J_0(\alpha r)}{\partial x\partial y} & i\left(M\frac{\partial^2 J_0(\alpha r)}{\partial y^2} + N\frac{\partial^2 J_0(\alpha r)}{\partial x^2}\right) & -P\frac{\partial J_0(\alpha r)}{\partial y} \\ -iS\frac{\partial J_0(\alpha r)}{\partial x} & -iS\frac{\partial J_0(\alpha r)}{\partial y} & R \end{pmatrix}, \quad (5)$$

где производные J_0 выражаются через функции Бесселя

$$\frac{\partial J_0(\alpha r)}{\partial x} = -J_1(\alpha r)\frac{x}{r}, \quad (6)$$

$$\frac{\partial J_0(\alpha r)}{\partial y} = -J_1(\alpha r)\frac{y}{r}, \quad (7)$$

$$\frac{\partial^2 J_0(\alpha r)}{\partial x^2} = \frac{1}{\alpha r^3}\left(\alpha r J_2(\alpha r)x^2 - J_1(\alpha r)(x^2+\alpha y^2)\right), \quad (8)$$

$$\frac{\partial^2 J_0(\alpha r)}{\partial y^2} = \frac{1}{\alpha r^3}\left(\alpha r J_2(\alpha r)y^2 - J_1(\alpha r)(y^2+\alpha x^2)\right), \quad (9)$$

$$\frac{\partial^2 J_0(\alpha r)}{\partial x\partial y} = \frac{xy}{\alpha r^3}\left(\alpha r J_2(\alpha r) - J_1(\alpha r)(1-\alpha)\right), \quad (10)$$

$$r = \sqrt{x^2+y^2}, \quad (11)$$

а компоненты M, N, S, P, R – это скалярные произведения вектора с экспонентами и столбцов обратной матрицы от достаточно плохо обусловленной матрицы, эти функции зависели от переменных $\alpha, z = 0$ и считались все вместе одной функцией (возвращавшей комплексный вектор). Поскольку эти функции были достаточно трудно вычисляемы, я их мемоизировал. Программа заработала примерно в 5 раз быстрее, вдобавок я получил возможность оценивать её работу ещё в процессе выполнения, отслеживая длину получающегося словаря: при нормальной работе длина словаря была в пределах 6-10 тысяч, а если она возрастала до 15 тысяч и больше, это говорило о том, что контур интегрирования проходит близко к комплексным полюсам функций M, N, R, S, P, из-за чего метод интегрирования дробит шаг и вычисляет намного больше значений; в таком случае программа будет работать в 2-3 раза дольше, а в результате больше трети найденных значений интеграла будут либо выбросами, либо NaN; и теперь мне не нужно было ждать полчаса, чтобы это узнать, потому что я смог отслеживать качество по словарю.

Но работа программы в полчаса мне казалась слишком длинной, поэтому я решил провести мемоизацию ещё и функций Бесселя $J_1(\alpha r), J_2(\alpha r)$ (как единую функцию двух аргументов, порядка и обычного аргумента $c = \alpha r$), поскольку их вычисления тоже повторялись много раз (но уже не были настолько затратными). В итоге программа стала требовать на 150Мб больше памяти и выполнялась в два раза дольше, потому что легче оказалось подсчитывать тысячи значений, чем сохранять и искать. Тогда я подумал и понял, что мне не нужно множество функций Бесселя J_ν, а нужно лишь две, поэтому я их мемоизировал по отдельности; памяти потребовалось всего на 20Мб больше, программа стала считать быстрее на 5%, но при расширении задачи это всё равно требовало огромных затрат на память, что меня не устроило, поэтому я убрал мемоизацию функций Бесселя.

1.2.3 Кэширование

1.3 Экономия памяти в ущерб скорости

2 Распараллеливание

3 Упрощению процесса программирования

3.0.1 Выделение повторяющихся или независимых блоков в отдельные методы

Хоть этот принцип и кажется очевидным, многие используют его лишь наполовину, программируя какие-то вычислительные участки программы, а в остальном забывая. Я видел код, в котором решение СЛАУ (как бы самая важная функция программы) выделялось в отдельный метод, но при этом операции, например, вывода матрицы на консоль (для отладки) представляли собой одни и те же строки вложенных циклов, которые в неизменном виде присутствовали в пяти местах. Это засоряло код и вынуждало делать одни и те же изменения в пяти (!) местах, чтобы исправить какие-то несовершенства. Ещё хуже бывает ситуация при проектировании пользовательского интерфейса: какие-то вычислительные операции могут выполняться в отдельных классах, расположенных в специально сделанной библиотеке, но код взаимодействия с окном или формой написал "в лоб как будто его писал школьник; и по этой причине расширение функциональности приложения становится всё более трудной задачей.

3.0.2 Комбинирование разных методов решения задачи

3.0.3 Решение более простых задач, пусть и с лишними решениями

В одном проекте мне была поставлена промежуточная задача найти на некотором отрезке все (или хотя бы все) корни функции комплексного переменного вида

$$\Delta(\alpha, \omega) = 4adbc + (ad - bc)^2 \operatorname{ch}((\sigma_1 - \sigma_2)h) - (ad + bc)^2 \operatorname{ch}((\sigma_1 + \sigma_2)h) \tag{12}$$

для разных ω, причём $a = 2\mu\alpha^2 - (2\mu + \lambda)\kappa_1^2, b = 2i\mu\alpha^2\sigma_1, c = 2\mu\alpha^2\sigma_2, d = -i\mu\alpha^2(2\alpha^2 - \kappa_2^2), \sigma_i = \sqrt{\alpha^2 - \kappa_i^2}, \kappa_i = \kappa_i(\omega^2)$. Одной из особенностей этой функции являются экспоненты внутри гиперболических косинусов, из-за чего она очень быстро растёт и убывает. Ясное дело, я не мог взять эти корни в явном виде, поэтому искал их численно с помощью процедуры (метода), который за полгода до этого переписал с фортрана. Этим методом у нас пользовались уже два десятка лет, он работал вполне быстро, занимал всего строк 40 кода и находил намного больше корней, чем все методы, которые я до этого написал самостоятельно. Разумно полагать, что раньше у всех он работал приемлемо, но, к сожалению, именно на моей задаче стал находить лишние корни, то есть точки, в которых значение функции было не близким к нулю, а иногда было и двадцатизначным; кроме того, даже при распараллеливании в моём случае программа, использующая этот метод, начала работать заметные секунд 8-10, что было плохим знаком.

Я потратил несколько дней на решение проблемы. Искусственно удалив "лишние"корни, я увидел, что почти ничего не оставалось, то есть метод работал в корне плохо. Я взглянул на реализацию метода, но совсем ничего не понял, поскольку код был написал на древней версии фортрана ещё в 90-х годах, половина переменных имела непонятные названия, а ещё каждые 5 строк встречался оператор goto; я лишь смог понять, в каких местах происходит занесение корней в массив и в каком из мест заносятся неправильные корни, но не было ясно, что и где надо бы исправить. Тогда я попытался использовать эту процедуру на функции $\gamma\Delta$, где γ – некоторое число; очевидно, что корни этой функции будут те же самые, но за счёт изменения коэффициента можно как-то сгладить резкие скачки исходной функции Δ; как оказалось, ни маленькие γ, ни большие ничего не изменяли – разве что при очень больших γ происходило переполнение и в итоге ничего не находилось. Тогда я написал свою реализацию этого метода с использованием алгоритма двоичного поиска, причём написал несколько немного разных по сути методов, и все они стали находить в точности те же самые точки, что и исходная процедура с фортрана. Появилось предположение, что одна из составляющих проблемы – критерий существования корня на маленьком отрезке $[a, b]$; известно, что на отрезке $[a, b]$ комплексная функция $f(z)$ имеет нечётное число корней, если $f(a).\mathrm{Re} \cdot f(b).\mathrm{Re} \wedge f(a).\mathrm{Im} \cdot f(b).\mathrm{Im}$, но что делать, если вдруг на конце отрезка либо внутри действительная часть функции равна 0, а мнимая не равна? В общем, ответы так и не были найдены.

И чудом мне пришла идея решить более простую задачу поиска корней действительнозначной функции $f := f.\mathrm{Re} + f.\mathrm{Im}$, а после просто отсеять лишние корни. Реализация этой идеи заняла 5 минут, корни были найдены, а время работы программы даже уменьшилось, так как я включил в свой метод некоторые чужие, а отрезок с корнями всё равно обходился с маленьким шагом. Кроме этого, можно было бы искать корень на малом отрезке несколькими разными методами, ведь всё равно в конце процедуры стояла команда "отсеять лишнее и убрать повторы"(result.Distinct().Where(...)).

Вывод. Если исходную задачу решить трудно, попробуйте решить более простую задачу, чьи решения содержат и решения исходной, а затем отсейте лишнее.

Замечания. Я не использовал замену $f := f.\mathrm{Re} \cdot f.\mathrm{Im}$, потому что она потенциально давала намного больше лишних решений, к тому же плохо помогала ввиду резкого роста функции Δ (двадцатизначные значения ещё разумно складывать, но уж не перемножать). Так же можно заметить, что из Δ можно достать слагаемое, кратное α^4, пусть разницы я после этого не увидел.

4 Увеличение точности

4.1 Суммирование от меньшего к большему. Сортировка перед суммированием

Список литературы

[1] Жемчужины программирования. 2-е издание. — СПб.: Питер, 2002. — 272 с:

[2] Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик. - К.: Издательство «ДиаСофт», 2001.- 688 с.

[3] Вычислительная математика И программирование: Учеб. пособие для студентов втузов.——М.: Высш. шк., 1990.- 544 с.: