

Отчёт

по лабораторной работе №6

Выполнил:

студент группы

6301-030301D

Панин Ю. В.

Проверил:

Борисов Д. С.

Содержание

Содержание	2
Задание 1	3
Задание 2	7
Задание 3	10
Задание 4	15

Задание 1

В класс Functions был добавлен метод, реализующий нахождение интеграла функции методом трапеций на заданном отрезке с заданным шагом. Далее был преобразован класс, и запущена программа на выполнение. Вывод программы согласуется с поставленными в работе задачами:

```
// 1. Метод интегрирования по методу трапеций
public static double integrate(Function f, double left, double right, double step) {
    if (left < f.getLeftDomainBorder() || right > f.getRightDomainBorder()) {
        throw new IllegalArgumentException("Интервал интегрирования выходит за границы области определения функции");
    }
    if (step <= 0) {
        throw new IllegalArgumentException("Шаг интегрирования должен быть положительным");
    }

    double integral = 0.0;
    double x = left;

    while (x < right) {
        double nextX = Math.min(x + step, right);
        double y1 = f.getFunctionValue(x);
        double y2 = f.getFunctionValue(nextX);

        // Проверка на NaN
        if (Double.isNaN(y1) || Double.isNaN(y2)) {
            return Double.NaN;
        }

        integral += (y1 + y2) * (nextX - x) / 2.0;
        x = nextX;
    }

    return integral;
}
```

```
// 2. Метод для определения оптимального шага (для тестирования)
public static double findOptimalStep(Function f, double left, double right, double targetError) {
    double theoretical = Math.exp(1) - 1; // Для exp:  $\int_0^1 e^x dx = e - 1$ 
    double step = 0.1;
    double error = Double.MAX_VALUE;

    while (error > targetError) {
        double numerical = integrate(f, left, right, step);
        error = Math.abs(theoretical - numerical);
        step /= 2;
        if (step < 1e-10) break;
    }

    return step * 2; // Возвращаем последний успешный шаг
}
```

```
PS C:\Users\yura-\Lab-6-2025> cd src
PS C:\Users\yura-\Lab-6-2025\src> javac functions/meta/*.java
PS C:\Users\yura-\Lab-6-2025\src> javac functions/basic/*.java
PS C:\Users\yura-\Lab-6-2025\src> javac functions/*.java
PS C:\Users\yura-\Lab-6-2025\src> javac Main.java
PS C:\Users\yura-\Lab-6-2025\src> java Main
```

=== Тестирование задания 1: Метод интегрирования ===

Функция: $f(x) = e^x$

Интеграл от 0 до 1

Теоретическое значение: $\text{Int}(e^x dx)$ на отрезке $(0, 1) = e - 1 = 1,7182818285$

1. Интегрирование с разными шагами:

=====

Шаг	Результат	Ошибка	Относительная ошибка
1,0000	1,8591409142	0,1408590858	0,0819767069
0,5000	1,7539310925	0,0356492640	0,0207470413
0,1000	1,7197134914	0,0014316629	0,0008331945
0,0500	1,7186397889	0,0003579605	0,0002083247
0,0100	1,7182961475	0,0000143190	0,0000083333
0,0050	1,7182854082	0,0000035798	0,0000020833
0,0010	1,7182819716	0,0000001432	0,0000000833
0,0005	1,7182818643	0,0000000358	0,0000000208
0,0001	1,7182818299	0,0000000014	0,0000000008

2. Поиск оптимального шага:

=====

Целевая точность: 0,0000001000

Процесс поиска:

Итерация	Шаг	Результат	Ошибка
0	0,100000000	1,7197134914	0,0014316629
1	0,050000000	1,7186397889	0,0003579605
2	0,025000000	1,7183713214	0,0000894929
3	0,012500000	1,7183042019	0,0000223734
4	0,006250000	1,7182874218	0,0000055934
5	0,003125000	1,7182832268	0,0000013983
6	0,001562500	1,7182821780	0,0000003496

!Достигнута точность 0,0000001000 при шаге 0,0007812500

3. Проверка граничных случаев:

=====

3.1. Интегрирование за пределами области определения (левая граница):

Результат: 2,7205012481

3.2. Интегрирование за пределами области определения (правая граница):

Результат: Infinity

3.3. Некорректный шаг интегрирования:

!Поймано исключение: Шаг интегрирования должен быть положительным

3.4. Шаг интегрирования = 0:

!Поймано исключение: Шаг интегрирования должен быть положительным

3.5. Левая граница = правой границе:

Результат: 0,0000000000

4. Интегрирование других функций:

=====

4.1. $\text{Int}(\sin(x))dx$ на отрезке (0,1):

Теоретическое: 0,4596976941

Численное: 0,4596976558

Ошибка: 0,0000000383

4.2. $\text{Int}(\cos(x)dx)$ на отрезке (0, 1):

Теоретическое: 0,8414709848

Численное: 0,8414709147

Ошибка: 0,0000000701

4.3. $\text{Int}(x^2dx)$ на отрезке (0, 1) (проверка точности метода):

Шаг 0,5000: результат = 0,3750000000, ошибка = 0,0416666667

Шаг 0,1000: результат = 0,3350000000, ошибка = 0,0016666667

Шаг 0,0100: результат = 0,3333500000, ошибка = 0,0000166667

Шаг 0,0010: результат = 0,3333335000, ошибка = 0,0000001667

```

5. Проверка на нецелое количество шагов:
=====
5.1. Область интегрирования 0.7, шаг 0.3:
    Количество шагов: (0.7 - 0) / 0.3 = 2.333...
    Результат: 1,0200690377

5.2. Область интегрирования 1.0, шаг 0.7:
    Количество шагов: (1.0 - 0) / 0.7 = 1.428...
    Результат: 1,7646186280

```

```

6. Автоматический подбор шага для точности 1e-7:
=====
Метод половинного деления:
| Шаг          | Результат      | Ошибка          |
|-----|-----|-----|
| 0,10000000 | 1,7197134914 | 0,0014316629 |
| 0,05000000 | 1,7186397889 | 0,0003579605 |
| 0,02500000 | 1,7183713214 | 0,0000894929 |
| 0,01250000 | 1,7183042019 | 0,0000223734 |
| 0,00625000 | 1,7182874218 | 0,0000055934 |
| 0,00312500 | 1,7182832268 | 0,0000013983 |
| 0,00156250 | 1,7182821780 | 0,0000003496 |

Итоговый шаг: 0,0015625000
Ошибка: 0,0000000874
!Требуемая точность достигнута!

```

```

7. Проверка производительности:
=====
Время выполнения 100000 интегрирований: 1995 мс
Среднее время одного интегрирования: 0,019950 мс

```

Задание 2

Была создана директория threads, в которой будут описываться методы, связанные с потоками. Далее в пакете threads был описан класс Task, объект которого хранит ссылку на объект интегрируемой функции, границы области интегрирования, шаг дискретизации, а также целочисленное поле, хранящее количество выполняемых заданий.

Затем в главном классе программы описан метод `nonThread()`, реализующий последовательную (без применения потоков инструкций) версию программы. Данный метод выполняет действия согласно заданию. Программа была запущена на выполнение, её вывод согласуется с поставленными задачами:

```
package threads;

import functions.Function;

public class Task {
    private Function function;
    private double left;
    private double right;
    private double step;
    private int tasksCount;
    private volatile boolean isReady = false;
    private volatile boolean isCompleted = false;

    // Конструктор
    public Task(int tasksCount) {
        this.tasksCount = tasksCount;
    }

    // Геттеры и сеттеры
    public synchronized void setTask(Function function, double left, double right, double step) {
        this.function = function;
        this.left = left;
        this.right = right;
        this.step = step;
        isReady = true;
        isCompleted = false;
    }

    public synchronized void getTask() {
        isReady = false;
    }
}
```



```

public Function getFunction() {
    return function;
}

public double getLeft() { no usages
    return left;
}

public double getRight() { no usages
    return right;
}

public double getStep() { no usages
    return step;
}

public int getTasksCount() { no usages
    return tasksCount;
}

public boolean isReady() { no usages
    return isReady;
}

public boolean isCompleted() { no usages
    return isCompleted;
}

public void setCompleted(boolean completed) {
    isCompleted = completed;
}
}

```

```

private static void nonThread() { 1 usage
    int tasksCount = 10; // Уменьшено для наглядности
    Task task = new Task(tasksCount);

    for (int i = 0; i < tasksCount; i++) {
        // 1. Создаем логарифмическую функцию со случайным основанием
        double base = 1 + random.nextDouble() * 9; // от 1 до 10
        Log logFunc = new Log(base);

        // 2. Левая граница: от 0 до 100
        double left = random.nextDouble() * 100;

        // 3. Правая граница: от 100 до 200
        double right = 100 + random.nextDouble() * 100;

        // 4. Шаг дискретизации: от 0 до 1
        double step = random.nextDouble();

        // 5. Выводим сообщение
        System.out.printf(" Source %.4f %.4f %.4f\n", left, right, step);

        // 6. Вычисляем интеграл
        double result = Functions.integrate(logFunc, left, right, step);

        // 7. Выводим результат
        System.out.printf(" Result %.4f %.4f %.4f %.10f\n", left, right, step, result);
    }
}

```

```
2. Последовательная версия (nonThread):
Source 81,0289 143,3632 0,5250
Result 81,0289 143,3632 0,5250 135,5123468938
Source 2,2462 125,8038 0,2487
Result 2,2462 125,8038 0,2487 779,6821779049
Source 91,2898 172,1722 0,4672
Result 91,2898 172,1722 0,4672 179,3771234853
Source 82,0724 171,5939 0,7619
Result 82,0724 171,5939 0,7619 274,7107858710
Source 12,0108 159,2088 0,8124
Result 12,0108 159,2088 0,8124 790,6710713819
Source 79,7814 154,9095 0,4582
Result 79,7814 154,9095 0,4582 182,4778494286
Source 28,1387 192,0251 0,8228
Result 28,1387 192,0251 0,8228 425,3302823715
Source 54,3370 110,6865 0,1754
Result 54,3370 110,6865 0,1754 130,0737527524
Source 6,2650 175,0789 0,4010
Result 6,2650 175,0789 0,4010 333,1392620122
Source 31,0927 136,3869 0,1755
Result 31,0927 136,3869 0,1755 959,0781842331
```

Задание 3

В пакете threads были описаны ещё два класса: SimpleGenerator и SimpleIntegrator. Класс SimpleIntegrator реализует интерфейс Runnable, получает в конструкторе и сохраняет в своё поле ссылку на объект типа Task, а в методе run() в цикле решаются задачи, данные для которых берутся из полученного объекта задания, а также выводится сообщения в консоль.

В главном классе программы создан метод simpleThreads(). В нём создан объект задания, указано количество выполняемых заданий (минимум 100), созданы и запущены два потока вычислений, основанных на описанных классах SimpleGenerator и SimpleIntegrator. Работа метода была проверена путем вызова его в методе main

NullPointerException возникает в интегрирующем потоке, когда он пытается выполнить задание, которое либо:

- Еще не было полностью инициализировано генератором (например, `task.getFunction()` возвращает `null`).
- Было перезаписано генератором во время чтения интегрирующим потоком.
- Уже помечено как выполненное, но интегрирующий поток все еще пытается его прочитать.
- Гонка данных (Race Condition): Генератор устанавливает задание (устанавливает поля), но интегрирующий поток начинает читать до того, как все поля установлены.

Несогласованные данные (например, левая граница из одного задания, а правая граница и шаг из другого) возникают из-за состояния гонки (race condition) и неатомарности операций:

- Генератор начинает устанавливать новое задание
- После установки левой границы, но до установки правой границы и шага, происходит переключение контекста. Интегрирующий поток начинает читать задание и получает:
 - Новую левую границу (уже установлена).
 - Старую правую границу (еще не обновлена)
 - Старый шаг (еще не обновлен).
 - Результат: получается "гибридное" задание, которое никогда не генерировалось

SimpleGenerator:

```
package threads;

import functions.basic.Log;
import java.util.Random;

public class SimpleGenerator implements Runnable { 1 usage
    private Task task; 8 usages
    private Random random; 5 usages
    private int generatedCount; 3 usages

    public SimpleGenerator(Task task) { 5 usages
        this.task = task;
        this.random = new Random();
        this.generatedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

        while (generatedCount < tasksCount) {
            synchronized (task) {
                // Ждем, пока предыдущее задание не будет выполнено
                while (task.isReady() && !task.isCompleted()) {
                    try {
                        task.wait();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return;
                    }
                }
            }
        }
    }
}
```

```

        // Создаем новое задание
        double base = 1 + random.nextDouble() * 9;
        Log logFunc = new Log(base);
        double left = random.nextDouble() * 100;
        double right = 100 + random.nextDouble() * 100;
        double step = random.nextDouble();

        // Устанавливаем задание
        task.setTask(logFunc, left, right, step);
        generatedCount++;

        // Выводим сообщение
        System.out.printf("Generator: Source %.4f %.4f %.4f%n", left, right, step);

        // Уведомляем интегратор
        task.notifyAll();
    }

    // Небольшая задержка для наглядности
    try {
        Thread.sleep(millis: 10);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return;
    }
}
}
}

```

SimpleIntegrator:

```

package threads;

import functions.Functions;

public class SimpleIntegrator implements Runnable { 1 usage
    private Task task; 12 usages
    private int integratedCount; 3 usages

    public SimpleIntegrator(Task task) { 5 usages
        this.task = task;
        this.integratedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

        while (integratedCount < tasksCount) {
            synchronized (task) {
                // Ждем, пока появится задание
                while (!task.isReady() || task.isCompleted()) {
                    try {
                        task.wait();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return;
                    }
                }
            }
        }
    }
}

```

```

        // Получаем задание
        double left = task.getLeft();
        double right = task.getRight();
        double step = task.getStep();

        // Вычисляем интеграл
        double result = Functions.integrate(task.getFunction(), left, right, step);

        // Отмечаем как выполненное
        task.setCompleted(true);
        integratedCount++;

        // Выводим результат
        System.out.printf("Integrator: Result %.4f %.4f %.4f %.10f%n",
            left, right, step, result);

        // Уведомляем генератор
        task.notifyAll();
    }

    // Небольшая задержка для наглядности
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return;
    }
}
}
}

```

Main:

```

private static void simpleThreads() { 1 usage
    int tasksCount = 10; // Уменьшено для наглядности
    Task task = new Task(tasksCount);

    // Создаем потоки
    Thread generatorThread = new Thread(new SimpleGenerator(task));
    Thread integratorThread = new Thread(new SimpleIntegrator(task));

    // Устанавливаем приоритеты (опционально)
    generatorThread.setPriority(Thread.NORM_PRIORITY);
    integratorThread.setPriority(Thread.NORM_PRIORITY);

    // Запускаем потоки
    generatorThread.start();
    integratorThread.start();

    // Ждем завершения потоков
    try {
        generatorThread.join();
        integratorThread.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println(" Все задания выполнены!");
}
}

```

```
3. Простая многопоточная версия (simpleThreads):
Generator: Source 5,8290 136,4218 0,9320
Integrator: Result 5,8290 136,4218 0,9320 428,6132336692
Generator: Source 27,2290 127,0935 0,3128
Integrator: Result 27,2290 127,0935 0,3128 283,8337042180
Generator: Source 9,3635 140,2910 0,0538
Integrator: Result 9,3635 140,2910 0,0538 281,7092715216
Generator: Source 49,6925 113,5412 0,0967
Integrator: Result 49,6925 113,5412 0,0967 126,8881412651
Generator: Source 63,9485 169,1855 0,8095
Integrator: Result 63,9485 169,1855 0,8095 260,5072503377
Generator: Source 69,8478 125,5048 0,1777
Integrator: Result 69,8478 125,5048 0,1777 140,4170253681
Generator: Source 52,8440 147,8134 0,1271
Integrator: Result 52,8440 147,8134 0,1271 290,6850459708
Generator: Source 28,0859 184,4688 0,8794
Integrator: Result 28,0859 184,4688 0,8794 592,2834206631
Generator: Source 65,1839 189,0957 0,9050
Integrator: Result 65,1839 189,0957 0,9050 487,0870718053
Generator: Source 47,9570 164,7930 0,7340
Integrator: Result 47,9570 164,7930 0,7340 361,6520153356
Все задания выполнены!
```

Задание 4

Для решения проблемы того, что не все сгенерированные задания оказываются выполнены интегрирующим потоком, в пакет `threads` были добавлены два следующих класса: `Generator` и `Integrator`. Их отличие от предыдущих методов заключается в том, что для их реализации используется метод семафора.

Далее в классе `Main` был добавлен метод `complicatedThreads`, написанный для выполнения поставленных задач, и программа была запущена на выполнение. В итоге был получен вывод, показывающий правильность работы программы:

Generator:

```

package threads;

import functions.basic.Log;
import java.util.Random;

public class Generator extends Thread {
    private Task task;
    private Semaphore semaphore;
    private Random random;
    private int generatedCount;
    private volatile boolean running = true;

    public Generator(Task task, Semaphore semaphore) {
        this.task = task;
        this.semaphore = semaphore;
        this.random = new Random();
        this.generatedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

```

```

        try {
            while (generatedCount < tasksCount && running) {
                semaphore.beginWrite();

                // Создаем новое задание
                double base = 1 + random.nextDouble() * 9;
                Log logFunc = new Log(base);
                double left = random.nextDouble() * 100;
                double right = 100 + random.nextDouble() * 100;
                double step = random.nextDouble();

                // Устанавливаем задание
                task.setTask(logFunc, left, right, step);
                generatedCount++;

                // Выводим сообщение
                System.out.printf("Generator[%d]: Source %.4f %.4f %.4f\n",
                    generatedCount, left, right, step);

                semaphore.endWrite();

                // Небольшая задержка
                Thread.sleep(10);
            }
        } catch (InterruptedException e) {
            System.out.println("Generator прерван");
        } finally {
            running = false;
        }
    }

    public void stopGenerator() {
        running = false;
        this.interrupt();
    }
}

```

Integrator:


```

package threads;

import functions.Functions;

public class Integrator extends Thread { 4 usages
    private Task task; 9 usages
    private Semaphore semaphore; 3 usages
    private int integratedCount; 4 usages
    private volatile boolean running = true; 3 usages

    public Integrator(Task task, Semaphore semaphore) { 6 usages
        this.task = task;
        this.semaphore = semaphore;
        this.integratedCount = 0;
    }

    @Override
    public void run() {
        int tasksCount = task.getTasksCount();

        try {
            while (integratedCount < tasksCount && running) {
                semaphore.beginRead();

                // Проверяем, готово ли задание
                if (task.isReady() && !task.isCompleted()) {
                    // Получаем задание
                    double left = task.getLeft();
                    double right = task.getRight();
                    double step = task.getStep();

                    // Вычисляем интеграл

```

```

        // Вычисляем интеграл
        double result = Functions.integrate(task.getFunction(), left, right, step);

        // Отмечаем как выполненное
        task.setCompleted(true);
        integratedCount++;

        // Выводим результат
        System.out.printf("Integrator[%d]: Result %.4f %.4f %.4f %.10f%n",
            integratedCount, left, right, step, result);
    }

    semaphore.endRead();

    // Небольшая задержка
    Thread.sleep( millis: 10);
}
} catch (InterruptedException e) {
    System.out.println("Integrator прерван");
} finally {
    running = false;
}
}

public void stopIntegrator() { 1 usage
    running = false;
    this.interrupt();
}
}
}

```

Main:

```
private static void complicatedThreads() { 1 usage
    int tasksCount = 10; // Уменьшено для наглядности
    Task task = new Task(tasksCount);
    Semaphore semaphore = new Semaphore();

    // Создаем потоки
    Generator generator = new Generator(task, semaphore);
    Integrator integrator = new Integrator(task, semaphore);

    // Устанавливаем приоритеты (опционально)
    generator.setPriority(Thread.NORM_PRIORITY);
    integrator.setPriority(Thread.NORM_PRIORITY);

    // Запускаем потоки
    generator.start();
    integrator.start();

    // Ждем 50 мс и прерываем
    try {
        Thread.sleep( millis: 50);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // Прерываем потоки
    System.out.println(" Прерывание потоков после 50 мс...");
    generator.stopGenerator();
    integrator.stopIntegrator();
}
```

```
    // Ждем завершения потоков
    try {
        generator.join();
        integrator.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println(" Потоки завершены!");
}
```

Также проводим проверку потоков, изменяя приоритеты, для этого также вводим изменения в класс Main:

```
private static void testWithDifferentPriorities() { 1 usage
    int tasksCount = 20;

    // Тест 1: Генератор с высоким приоритетом
    System.out.println(" а) Генератор с высоким приоритетом:");
    testPriority(tasksCount, Thread.MAX_PRIORITY, Thread.MIN_PRIORITY);

    // Тест 2: Интегратор с высоким приоритетом
    System.out.println(" б) Интегратор с высоким приоритетом:");
    testPriority(tasksCount, Thread.MIN_PRIORITY, Thread.MAX_PRIORITY);

    // Тест 3: Одинаковые приоритеты
    System.out.println(" в) Одинаковые приоритеты:");
    testPriority(tasksCount, Thread.NORM_PRIORITY, Thread.NORM_PRIORITY);
}
```

```
private static void testPriority(int tasksCount, int generatorPriority, int integratorPriority) {
    Task task = new Task(tasksCount);
    Semaphore semaphore = new Semaphore();

    Generator generator = new Generator(task, semaphore);
    Integrator integrator = new Integrator(task, semaphore);

    generator.setPriority(generatorPriority);
    integrator.setPriority(integratorPriority);

    long startTime = System.currentTimeMillis();

    generator.start();
    integrator.start();

    try {
        generator.join();
        integrator.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    long endTime = System.currentTimeMillis();
    System.out.printf("    Время выполнения: %d мс\n", endTime - startTime);
}
```

В результате получается следующий вывод, что согласуется с поставленными задачами работы:

4. Версия с семафором (complicatedThreads):

```
Generator[1]: Source 96,7765 115,7889 0,2290
Integrator[1]: Result 96,7765 115,7889 0,2290 46,2496632534
Generator[2]: Source 36,1847 179,7274 0,1242
Generator[3]: Source 0,0160 144,6029 0,0759
Integrator[2]: Result 0,0160 144,6029 0,0759 995,9477090846
Generator[4]: Source 1,2473 107,3945 0,4088
Integrator[3]: Result 1,2473 107,3945 0,4088 180,5073915985
Generator[5]: Source 91,6642 182,6307 0,5498
Integrator[4]: Result 91,6642 182,6307 0,5498 210,0004515772
Прерывание потоков после 50 мс...
Generator прерван
Integrator прерван
Потоки завершены!
```

5. Тест с разными приоритетами:

а) Генератор с высоким приоритетом:

```
Generator[1]: Source 37,3274 143,3788 0,6107
Integrator[1]: Result 37,3274 143,3788 0,6107 216,5578448590
Generator[2]: Source 47,3258 148,6037 0,6517
Integrator[2]: Result 47,3258 148,6037 0,6517 423,0367411490
Generator[3]: Source 37,2686 143,8590 0,8759
Integrator[3]: Result 37,2686 143,8590 0,8759 304,5181977063
Generator[4]: Source 12,8775 170,4153 0,6610
Integrator[4]: Result 12,8775 170,4153 0,6610 346,5054095951
Generator[5]: Source 31,0688 137,3253 0,8461
Integrator[5]: Result 31,0688 137,3253 0,8461 968,9359799513
Generator[6]: Source 95,1195 156,7239 0,1656
Integrator[6]: Result 95,1195 156,7239 0,1656 181,0451674121
Generator[7]: Source 67,0213 186,1031 0,5632
Integrator[7]: Result 67,0213 186,1031 0,5632 317,0196535134
Generator[8]: Source 63,2369 137,8430 0,8858
Integrator[8]: Result 63,2369 137,8430 0,8858 153,2862823648
Generator[9]: Source 93,1389 138,2538 0,0267
Integrator[9]: Result 93,1389 138,2538 0,0267 93,9569941456
Generator[10]: Source 45,6040 118,4786 0,1897
Integrator[10]: Result 45,6040 118,4786 0,1897 583,4431160006
Generator[11]: Source 28,5490 197,8335 0,5815
Integrator[11]: Result 28,5490 197,8335 0,5815 410,0462970511
Generator[12]: Source 18,6398 176,5057 0,5030
Integrator[12]: Result 18,6398 176,5057 0,5030 1191,1377736644
Generator[13]: Source 83,9685 189,0045 0,5070
Generator[14]: Source 1,1866 182,3576 0,0173
Integrator[13]: Result 1,1866 182,3576 0,0173 334,2712495745
Generator[15]: Source 0,3973 173,7676 0,5636
Integrator[14]: Result 0,3973 173,7676 0,5636 454,2829629915
Generator[16]: Source 36,0533 111,2914 0,5753
Integrator[15]: Result 36,0533 111,2914 0,5753 151,1816673579
Generator[17]: Source 0,1707 108,0780 0,4998
Integrator[16]: Result 0,1707 108,0780 0,4998 334,7230545002
Generator[18]: Source 65,3629 145,7969 0,8630
Integrator[17]: Result 65,3629 145,7969 0,8630 256,0147803926
Generator[19]: Source 69,5129 146,1134 0,7735
Integrator[18]: Result 69,5129 146,1134 0,7735 201,8495719041
Generator[20]: Source 31,3464 190,4137 0,2864
Integrator[19]: Result 31,3464 190,4137 0,2864 408,1301975827
```

