Оглавление

[Оглавление 1](#_Toc162291616)

[1. Введение 2](#_Toc162291617)

[2. Теоретические основы численного интегрирования 4](#_Toc162291618)

[2.1. Общий подход к численному интегрированию 4](#_Toc162291619)

[2.2. Метод средних прямоугольников 5](#_Toc162291620)

[2.3. Метод трапеций 5](#_Toc162291621)

[2.4. Метод Симпсона 6](#_Toc162291622)

[2.5. Оценка погрешности вычисления определенного интеграла 6](#_Toc162291623)

[3. Основные средства параллельного программирования в языке java 8 7](#_Toc162291624)

[3.1. Интерфейс Runnable 7](#_Toc162291625)

[3.2. Класс Executor 7](#_Toc162291626)

[3.3. Интерфейс Future 8](#_Toc162291627)

[3.4. Проблема совместного доступа к одной переменной (разделение ресурса между потоками) 8](#_Toc162291628)

[4. Разработка программы и анализ результатов работы 10](#_Toc162291629)

[4.1. Определение исходных данных для составления программы 10](#_Toc162291630)

[4.2. Краткое описание работы программы 12](#_Toc162291631)

[4.3. Результат работы программы 13](#_Toc162291632)

[4.4. Сравнение полученных результатов 15](#_Toc162291633)

[Заключение 17](#_Toc162291634)

[Список использованных источников 19](#_Toc162291635)

[Приложение 1. Листинг программ 20](#_Toc162291636)

[Приложение 2. Пример скриншота результатов работы программы (консоль) 27](#_Toc162291637)

# 1. Введение

Благодаря широкому внедрению электронно-вычислительных машин, использование численных методов стало значительно доступнее. Множество сложных технологических процессов требуют точного численного моделирования. Зачастую эти процессы описывается функциональными зависимостями, требующими вычисления определенного интеграла. Среди самых простых таких задач следует указать вычисление площадей, объемов, работы сил и так далее [2]. Известно, что далеко не все функции имеют первообразные. В связи с этим используется два метода вычисления определенных интегралов: метод приближенного интегрирования и методы численного интегрирования [3].

Методы приближенного интегрирования заключаются в разложении подынтегральной функции в ряд Тейлора и, дальнейшего почленного интегрирования членов ряда. Недостатком этого метода является то, что требованием для разложения в ряд является дифференцируемость подынтегральной функции на интервале интегрирования, причем до порядка, который требуется для разложения функции в ряд Тейлора. Таким свойством обладают тоже далеко не все функции.

Методы численного интегрирования лишены требования дифференцируемости подынтегральной функции. Для использования этих методов достаточно выполнения условия непрерывности подынтегральной функции на интервале интегрирования. В связи с этим, методы численного интегрирования имеют более широкую применимость.

В середине прошлого десятилетия на рынок компьютерных комплектующих стали поступать многоядерные процессоры фирм Intel и AMD. Таким образом, технология параллельного или многопоточного выполнения программ стала доступна не только владельцам супер-компьютеров, но пользователям персональных компьютеров. Параллельное выполнение программ позволяет повысить скорость вычислений. Но, при этом не достаточно просто разбить вычислительную задачу на подзадачи и запустить их на выполнение одновременно. При выборе режима параллельного выполнения важно выделить участок работы алгоритма программы, который можно распараллелить так, чтобы потоки выполнения не тормозили друг друга, и, при этом учесть ресурсы (свободные ядра) процессора [7].

Таким образом, целью данной работы является исследование возможностей алгоритмов численного интегрирования для реализации с помощью параллельного программирования. Для выполнения работы требуется исследовать три алгоритма численного интегрирования: метод средних прямоугольников, метод трапеций и метод Симпсона, а также инструменты создания многопоточных программ на языке java. Также требуется спланировать, выполнить и сравнить результаты экспериментов по численному интегрированию в трех режимах: однопоточном, многопоточном с использованием количества параллельных потоков значительно, превышающих количество ядер процессора и многопоточном с количеством потоков равным количеству ядер процессора. Для проверки надежности результатов повторно выполнить эксперименты для двух различных функций.

# 2. Теоретические основы численного интегрирования

## 2.1. Общий подход к численному интегрированию

Обычно понятие определенного интеграла вводится как предел интегральной суммы при не ограниченном увеличении числа точек разбиения отрезка *[a ,b]*. Интегральной суммой называется сумма площадей элементарных прямоугольников, которые получаются в результате разбиения отрезка *[a,b]* на *n* элементарных отрезков и построения боковых сторон прямоугольников, проходящих через узловые точки *xi, i = 0,1, 2 ..n*.

, где .

В результате этой процедуры искомая площадь криволинейной трапеции заменяется площадью ступенчатой фигуры, состоящей из суммы площадей отдельных прямоугольников, интегральной суммой. Площадь этой ступенчатой фигуры при стремится к площади криволинейной трапеции.

Общий способ построения формул численного интегрирования состоит в том, чтобы заменить интеграл конечной суммой простых для вычисления с помощью ЭВМ функций.

В общем виде это можно представить так (квадратурная формула [1]):

где – коэффициент. Это приближенная формула, в общем случае ее погрешность можно оценить как

Для применения метода, отрезок *[a,b]*, на котором требуется найти значение интеграла, разбивают узлами на равные интервалы *a<x1<x2<…xN<b* и представляют интеграл в виде суммы интегралов:

Затем подынтегральная функция а частичном отрезке заменяется некоторым интерполяционным полиномом невысокой степени. Таким образом, достаточно разработать квадратурную формулу для одного интервала, а затем применить ко всем интервалам в рамках отрезка *[a,b]* [1, 2, 3, 5].

При разработке квадратурной формулы принимается во внимание тот факт, что геометрический смысл интеграла – площадь под графиком подынтегральной функции.

## 2.2. Метод средних прямоугольников

В методе средних прямоугольников, для получения реализации квадратурной формулы принимают приближение о том, что площадь подынтегральной функции на интервале между узлами *xi…xi+1*приблизительно ровна площади прямоугольника высотой, равной значению функции *f (xi-0,5)* по середине интервала *xi-1…xi* и шириной *xi+1-xi*. Таким образом, итоговая формула расчета определенного интеграла [1, 2, 3, 5]:

## 2.3. Метод трапеций

В качестве более точного приближения можно заменить площадь подынтегральной функции на интервале между узлами *xi…xi+1* не площадью прямоугольника, а площадью трапеции, основания которой равны значению подынтегральной функции в точках *xi-1* и *xi*, а высота интервалу *xi-xi-1*.

Если узлы на отрезке *[a,b]* размещены равномерно и, если упростить обозначение , и заменить его *fi* , а также представить , то в итоге получим [1, 2, 3, 5]:

## 2.4. Метод Симпсона

Еще более точное приближение, а значит меньшее количество итераций алгоритма вычисления, позволяет получить метод Симпсона. В этом методе подынтегральная функция заменяется на квадратичную параболу вида . Коэффициенты определяются для каждого интервала отдельно, исходя из условия максимально близкой аппроксимации исходной функции параболой. В результате [1, 2, 3, 5] дальнейшего упрощения получаем итоговую формулу:

## 2.5. Оценка погрешности вычисления определенного интеграла

Для оценки погрешности каждого метода используются различные формулы [1, 2, 3, 5]. Для методов численного интегрирования, основанных на использовании прямоугольников, берутся вычисленные значения интегралов при двух разных интервалах , где , то есть в два раза меньше. Результирующая формула имеет вид:

Для оценки погрешности вычисления определенного интеграла методом трапеций используется более сложная формула: , где максимум второй производной на отрезке *[a,b]*.

В оценке погрешности вычисления определенного интеграла методом Симпсона используется *-* максимум четвертой производной на отрезке *[a,b]*. Итоговая формула имеет вид:

# 3. Основные средства параллельного программирования в языке java 8

## 3.1. Интерфейс Runnable

Впервые библиотека классов для параллельного программирования на java была представлена вместе с выходом Java 5 и с тех пор постоянно развивается с каждой новой версией Java [4,6] .

Все современные операционные системы поддерживают параллельное выполнение кода с помощью процессов и потоков. Процесс — это экземпляр программы, который запускается независимо от остальных. Например, запускается программа на Java, ОС создает новый процесс, который работает параллельно с другими. Внутри процессов можно создавать потоки [8].

Прежде чем запустить поток, ему надо предоставить участок кода, который обычно называется «задачей» (task). Это делается через реализацию интерфейса Runnable или расширение класса Thread. Оба эти артефакта требуют реализацию метода run(), не возвращающий ничего. Внутри этого метода реализуется последовательность команд, которые выполняются в отдельном потоке. Для запуска потока достаточно после создания Thread выполнить его метод start();. Кроме перечисленных методов потоки можно приостанавливать, дожидаться его завершения, проверять не закончил ли поток свою работу.

## 3.2. Класс Executor

Современная библиотека Concurrency API не приветствует низкоуровневую работу с потоком по средствам прямого вызова методов Thread. Для этого используются более удобные и безопасные механизмы, одни из которых является ExecutorServic - сервис-исполнитель [4,6]. Исполнители выполняют задачи асинхронно и обычно используют пул потоков. Пул потоков позволяет автоматически производить параллельный запуск потоков, но при этом ограничить количество одновременно работающих потоков. Например, если в пул отправлено 20 потоков, но в настройках пула указано что одновременно могут работать только 5, то пул запускает первые 5 потоков, а затем, как только один поток закончил работу, пул запускает вместо него следующий из 15 оставшихся и так до тех пор, пока в пуле будут неотработавшие потоки. При этом в распоряжении программиста есть сервисы - исполнители с одним доступным потоком в пуле, а также настраиваемой размерностью пула.

## 3.3. Интерфейс Future

Кроме Runnable, сервисы- исполнители могут принимать другой вид задач, который называется Callable. Callable — это также интерфейс, но, в отличие от Runnable, он может возвращать значение. Callable-задачи также могут быть переданы сервисам-исполнителям [4,6]. С получением результата работы потока есть проблема, заключающаяся в том что мы не знаем когда поток выдаст результат. Поскольку метод submit() не ждет завершения задачи, исполнитель не может вернуть результат задачи сразу. Вместо этого исполнитель возвращает специальный объект Future, у которого можно позже запросить результат задачи. Объект Future представляет результат работы потока, но заполнен значением он будет только после того, как поток выполнится.

Сервисы - исполнители могут принимать список задач на выполнение из пула задач с помощью метода invokeAll(), который принимает коллекцию callable-задач и возвращает список из Future. Этот метод удобен тем, что позволяет дождаться выполнения всех потоков из пула, что очень удобно для целей контроля времени выполнения потоков в пуле.

## 3.4. Проблема совместного доступа к одной переменной (разделение ресурса между потоками)

Важной проблемой работы с потоками, является то, что потоки могут обращается как для чтения, так и для записи к одной и той же переменной. Но мы не можем контролировать время работы потока, и поэтому потоки могут обращаться к одной и той же переменной в одно и тоже время. В итоге один поток может успеть изменить значение переменной, тогда как другой будет оперировать старым значением этой переменной [4,6].

Для предотвращения такой ситуации используется синхронизация потоков. Существует несколько способов синхронизации, как на чтение, так и на запись, но все они основаны на блокировках. Блокировка представляет собой процесс, когда первый поток захватывает переменную и блокирует для нее доступ другим потокам. Другие потоки также по очереди смогут обратиться к переменной только тогда, когда первый поток закончит работать с переменной и снимет блокировку.

Блокировки работают для методов, объектов и переменных простых типов. Соответственно, чем меньше заблокированный участок кода или объект, тем выгоднее блокировка, так как потоки, отличные от захватившего такой объект потока, будут меньше простаивать [4,6].

В работе используется самый простой тип блокировки на уровне методов, вызываемый с помощью служебного слова synchronized. В результате в один и тот же метод, в одно и то же время может заходить один поток.

# 4. Разработка программы и анализ результатов работы

4.1. Определение исходных данных для составления программы

(2 функции, количество итераций и методика расчета времени работы алгоритма)

Программа вычисляет значения определенных интегралов двух функций:

* (Рисунок 1.)
* (Рисунок 2.)

На отрезке *x* [0..40]. Для наглядной загрузки процессора количество интервалов разбиения *N* равно 100000.

В качестве алгоритмов численного интегрирования в программе исследуются 3 алгоритма: метод Симпсона, метод средних прямоугольников и метод трапеций.

Для распараллеливания работы алгоритма по вычислению определенного интеграла используется следующая методика:

1. Выбирается количество одновременных потоков: для эксперимента выбрано 3 значения: 1 поток, 50 потоков и количество потоков равно количеству ядер процессора (определяется автоматически).
2. Участок интегрирования [0..40] по *x* разбивается поровну на количество потоков. Потоки добавляются в пул потоков и готовы к выполнению.
3. Фиксируется время запуска пула потоков в работу (начала эксперимента). Пул потоков запускается на выполнение. По завершению пула потоков фиксируется время окончания эксперимента. Вычисляется разность времени окончания и начала эксперимента в миллисекундах.

Таким образом, для каждой функции эксперимент выполняется по 9 раз: сначала функция вычисляется методом Симпсона тремя способами распараллеливания, затем методом средних прямоугольников тремя способами распараллеливания и методом трапеций тоже тремя методами распараллеливания.

Кроме того, как показывает практика, java машина не сразу входит в рабочий режим, поэтому есть необходимость многократного запуска каждого эксперимента.

Поэтому каждый эксперимент запускается по 10 раз и для каждой такой серии экспериментов (с одинаковым численным методом интегрирования и режимом распараллеливания) вычисляется среднее время эксперимента в миллисекундах.

Также, в ходе каждой такой серии из десяти экспериментов фиксируется не только время выполнения эксперимента, но и полученное значение интеграла и погрешность метода. В связи с этим, для метода средних прямоугольников запускается дополнительный эксперимент с удвоенным значением количества интервалов разбиения *N*=200000.

Для метода Симпсона, так как h=0,00001, можно полагать, что погрешность не будет значительно отличаться от h4.

Для метода трапеций, при таком малом значении h, тоже не имеет смысла вычислять максимум второй производной, достаточно вычислить значение , что производится программой.

Рисунок 1. График функции 1

Рисунок 2. График функции 2

## 4.2. Краткое описание работы программы

Программа имеет консольный вид, не использует диалог с пользователем. Программа запускается, выполняет программу экспериментов, выводит результат работы в консоль и прекращает работу.

В главном классе программы CalculatingNumericalIntegration определяется программа экспериментов. Каждый эксперимент Result получает исходные данные: a,b,N, а также фабрику функций, фабрику вычислительных потоков и количество одновременных потоков. В ходе эксперимента, заполняются поля объекта класса Result, предназначенные для фиксации результатов эксперимента: среднее значение времени выполнения, значение интеграла и погрешность метода. Все эксперименты сохраняются в коллекцию ArrayList.

Фабрики функций и фабрики вычислительных потоков используются для исключения необходимости синхронизации потоков и внесении в эксперимент искажений на время простоя потоков из-за блокировки ресурсов. Так для каждого эксперимента в серии создаётся свой вычислительный поток, а в каждой итерации вычислений создается свой экземпляр класса функции.

Для проведения эксперимента, объект класса Result с заполненными полями исходных данных извлекается из коллекции экспериментов и запускается на выполнение методом startExperiment(Result result) класса CalcOrganizer. Именно в объекте этого класса создается пул вычислительных потоков с настройками, указанными в объекте Result, запускается на выполнение и получается результат эксперимента.

## 4.3. Результат работы программы

Результат работы программы скопирован из консоли и приведен ниже.

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод Симпсона' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 1 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292392591775, Погрешность вычислений E= 2.5600000000000006E-14, Среднее время выполнения T=16

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод Симпсона' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 50 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292392591415, Погрешность вычислений E= 2.5600000000000006E-14, Среднее время выполнения T=10

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод Симпсона' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292392591523, Погрешность вычислений E= 2.5600000000000006E-14, Среднее время выполнения T=10

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 1 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292237499714, Погрешность вычислений E= 0.0, Среднее время выполнения T=11

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 50 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.1626429223749987, Погрешность вычислений E= 0.0, Среднее время выполнения T=13

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292237499817, Погрешность вычислений E= 0.0, Среднее время выполнения T=7

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 200000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292353818383, Погрешность вычислений E= 0.0, Среднее время выполнения T=16

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод трапеций' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 1 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292702774033, Погрешность вычислений E= 8.533333333333334E-4, Среднее время выполнения T=12

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод трапеций' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 50 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.16264292702774386, Погрешность вычислений E= 8.533333333333334E-4, Среднее время выполнения T=16

Результат вычисления интеграла функции (cos(x)/(x+2) методом 'метод трапеций' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=0.1626429270277441, Погрешность вычислений E= 8.533333333333334E-4, Среднее время выполнения T=13

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод Симпсона' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 1 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.5898657154944682E8, Погрешность вычислений E= 2.5600000000000006E-14, Среднее время выполнения T=14

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод Симпсона' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 50 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.5898657154944584E8, Погрешность вычислений E= 2.5600000000000006E-14, Среднее время выполнения T=13

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод Симпсона' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.5898657154944608E8, Погрешность вычислений E= 2.5600000000000006E-14, Среднее время выполнения T=8

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 1 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.589865715290475E8, Погрешность вычислений E= 0.0, Среднее время выполнения T=11

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 50 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.589865715290469E8, Погрешность вычислений E= 0.0, Среднее время выполнения T=12

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.5898657152904746E8, Погрешность вычислений E= 0.0, Среднее время выполнения T=6

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод средних прямоугольников' на отрезке [0.0,40.0] при использовании 200000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.5898657154434648E8, Погрешность вычислений E= 0.0, Среднее время выполнения T=10

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод трапеций' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 1 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.589865715902424E8, Погрешность вычислений E= 8.533333333333334E-4, Среднее время выполнения T=12

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод трапеций' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 50 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.5898657159024322E8, Погрешность вычислений E= 8.533333333333334E-4, Среднее время выполнения T=10

Результат вычисления интеграла функции x\*x/(x+12)^(-2.5) методом 'метод трапеций' на отрезке [0.0,40.0] при использовании 100000 интервалов. Работа выполнялась в 4 потоках, при количестве ядер процессора 4. Результат следующий:Значение интеграла I=2.58986571590244E8, Погрешность вычислений E= 8.533333333333334E-4, Среднее время выполнения T=6

Для анализа данных экспериментов удобнее представить их в виде таблиц для каждой функции (Таблица 1 и Таблица 2).

Таблица 1. Результаты экспериментов для функции №1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | |
| Количество потоков | Метод Симпсона | | | Метод средних прямоугольников | | | Метод трапеций | | |
| I | E | T, мс | I | E | T, мс | I | E | T, мс |
| 1 | 0.16264292392591775 | 2.566E-14 | 16 | 0.16264292237499714 | - | 11 | 0.16264292702774033 | 8.53E-4 | 12 |
| 50 | 0.16264292392591415 | 2.566E-14 | 10 | 0.1626429223749987 | - | 13 | 0.16264292702774386 | 8.53E-4 | 16 |
| Равно количеству ядер процессора | 0.16264292392591523 | 2.566E-14 | 10 | 0.16264292237499817 | (0.16264292353818383 -0.16264292237499817)/3=3,8772855E-10 | 7 | 0.1626429270277441 | 8.53E-4 | 13 |

Таблица 2. Результаты экспериментов для функции №2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | |
| Количество потоков | Метод Симпсона | | | Метод средних прямоугольников | | | Метод трапеций | | |
| I | E | T, мс | I | E | T, мс | I | E | T, мс |
| 1 | 2.5898657154944682E8 | 2.56E-14 | 14 | 2.589865715290475E8 | - | 11 | 2.589865715902424E8 | 8.53E-4 | 12 |
| 50 | 2.5898657154944584E8 | 2.56E-14 | 13 | 2.589865715290469E8 | - | 12 | 2.5898657159024322E8 | 8.53E-4 | 10 |
| Равно количеству ядер процессора | 2.5898657154944608E8 | 2.56E-14 | 8 | 2.5898657152904746E8 | (2.5898657154434648E8-2.5898657154944608E8)/3=1,69986Е-2 | 6 | 2.58986571590244E8 | 8.53E-4 | 6 |

Для вычисления погрешности метода средних прямоугольников воспользуемся формулой и вычисленным, в ходе серии экспериментов, значением определенного интеграла при N=200000.

## 4.4. Сравнение полученных результатов

Анализируя результаты экспериментов, приведенные в таблицах 1 и 2 можно сделать следующие выводы:

1. Во всех методах расчета для каждой функции получены близкие с высокой точностью значения определенных интегралов, что говорит о том, что для вычисления использовались корректно написанные программы;
2. Наибольшая точность метода достигнута при использовании метода Симпсона, худшая точность - метода при использовании метода средних прямоугольников;
3. Сравнивая однопоточную реализацию и многопоточную реализацию методов численного интегрирования можно заключить, что многопоточные алгоритмы без использования информации о количестве ядер процессора и значительно превышающие это количество, работают даже дольше чем, если бы такая же работа запускалась в одном потоке. Это можно объяснить тем, что на создание пула потоков тоже затрачиваются ресурсы процессора и значительное количество памяти.
4. Напротив, при учете количества свободных ядер процессора, распараллеливание вычислительных задач позволяет значительно ускорить вычисление, в некоторых из приведенных экспериментов – до 2-ух раз быстрее.
5. Метод Симпсона имеет время выполнения сравнимое по величине с другими приведенными, но при равных исходных данных экспериментов его погрешность метода значительно меньше. То есть, для сравнимых значений погрешности методов необходимо уменьшить количество интервалов разбиения в несколько раз. При этом увеличится и скорость вычисления определенного интеграла.

# Заключение

В курсовой работе реализованы три метода численного интегрирования: метод Симпсона, метод средних прямоугольников, метод трапеций. Все три метода имеют возможность приведения к параллельным вычислениям по средствам разбиения интервала интегрирования на участки и вычисления определенного интеграла на каждом участке в отдельном потоке.

Для сравнения скорости работы проводились эксперименты по вычислению определенных интегралов двух функций с использованием однопоточной реализации вычислений, многопоточной реализации с использованием количества потоков значительно большего, чем количество ядер процессора, а также многопоточной реализации с количеством потоков, равных количеству ядер процессора.

Наиболее выгодным алгоритмом с точки зрения точности и скорости выполнения является метод Симпсона.

При использовании параллельного программирования важно учитывать ресурсы процессора, а именно количество ядер, на которых возможно одновременно выполнить вычислительные потоки.

# Список использованных источников

1. Блюмин А.Г.Численные методы вычисления интегралов и решения задач для обыкновенных дифференциальных уравнений: Методические указания к выполнению лабораторных работ по курсу «Численные методы». — М.: МГТУ им. Н.Э.Баумана, 2008. — 74 с.
2. Мышенков В.И. Численные методы. Часть первая: Учебное пособие для студентов специальности 0101.07. – М.:МГУЛ,2001. – 120 с.: ил.
3. Формалев В. Ф. Численные методы. – М.: ФИЗМАТЛИТ, 2004. – 400 с.
4. Хорстманн Кей С. Java SE 8. Вводный курс М.: Вильямс, 2014. — 208 c.
5. [Черкасов, М.А.](http://www.libex.ru/?cat_author=%D7%E5%F0%EA%E0%F1%EE%E2,%20%CC.%C0.&author_key=215) Численные методы. Решение задач Издательство: М.: МАИ92 страниц; 2007 г.
6. Шилдт Г. Java. Полное руководство, 8-еизд.: Пер. с англ.—М .:ООО “И.Д. Вильямс”, 2012.—1104с .: ил.—Парал.тит. англ.
7. <https://software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction> (дата обращения 23.03.2018).
8. <https://tproger.ru/translations/java8-concurrency-tutorial-1/>(дата обращения 23.03.2018)

# Приложение 1. Листинг программ

package calculating\_simpson\_thread2;

import java.util.ArrayList;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class CalcOrganizer {

Double S=0.0;

long startTime, stopTime;

int AverageTimeCalc=0;

public static final int NUMBERofEXPERIMENTS=10;

ArrayList <Result>arrayOfResult;

int i\_result=-1;

public CalcOrganizer(ArrayList <Result>arrayOfResult) {

this.arrayOfResult=arrayOfResult;

goNext();

}

public void startExperiment(Result result)

{

AverageTimeCalc=0;

for (int experiment\_number=1; experiment\_number<CalcOrganizer.NUMBERofEXPERIMENTS+1; experiment\_number++)

{

System.out.println("Приближенное вычисление интеграла функции "+result.getFactoryFunction().Text()+" на отрезке ["+result.geta()+".."+result.getb()+"] методом '"+result.getThreadFactory().Text()+"'. Эксперимент №"+experiment\_number);

ExecutorService executor=null;

ArrayList<Callable<Integer>> tasks = new ArrayList<>();

switch (result.getThreadType()) {

case CalculatingNumericalIntegration.SINGLE\_THREAD: {executor=Executors.newSingleThreadExecutor();

tasks.add(result.getThreadFactory().getThread(result.geta(),result.getb(),result.getN(),this,result.getFactoryFunction()));

break;}

case CalculatingNumericalIntegration.ANY\_THREAD:{executor=Executors.newFixedThreadPool(50);

int nN=(int) Math.round(result.getN()/50);

double d=(result.getb()-result.geta())/50;

for (int j=0;j<50; j++ ) { double aA=result.geta()+j\*d; double bB=result.geta()+(j+1)\*d; // System.out.println("nN="+nN+" d="+d+" aA="+aA+" bB="+bB); tasks.add(result.getThreadFactory().getThread(aA,bB,nN,this,result.getFactoryFunction()));

}

break;}

default:{

int availableProcessors=CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD;

executor=Executors.newFixedThreadPool(availableProcessors);

int nN=(int) Math.round(result.getN()/availableProcessors);

double d=(result.getb()-result.geta())/availableProcessors;

for (int j=0;j<availableProcessors; j++ )

{

double aA=result.geta()+j\*d;

double bB=result.geta()+(j+1)\*d;

//System.out.println("nN="+nN+" d="+d+" aA="+aA+" bB="+bB);

tasks.add(result.getThreadFactory().getThread(aA,bB,nN,this,result.getFactoryFunction()));

}

break;}

}

S=0.0;

startTime=System.currentTimeMillis();

// старт

//ожидание завершения потоков

try {executor.invokeAll(tasks);} catch (InterruptedException ex) {System.out.println("Ошибка выполнения потоков:"+ex.getMessage());}

// все потоки закончены

stopTime=System.currentTimeMillis();

executor.shutdownNow();

System.out.println("В эксперименте №"+experiment\_number+" получено значение "+S+" за время "+(stopTime-startTime)+" (миллисекунд)");

AverageTimeCalc=AverageTimeCalc+(int)(stopTime-startTime);

}

result.setResult(S, result.getThreadFactory().ErrorCalc(result), AverageTimeCalc/CalcOrganizer.NUMBERofEXPERIMENTS);

goNext();

}

public synchronized void Notification(Double ds) {S=S+ds;}

public void goNext()

{

i\_result++;

if (i\_result<arrayOfResult.size()) {startExperiment(arrayOfResult.get(i\_result));}

else {arrayOfResult.stream().forEach(exp->System.out.println(exp.toString()));}

}

}

package calculating\_simpson\_thread2;

import java.util.concurrent.Callable;

public interface CalcThread extends Callable <Integer> {

public Integer call();

}

package calculating\_simpson\_thread2;

import java.util.ArrayList;

public class CalculatingNumericalIntegration {

public static final int SINGLE\_THREAD=1;

public static final int ANY\_THREAD=50;

public static final int AVAILABLE\_PROCESSOR\_THREAD= Runtime.getRuntime().availableProcessors();

static ArrayList <Result>arrayOfResult=new ArrayList<>();

static CalcOrganizer calcOrganizer;

public static void main(String[] args) {

int N=100000;

final double a=0;

final double b=40;

FactoryFunction factoryFunction1=new FactoryF1();

ThreadFactory threadFactory1=new SimpsonThreadFactory();

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.SINGLE\_THREAD,threadFactory1,factoryFunction1));

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.ANY\_THREAD,threadFactory1,factoryFunction1));

arrayOfResult.add(new Result(N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory1,factoryFunction1));

ThreadFactory threadFactory2=new MediumRectThreadFactory();

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.SINGLE\_THREAD,threadFactory2,factoryFunction1));

arrayOfResult.add(new Result(N,a,b,

arrayOfResult.add(new Result(N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory2,factoryFunction1));

arrayOfResult.add(new Result(2\*N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory2,factoryFunction1)); // для оценки погрешности

ThreadFactory threadFactory3=new TrapezeThreadFactory();

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.SINGLE\_THREAD,threadFactory3,factoryFunction1));

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.ANY\_THREAD,threadFactory3,factoryFunction1));

arrayOfResult.add(new Result(N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory3,factoryFunction1));

FactoryFunction factoryFunction2=new FactoryF2();

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.SINGLE\_THREAD,threadFactory1,factoryFunction2));

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.ANY\_THREAD,threadFactory1,factoryFunction2));

arrayOfResult.add(new Result(N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory1,factoryFunction2));

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.SINGLE\_THREAD,threadFactory2,factoryFunction2));

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.ANY\_THREAD,threadFactory2,factoryFunction2));

arrayOfResult.add(new Result(N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory2,factoryFunction2));

arrayOfResult.add(new Result(2\*N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory2,factoryFunction2)); // для оценки погрешности

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.SINGLE\_THREAD,threadFactory3,factoryFunction2));

arrayOfResult.add(new Result(N,a,b,CalculatingNumericalIntegration.ANY\_THREAD,threadFactory3,factoryFunction2));

arrayOfResult.add(new Result(N,a,b, CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD,threadFactory3,factoryFunction2));

calcOrganizer = new CalcOrganizer(arrayOfResult);

}

}

package calculating\_simpson\_thread2;

public class FactoryF1 implements FactoryFunction {

@Override

public Function getF() {

return new Function1();

}

@Override

public String Text() {

return "(cos(x)/(x+2)";

}

}package calculating\_simpson\_thread2;

public class FactoryF2 implements FactoryFunction {

@Override

public Function getF() {

return new Function2();

}

@Override

public String Text() {

return "x\*x/(x+12)^(-2.5)";

}

}

package calculating\_simpson\_thread2;

public interface FactoryFunction {

public Function getF();

public String Text();

}package calculating\_simpson\_thread2;

public interface Function {

public double getFunction(double x);

}

package calculating\_simpson\_thread2;

public class Function1 implements Function {

public double getFunction(double x) { return (Math.cos(x))/(x+2);}

}

package calculating\_simpson\_thread2;

public class Function2 implements Function{

@Override

public double getFunction(double x) {

{ return (x\*x)/(Math.pow((x+12),-2.5));}

}

}

package calculating\_simpson\_thread2;

public class MediumRectCalcThread implements CalcThread {

int N;

double a,b,x;

CalcOrganizer mainThread;

double S;

FactoryFunction ff;

public MediumRectCalcThread(double a, double b, int n, CalcOrganizer calcOrganizer, FactoryFunction factoryFunction) {

this.a=a;

this.b=b;

this.N=n;

this.mainThread=calcOrganizer;

this.ff=factoryFunction;

}

@Override

public Integer call() {

double h=(b-a)/N;

S=0;

for(int i=0;i<N;i++)

{

x=a+i\*h+h/2;

S=S+ff.getF().getFunction(x)\*h;

}

this.mainThread.Notification(S);

return -1;

}

}

package calculating\_simpson\_thread2;

public class MediumRectThreadFactory implements ThreadFactory{

@Override

public CalcThread getThread(double a, double b , int n, CalcOrganizer calcOrganizer, FactoryFunction factoryFunction) {

return new MediumRectCalcThread(a,b,n,calcOrganizer,factoryFunction);

}

@Override

public String Text() {

return "метод средних прямоугольников";

}

@Override

public double ErrorCalc(Result r) {

return 0;

}

}

package calculating\_simpson\_thread2;

public class Result {

private int N;

private double a;

private double b;

private int ThreadType;

private ThreadFactory threadFactory;

private FactoryFunction factoryFunction;

private double Sum;

private double CalcError;

private int AverageTimeCalc;

Result(int N, double a, double b, int threadType, ThreadFactory threadFactory , FactoryFunction factoryFunction) {

this.N=N;

this.a=a;

this.b=b;

this.ThreadType=threadType;

this.threadFactory=threadFactory;

this.factoryFunction=factoryFunction;

}

public void setN( int N) {this.N=N;}

public void seta( double a) {this.a=a;}

public void setb( double b) {this.b=b;}

public void setThreadType( int ThreadType) {this.ThreadType=ThreadType;}

public void setThreadFactory( ThreadFactory threadFactory) {this.threadFactory=threadFactory;}

public void setFactoryFunction(FactoryFunction factoryFunction) {this.factoryFunction=factoryFunction; }

public void setSum( double Sum) {this.Sum=Sum;}

public void setCalcError( double CalcError) {this.CalcError=CalcError;}

public void setAverageTimeCalc( int AverageTimeCalc) {this.AverageTimeCalc=AverageTimeCalc;}

public int getN() {return this.N;}

public double geta() {return this.a;}

public double getb() {return this.b;}

public int getThreadType() {return this.ThreadType;}

public ThreadFactory getThreadFactory( ) {return this.threadFactory;}

public FactoryFunction getFactoryFunction() {return this.factoryFunction;}

public double getSum() {return this.Sum;}

public double getCalcError() {return this.CalcError;}

public int getAverageTimeCalc() {return this.AverageTimeCalc;}

public void setResult(double Sum, double CalcError,int AverageTimeCalc)

{

this.setSum(Sum);

this.setCalcError(CalcError);

this.setAverageTimeCalc(AverageTimeCalc);

}

public String toString ()

{

return "Результат вычисления интеграла функции "+this.factoryFunction.Text()+" методом '"+this.threadFactory.Text()+"' на отрезке ["+this.geta()+","+this.getb()+"] при использовании "+this.getN()+" интервалов. "+

"Работа выполнялась в "+this.getThreadType()+" потоках, при количестве ядер процессора "+CalculatingNumericalIntegration.AVAILABLE\_PROCESSOR\_THREAD+

". Результат следующий:Значение интеграла I="+this.getSum()+", Погрешность вычислений E= "+this.getCalcError()+", Среднее время выполнения T="+this.getAverageTimeCalc();

}

}

package calculating\_simpson\_thread2;

public class SimpsonCalcThread implements CalcThread{

int N;

double a,b,x;

CalcOrganizer mainThread;

double S;

FactoryFunction ff;

public SimpsonCalcThread(double a,double b, int N, CalcOrganizer mainThread, FactoryFunction ff) {

this.a=a;

this.b=b;

this.N=N;

this.mainThread=mainThread;

this.ff=ff;

}

@Override

public Integer call() {

double h=(b-a)/N;

S=(ff.getF().getFunction(a)+ff.getF().getFunction(b))\*(h/3);

for(int i=1;i<N;i++)

{

x=a+i\*h;

if (i % 2!=0) {S=S+ff.getF().getFunction(x)\*(h/3)\*4;}

else {S=S+ff.getF().getFunction(x)\*(h/3)\*2;}

}

this.mainThread.Notification(S);

return -1;

}

}

package calculating\_simpson\_thread2;

public class SimpsonThreadFactory implements ThreadFactory{

@Override

public CalcThread getThread(double a, double b , int n, CalcOrganizer calcOrganizer, FactoryFunction factoryFunction) {

return new SimpsonCalcThread(a,b,n,calcOrganizer,factoryFunction);

}

@Override

public String Text() {

return "метод Симпсона";

}

@Override

public double ErrorCalc(Result r) {

double h=(r.getb()-r.geta())/r.getN();

return Math.pow(h,4);

*}*

}

package calculating\_simpson\_thread2;

public interface ThreadFactory {

public CalcThread getThread(double a, double b , int n, CalcOrganizer calcOrganizer ,FactoryFunction factoryFunction);

public String Text();

public double ErrorCalc(Result r);

}

package calculating\_simpson\_thread2;

public class TrapezeCalcThread implements CalcThread {

int N;

double a,b,x;

CalcOrganizer mainThread;

double S;

FactoryFunction ff;

public TrapezeCalcThread(double a, double b, int n, CalcOrganizer calcOrganizer, FactoryFunction factoryFunction) {

this.a=a;

this.b=b;

this.N=n;

this.mainThread=calcOrganizer;

this.ff=factoryFunction;

}

@Override

public Integer call() {

double h=(b-a)/N;

S=h\*(ff.getF().getFunction(a)+ff.getF().getFunction(b))/2;

for(int i=1;i<N;i++)

{

x=a+i\*h;

S=S+ff.getF().getFunction(x)\*h;

}

this.mainThread.Notification(S);

return -1;

}

}

package calculating\_simpson\_thread2;

public class TrapezeThreadFactory implements ThreadFactory{

@Override

public CalcThread getThread(double a, double b , int n, CalcOrganizer calcOrganizer, FactoryFunction factoryFunction) {

return new TrapezeCalcThread(a,b,n,calcOrganizer,factoryFunction);

}

@Override

public String Text() {

return "метод трапеций";

}

@Override

public double ErrorCalc(Result r) {

double h=(r.getb()-r.geta())/r.getN();

return Math.pow((r.getb()-r.geta()),3)\*h\*h/12;

}

}

# Приложение 2. Пример скриншота результатов работы программы (консоль)

