Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Маргарита Вадимовна Лашина

Выпускная квалификационная работа

Реализация поточной версии алгоритма Левенберга-Марквардта

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2019
«Современное программирование»

Научный руководитель: доцент, факультет математики и компьютерных наук, к.ф.- м.н., Д. С. Шалымов

Рецензент: зам.зав. лабораторией МФТИ, к.ф.-м.н. А. А. Нозик

Санкт-Петербург
2023 г.

Содержание

Введение

Быстрый рост объемов данных в сегодняшнюю цифровую эпоху произвел революцию в области алгоритмов оптимизации, создав новые задачи и возможности для решения крупномасштабных задач в режиме реального времени. В частности, методы нелинейной оптимизации, используемые для оценки параметров нелинейной математической модели путем минимизации разницы между наблюдаемыми данными и предсказаниями модели, играют решающую роль в различных областях науки и техники, от компьютерного зрения и машинного обучения до обработки сигналов и систем управления, где модели демонстрируют нелинейное поведение.

Одним из известных методов нелинейной оптимизации методом наименьших квадратов является алгоритм Левенберга-Марквардта. Алгоритм Левенберга-Марквардта, представляющий собой слияние методов градиентного спуска и методов Гаусса-Ньютона, демонстрирует замечательные свойства сходимости, что делает его популярным выбором для решения сложных задач оптимизации. Однако, использование столь полезного алгоритма может быть затруднено, поскольку существующие реализации есть лишь на нескольких языках и не могут быть запущены на многих платформах. В современном технологическом мире появляется много приложений алгоритмов, каждое из которых может нуждаться в своей платформе для запуска, поэтому вопрос наличия вычислительных методов, которые могут адаптироваться и использовать разные платформы, очень актуален. Одним из проектов, занимающихся разработкой мультиплатформенной математической библиотеки, является проект KMath. Библиотека KMath разработана для решения разнообразных задач, связанных с математикой и научными вычислениями и ее мультиплаформенность достигается благодаря использовании технологии Kotlin Multiplatform. В рамках этой дипломной работы был разработан алгоритм Левенберга-Марквардта на языке Kotlin и внедрен в кодовую базу проекта KMath с дальнейшим возможным использованием алгоритма любым пользователем этой библиотеки на различных платформах.

Традиционный алгоритм Левенберга-Марквардта описан в первую очередь для обработки статических наборов данных, и в эпоху больших данных,

когда данные генерируются и обновляются с беспрецедентной скоростью, возникает потребность в методах оптимизации, которые могут справиться с динамичными потоковыми данными. В этой работе также представлена поточная версия алгоритма Левенберга-Марквардта, которая является адаптацией, предназначенной для обработки динамических данных, предлагающая значительные преимущества для приложений, которые получают данные в реальном времени.

Постановка задачи

Основными целями данной ВКР являются разработка мультиплатформенной статической версии алгоритма Левенберга-Марквардта на языке Kotlin, а также разработка поточной версии алгоритма с использованием статического варианта. Для достижения этих целей были поставлены следующие задачи:

1. Выполнить обзор предметной области.
2. Реализовать статическую версию алгоритма Левенберга-Марквардта на языке Kotlin в мультиплатформенном проекте KMath.
3. Провести тестирование алгоритма с запуском на разных платформах.
4. Реализовать поточную версию алгоритма Левенберга-Марквардта и провести тестирование.

1. Обзор

1.1. Технология Kotlin Multiplatform и проект KMath

В современной экспериментальной науке значительная часть работы связана с компьютерами и программным обеспечением. Это означает, что качество программного обеспечения и инструментов для разработки существенно влияет на эффективность работы. Традиционно, разработка программного обеспечения для научных целей производится на языках C++ и Python. Проблема использования C++ заключается в том, что для поддержания разработки требуется большой опыт программирования. Язык Python, с другой стороны, предоставляет широкий спектр инструментов и не требует высококлассных навыков работы с ним. Проблема языка Python в том, что он не предназначен для крупномасштабной разработки. Разработка на Python ограничена его динамической типизацией и низкой производительностью в режиме интерпретации. В этой ситуации очень важно искать новые инструменты и язык Kotlin оказывается хорошим решением благодаря простоте работы с ним, удобству ведения крупномасштабных разработок, неплохой скорости и главное, наличию технологии Kotlin Multiplatform [4].

Kotlin Multiplatform (KMP) — это технология, предоставляемая языком программирования Kotlin, основная идея которой состоит в том, чтобы иметь возможность писать общий код, который может быть переиспользован на различных платформах, в то время как платформозависимая часть кода может быть написана с учетом особенностей каждой платформы. Kotlin Multiplatform позволяет использовать код на таких платформах как JVM, Android, iOS, JavaScript и других. Кроме того, Kotlin Multiplatform полезен для авторов библиотек. Технология позволяет создавать библиотеки с общим кодом и его специфичными для платформы реализациями. Поддерживается несколько сред выполнения: виртуальная машина Java (JVM), браузер и нативная компиляция. Одной из библиотек, использующих технологию Kotlin Multiplatform, является проект KMath.

KMath[1] — это библиотека, разработанная на платформе Kotlin для

[1] https://github.com/SciProgCentre/kmath

решения разнообразных задач, связанных с математикой и научными вычислениями. Библиотека является основанным на Kotlin аналогом Python библиотеки NumPy, однако, в отличие от numpy и scipy, она модульная и имеет легковесное ядро. Это позволяет подгружать только нужные модули при работе с ней.

Главными особенностями библиотеки KMath являются:

1. Многоплатформенная поддержка: KMath поддерживает Kotlin/JVM, Kotlin/JS и Kotlin/Native, что позволяет разработчикам использовать его функции на разных платформах.

2. Алгебраические структуры. Библиотека включает набор алгебраических структур для облегчения работы со сложными математическими типами данных и операциями.

3. Расширенные числовые типы: KMath расширяет собственные числовые типы Kotlin (такие как целые числа, числа с плавающей запятой) комплексными числами, векторами и матрицами, которые часто необходимы для математических вычислений.

4. Математические функции. KMath содержит набор стандартных математических функций, таких как тригонометрические, логарифмические, экспоненциальные и специальные функции.

5. Линейная алгебра. KMath поддерживает операции работы с матрицами, что позволяет разработчикам проще решать задачи линейной алгебры.

6. Расширяемость: KMath поддерживает пользовательские реализации математических объектов и операций, что позволяет разработчикам легко расширять библиотеку в соответствии со своими конкретными потребностями.

7. Интеграция с другими библиотеками: KMath можно легко интегрировать с другими популярными библиотеками Kotlin, такими как kotlinx.serialization для сериализации математических структур или kotlinx.coroutines для параллельных вычислений.

Резюмируя, KMath — это мощная и гибкая библиотека, а главное, она написана с использованием технологии Kotlin Multiplatform. Код, полученный в ходе работы над данной ВКР, разработан в рамках проекта KMath. Благодаря этому, он может быть использован на разных платформах, что является главной уникальностью этой работы.

1.2. Статический алгоритм Левенберга-Марквардта

Алгоритм Левенберга-Марквардта [1, 3] — это один из существующих методов нелинейной оптимизации, который используется для нахождения оптимальных параметров модели, которые наилучшим образом описывают набор данных. Это итерационный алгоритм, который сочетает в себе преимущества метода градиентного спуска, который хорошо работает для линейных задач, и метода Гаусса-Ньютона, который хорошо работает для нелинейных задач. Он был разработан в 1944 году американским математиком Джоном Левенбергом и датским математиком К. Г. Марквардтом и широко используется в различных областях, таких как подбор кривых, компьютерное зрение, робототехника и оценка параметров.

При подгонке функции $\hat{y}(t, p)$ независимой переменной t и вектора из n параметров p к набору из m точек (t_i, y_i) обычно удобно минимизировать сумму взвешенных квадратов ошибок:

$$\chi^2(p) = \sum_{i=1}^{m} \left(\frac{y(t_i) - \hat{y}(t_i; p)}{\sigma_{y_i}} \right)^2$$

где σ - ошибка измерения для $y(t_i)$. Обычно весовая матрица W равна диагональной со значениями $1/\sigma_{y_i}^2$. Более формально, W может быть приравнена к обратной ковариационной матрице ошибки измерения, в необычном случае, когда она известна. Если функция $\hat{y}(t, p)$ нелинейна по параметрам модели p, то минимизацию $\chi^2(p)$ по параметрам необходимо проводить итеративно. Цель каждой итерации состоит в том, чтобы найти возмущение h параметров p, уменьшающее $\chi^2(p)$.

Метод градиентного спуска — это метод минимизации, который итеративно обновляет значения параметров в направлении, противоположном

градиенту целевой функции. В этом случае градиент χ^2 по отношению к параметрам равен

$$\frac{\partial}{\partial p}\chi^2 = 2(y - \hat{y}(p))^T W \frac{\partial}{\partial p}(y - \hat{y}(p)) = -2(y - \hat{y}(p))^T W J$$

где J - матрица Якоби $\left(\partial\hat{y}/\partial p\right)$. Обновление возмущения h, которое перемещает параметры в направлении наискорейшего спуска задается выражением

$$h = \alpha J^T W (y - \hat{y})$$

где положительная скалярная величина α определяет длину шага в направлении наискорейшего спуска.

Метод Гаусса-Ньютона — это также итеративный метод минимизации. Возмущение h в этом методе используется следующее:

$$h = \frac{J^T W (y - \hat{y})}{J^T W J}$$

Алгоритм Левенберга-Марквардта адаптивно изменяет обновления параметров между методами градиентного спуска и Гаусса-Ньютона.

$$h = \frac{J^T W (y - \hat{y})}{J^T W J + \lambda \cdot I} \tag{1}$$

Параметр λ может также масштабироваться с помощью $diag(J^T W J)$.

$$h = \frac{J^T W (y - \hat{y})}{J^T W J + \lambda \cdot diag(J^T W J)} \tag{2}$$

На итерации i шаг h оценивается путем сравнения $\chi^2(p)$ с $\chi^2(p + h)$. Шаг совершается, если метрика ρ превышает заданный пользователем порог ϵ_4. Эта метрика является мерой фактического улучшения $\chi^2(p)$.

$$\rho = \frac{\chi^2(p) - \chi^2(p+h)}{|(y-\hat{y})^T W (y-\hat{y}) - (y-\hat{y}-Jh)^T W (y-\hat{y}-Jh)|}$$
$$= \frac{\chi^2(p) - \chi^2(p+h)}{|h^T(\lambda_i h + J^T W (y-\hat{y}(p)))|} \quad \text{для } h \text{ из уравнения (1)} \tag{3}$$
$$= \frac{\chi^2(p) - \chi^2(p+h)}{h^T(\lambda_i diag(J^T W J) h + J^T W (y-\hat{y}(p)))} \quad \text{для } h \text{ из уравнения (2)} \tag{4}$$

Если на итерации $\rho_i(h) > \epsilon_4$, то $p+h$ значительно лучше, чем p, p заменяется на $p+h$ и λ уменьшается в несколько раз. В противном случае λ увеличивается в несколько раз, и алгоритм продолжает работу. В приведенном далее коде показаны три возможных способа инициализации и обновления λ и p:

1. $\lambda_0 = \lambda_o$; λ_o задается пользователем;
   используя уравнение (2) для h и уравнение (4) для ρ
   если $\rho_i(h) > \epsilon_4$: $p \leftarrow p+h$; λ_{i+1} =max$(\lambda_i / L_\downarrow, 10^{-7})$, где $L_\downarrow$ - задаваемый коэффициент уменьшения λ; иначе: $\lambda_{i+1} = \min(\lambda_i L_\uparrow, 10^{-7})$, где $L_\uparrow$ - задаваемый коэффициент уменьшения λ;

2. $\lambda_0 = \lambda_o \max\Big(diag(J^T W J)\Big)$; λ_o задается пользователем.
   используя уравнение (1) для h и уравнение (3) для ρ
   $\alpha = \Big((J^T W (y-\hat{y}(p)))^T h\Big) / \Big((\chi^2(p+h) - \chi^2(p))/2 + 2(J^T W (y-\hat{y}(p)))^T h\Big)$;
   если $\rho_i(\alpha h) > \epsilon_4 : p \leftarrow p + \alpha h$; $\lambda_{i+1} = \max(\lambda_i/(1+\alpha), 10^{-7})$;
   иначе: $\lambda_{i+1} = \lambda_i + |\chi^2(p+\alpha h) - \chi^2(p)|/(2\alpha)$;

3. $\lambda_0 = \lambda_o \max\Big(diag(J^T W J)\Big)$; λ_o задается пользователем;
   используя уравнение (1) для h и уравнение (3) для ρ
   если $\rho_i(h) > \epsilon_4 : p \leftarrow p+h$; $\lambda_{i+1} = \lambda_i \max(1/3, 1-(2\rho_i - 1)^3)$; $\nu_i = 2$;
   иначе: $\lambda_{i+1} = \lambda_i \nu_i$; $\qquad \nu_{i+1} = 2\nu_i$;

   Сходимость достигнута, если выполнен из следующих трех критериев

- Сходимость по градиенту:

$$\max |J^T W (y-\hat{y})| < \epsilon_1$$

- Сходимость по параметрам:

$$\max |h_i/p_i| < \epsilon_2$$

- Сходимость по χ^2:

$$\chi^2/(m-n+1) < \epsilon_3.$$

2. Реализация статической версии

2.1. Входные данные

На вход алгоритм принимает следующие параметры:

1. `func`: функция от n независимых переменных t и m параметров p, возвращающая вектор из n значений y_{hat}, при котором каждое из y_{hati} посчитано при своем t_i.
2. `p`: стартовые значения параметров.
3. `t`: независимые переменные.
4. $\texttt{y}_{\texttt{dat}}$: набор реальных данных, посчитанных при заданных t, но с неизвестными параметрами, которые необходимо найти.
5. `weight`: весовая матрица.
6. `dp`: дельта при вычислении численной производной p.
7. $\texttt{p}_{\texttt{min}}$: нижняя граница значений параметров.
8. $\texttt{p}_{\texttt{max}}$: верхняя граница значений параметров.
9. `opts`: вектор алгоритмических параметров (максимальное число итераций, ϵ_1, ϵ_2, ϵ_3, ϵ_4, λ для инициализации, $L_\uparrow$ и $L_\downarrow$).

2.2. Описание хода алгоритма

1. Инициализация входных данных по умолчанию, если необходимо.
2. Подсчет $J^T W J$, $J^T W (dy)$, χ^2, а также вектора из n значений, посчитанных для текущих параметров и подсчет J.
3. Инициализация λ.
4. Запуск цикла

(a) Подсчет возмущения h в зависимости от типа обновления, заданного в алгоритмических параметрах.

(b) Подсчет новых параметров p с использованием посчитанного возмущения h.

(c) Подсчет вектора остаточных ошибок с использованием новых параметров.

(d) Выход из цикла в случае, если одна из остаточных ошибок превысила максимальное или минимальное допустимое значение Double.

(e) Подсчет χ^2.

(f) Подсчет метрики ρ фактического улучшения χ^2.

(g) Сравнение метрики с ϵ_4, обновление параметров и λ в зависимости от типа обновления.

(h) Сохранение текущих параметров, λ, χ^2.

(i) Проверка каждого из трех типов сходимости по очереди и завершение цикла в случае успешности одной из проверок.

(j) Выход из цикла в случае достижения максимального числа итераций.

5. Возврат результатов.

3. Результаты тестирования на разных платформах

В процессе работы было проведено тестирование нескольких функций[2] . Ниже приведен псевдокод каждой из них. В этом разделе численные значения будут приводиться с округлением до 4 знаков после запятой.

```
fun funcEasyForLm(t, p, settings) {
    val m = t.shape.component1()
    var y_hat = zeros(m, 1)

    if (settings.example_number == 1) {
        y_hat = exp((t.times(-1.0 / p[1, 0]))).times(p[0, 0]) +
        t.times(p[2, 0]).times(exp((t.times(-1.0 / p[3, 0]))))
    }
    else if (settings.example_number == 2) {
        val mt = t.max()
        y_hat = (t.times(1.0 / mt)).times(p[0, 0]) +
                (t.times(1.0 / mt)).pow(2).times(p[1, 0]) +
                (t.times(1.0 / mt)).pow(3).times(p[2, 0]) +
                (t.times(1.0 / mt)).pow(4).times(p[3, 0])
    }
    else if (settings.example_number == 3) {
        y_hat = exp((t.times(-1.0 / p[1, 0]))).times(p[0, 0]) +
        sin((t.times(1.0 / p[3, 0]))).times(p[2, 0])
    }

    return y_hat.as2D()
}
```

Листинг 1: Пример простой тестовой функции

Запуск оптимизации функции 1 производился со следующими входными данными. В качестве независимых переменных использовался вектор из 100 точек со значениями от 1 до 100. В качестве истинных параметров использовался вектор из 4 точек со значениями 20.0, 10.0, 1.0, 50.0. В качестве стартовых параметров использовались значения 5.0, 2.0, 0.2, 10.0. Для реальных данных использовался датасет из 100 точек, посчитанных при заданных независимых переменных и заданных истинных параметрах со случайным

[2] github.com/margarita0303/kmath/.../TestLmAlgorithm.kt

смещением. В качестве весовой матрицы была взяла одномерная матрица со значением 4.0. В качестве дельты при вычислении численной производной p было взято значение -0.01. В качестве ϵ_1, ϵ_2, ϵ_3, ϵ_4 взяты значения 1e-3, 1e-3, 1e-1, 1e-1.

В результате запуска тестовый пример был завершен на платформе JS за 292(мс), на платформе JVM за 228(мс), на платформе Native за 563(мс). Алгоритм вернул следующие параметры: 20.5317, 9.8296, 0.9976, 50.1745. Значение χ^2 при этом получилось 0.9131

```
fun funcMiddleForLm(t, p, settings): {
    val m = t.shape.component1()
    var y_hat = zeros(m, 1)

    val mt = t.max()
    for(i in 0 until p.shape.component1()){
        y_hat += (t.times(1.0 / mt)).times(p[i, 0])
    }

    for(i in 0 until 5){
        y_hat = funcEasyForLm(y_hat.as2D(), p, settings)
    }

    return y_hat.as2D()
}
```

Листинг 2: Пример более сложной тестовой функции

Запуск оптимизации функции 2 производился со следующими входными данными. В качестве независимых переменных использовался вектор из 100 точек со значениями от 1.0 до 100.0. В качестве истинных параметров использовался вектор из 20 точек со значениями от -24.0 до -5.0. В качестве стартовых параметров использовался вектор истинных параметров со смещением 0.9. Для реальных данных использовался датасет из 100 точек, посчитанных при заданных независимых переменных и заданных истинных параметрах. В качестве весовой матрицы была взяла одномерная матрица со значением 1.0. В качестве дельты при вычислении численной производной p

было взято значение -0.01. В качестве ϵ_1, ϵ_2, ϵ_3, ϵ_4 взяты значения 1e-5, 1e-5, 1e-5, 1e-5.

В результате запуска тестовый пример был завершен на платформе JS за 2(с) 692(мс), на платформе JVM за 1(с) 220(мс), на платформе Native за 7(с) 85(мс). Алгоритм вернул следующие параметры: -23.9717, -18.6686, -21.7971, -20.9681, -22.086, -20.5859, -19.0384, -17.4957, -15.9991, -14.576, -13.2441, -12.0201, -10.9256, -9.9878, -9.2309, -8.6589, -8.2365, -7.8783, -7.4598, -6.8511. Значение χ^2 при этом получилось менее 0.0001.

```
fun funcDifficultForLm(t, p, settings): {
    val m = t.shape.component1()
    var y_hat = zeros(m, 1)

    val mt = t.max()
    for(i in 0 until p.shape.component1()){
        y_hat = y_hat.plus( (t.times(1.0 / mt)).times(p[i, 0]) )
    }

    for(i in 0 until 4){
        y_hat = funcEasyForLm((y_hat + t), p, settings)
    }

    return y_hat.as2D()
}
```

Листинг 3: Пример сложной тестовой функции

Запуск оптимизации функции 3 производился со следующими входными данными. В качестве независимых переменных использовался вектор из 200 точек со значениями от -103.0 до 96.0. В качестве истинных параметров использовался вектор из 15 точек со значениями от -24.0 до -10.0. В качестве стартовых параметров использовался вектор истинных параметров со смещением 0.9. Для реальных данных использовался датасет из 200 точек, посчитанных при заданных независимых переменных и заданных истинных параметрах. В качестве весовой матрицы была взяла одномерная матрица со значением -0.0183. В качестве дельты при вычислении численной производ-

ной p было взято значение -0.01. В качестве ϵ_1, ϵ_2, ϵ_3, ϵ_4 взяты значения 1e-2, 1e-3, 1e-2, 1e-2.

В результате запуска тестовый пример был завершен на платформе JS за 1(мин) 55(с), на платформе JVM за 21(с) 801(мс), на платформе Native за 2(мин) 15(с). Алгоритм вернул следующие параметры: -23.6923, -16.8812, -21.5876, -21.0412, 0.9999, -17.276, -17.276, -17.276, -17.276, -17.276, -17.276, -17.276, -17.276, -17.2759, -17.2763. Значение χ^2 при этом получилось 138.5968. Добиться меньшего значения можно, уменьшая значения ϵ. Например, при значениях ϵ_1, ϵ_2, ϵ_3, ϵ_4, равных 1e-6, 1e-6, 1e-6, 1e-6, значение χ^2 уже достигнет 38.2724.

Запустить тесты с разными начальными значениями, посмотреть на полученные параметры, похожесть исходных данных и полученных при выходных параметрах, а также χ^2 можно в репозитории[3].

[3] github.com/margarita0303/kmath/.../StaticLm

4. Поточная версия

4.1. Реализация

Для реализации поточной версии[4] потребовалось использование асинхронных потоков в Kotlin. Асинхронные потоки в Kotlin представляют мощный механизм для работы с асинхронными операциями. Они позволяют писать асинхронный код, который выглядит последовательным и линейным, делая его более понятным и поддерживаемым. Асинхронный поток полезен для обработки данных в реальном времени, потоковой передачи данных и других ситуаций, когда данные передаются непрерывно и асинхронно. В Kotlin асинхронные потоки реализованы с использованием концепции корутин (coroutines), которая представляет легковесные потоки выполнения. Вместо того, чтобы создавать отдельные потоки для каждой асинхронной операции, корутины позволяют создавать логические потоки выполнения, которые могут быть приостановлены и возобновлены без необходимости блокировать ресурсы [2].

В этой работе представлено создание потока, в котором реальные данные генерируются на каждой итерации цикла с использованием случайного смещения для каждого значения, однако, разумеется, вместо генерации можно настроить получение данных из каких-то других ресурсов. Псевдокод функции создания потока приведен в листинге 6.

```
fun generateNewYDat(y_dat, delta){
    val n = y_dat.shape.component1()
    val y_dat_new = zeros(n, 1)
    for (i in 0 until n) {
        val randomEps = Random.nextDouble(delta + delta) - delta
        y_dat_new[i, 0] = y_dat[i, 0] + randomEps
    }
    return y_dat_new
}
```

Листинг 4: Генерация данных

[4] github.com/margarita0303/kmath/.../StreamingLm

```
fun streamLm(lm_func, startData, launchFrequencyInMs, numberOfLaunches)
: Flow<Matrix> = flow{

    var example_number, p_init, t, y_dat, weight, dp, p_min, p_max, consts, opts =
        = initFromStartData

    var steps = numberOfLaunches
    val isEndless = (steps <= 0)

    while (isEndless || steps > 0) {
        val result = DoubleTensorAlgebra.lm( lm_func, p_init, t, y_dat, weight,
            dp, p_min, p_max, consts, opts, 10, example_number
        )
        emit(result.result_parameters)
        delay(launchFrequencyInMs)
        p_init = result.result_parameters
        y_dat = generateNewYDat(y_dat, 0.1)
        if (!isEndless) steps -= 1
    }
}
```

Листинг 5: Создание потока

Для управления объектами из потока для интерфейса Flow определен ряд функций, одной из которых является функция `collect()`. В качестве параметра она принимает функцию, в которую передает возвращенный функцией `emit()` объект из потока. Таким образом, при вызове `collect()` происходит запуск потока, и программа “подписывается” на получение значений из него, то есть не ждет, когда функция возвратит все значения (а их также может быть бесконечно много), а получает строки по мере их отправки в поток через функцию `emit()`.

```
suspend fun main(){
    val startData = getStartDataForFuncEasy()
    // Создание потока:
    val lmFlow = streamLm(::func, startData, 1000,  10)
    // Запуск потока
    lmFlow.collect { parameters ->
        for (i in 0 until parameters.shape.component1()) {
            val x = (parameters[i, 0] * 10000).roundToInt() / 10000.0
            print("$x ")
            if (i == parameters.shape.component1() - 1) println()
        }
    }
}
```

Листинг 6: Создание и вызов потока в программе

4.2. Тестирование

Полученная реализация показывает хорошие результаты в случае, когда изначальная функция, которую необходимо оптимизировать, достаточно сложная, а потоковые данные меняются не сильно. Тогда каждое следующее вычисление делается гораздо быстрее, чем первое. С практической точки зрения это может быть полезно для отслеживания параметров функции, которая постоянно меняется во времени.

Например, если подставить в потоковую реализацию функцию 3, с такими же входными данными для нее, какие были описаны в предыдущей главе, то программа на JVM выдаст среднее значение подсчета параметров, за исключением первого запуска, равным 194.7755 (мс). Это значение меньше 1 секунды, что существенно меньше, чем 21(с) 801(мс) при одном статическом запуске.

```
suspend fun main(){
    val startData = getStartDataForFuncDifficult()
    // Создание потока:
    val lmFlow = streamLm(::funcDifficultForLm, startData, 0,  100)
    var initialTime = System.currentTimeMillis()
    var lastTime: Long
    val launches = mutableListOf<Long>()
    // Запуск потока
    lmFlow.collect { parameters ->
        lastTime = System.currentTimeMillis()
        launches.add(lastTime - initialTime)
        initialTime = lastTime
        for (i in 0 until parameters.shape.component1()) {
            val x = (parameters[i, 0] * 10000).roundToInt() / 10000.0
            print("$x ")
            if (i == parameters.shape.component1() - 1) println()
        }
    }

    println("Average without first is:
    ${launches.subList(1, launches.size - 1).average()}")
}
```

Листинг 7: Запуск поточной версии на сложной функции

Заключение

В ходе работы были достигнуты следующие результаты:

1. Выполнен обзор предметной области. Подробно рассмотрено устройство алгоритма Левенберга-Марквардта.

2. Реализована статическая версия алгоритма Левенберга-Марквардта на языке Kotlin в рамках проекта KMath. Полученная реализация может быть использована на различных платформах.

3. Проведено тестирование алгоритма на платформах: JS, JVM и Native. Результаты тестирования показали работоспособность полученной реализации. Среди использованных платформ наилучшую производительность показала JVM.

4. Реализована поточная версия алгоритма Левенберга-Маквардта. Проведено тестирование поточной версии на примере сложной функции. Результаты тестирования показали, что в случае, когда потоковые данные меняются не сильно, время подсчета параметров для каждого следующего набора данных значительно сокращается относительно времени, которое требуется на первый подсчет.

   Код реализации открыт и доступен в репозитории[5].

[5] github.com/margarita0303/kmath/tree/dev

Список литературы

[1] Gavin Henri P. The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems // Department of Civil and Environmental Engineering, Duke University. — 2019. — Vol. 19.

[2] Kotlin docs Asynchronous Flow. — `https://kotlinlang.org/docs/flow.html`. — Accessed: 2023.

[3] Marquardt Donald W. An Algorithm for Least-Squares Estimation of Nonlinear Parameters // Journal of the Society for Industrial and Applied Mathematics. — 1963. — Vol. 11, no. 2. — P. 431–441. — https://doi.org/10.1137/0111030.

[4] Nozik Alexander. Kotlin language for science and Kmath library // AIP Conference Proceedings / AIP Publishing LLC. — Vol. 2163. — 2019. — P. 040004.