ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЁВА»

КАФЕДРА «ТЕХНИЧЕСКАЯ КИБЕРНЕТИКА»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Объектно-ориентированное программирование»

ЛАБОРАТОРНАЯ РАБОТА НОМЕР 5

Студент Горейнов Д.В.

Группа 6301-030301D

Проверил Борисов.Д.С

Оценка _____

САМАРА 2025

Задание 1

Добавил в класс Functions метод, возвращающий значение интеграла функции, вычисленное с помощью численного метода.

Вычисление интеграла выполняется по методу трапеций.

Основная формула: $\frac{(x2-x1)(y1+y2)}{2}$, где x2-x1 является высотой трапеции, а $\frac{(y1+y2)}{2}$ является полусуммой оснований.

```
public static double integrate(Function function, double leftBorder, double rightBorder, double step)
{
    if (leftBorder >= rightBorder)
    {
        throw new IllegalArgumentException("Левая граница должна быть меньше правой: leftBorder=" + leftBorder + ", rightBorder=" + rightBorder);
    }
        if (step <= 0)
    {
        throw new IllegalArgumentException("Шаг должен быть положительным: step=" + step);
    }

        // Проверка области определения
        if (leftBorder < function.getLeftDomainBorder() || rightBorder > function.getRightDomainBorder())
        {
            throw new IllegalArgumentException(String.format("Интервал [%.2f, %.2f] выходит за границы области определения [%.2f, %.2f]",leftBorder, rightBorder, function.getLeftDomainBorder(), function.getRi
        }
    double integral = 0;
    double x=leftBorder;
    int n= (int) Math.ceil((rightBorder-leftBorder)/step);
    for(int i=0; i<n;i++)
    {
        double x1=x;
        double x2 = (i == n - 1) ? rightBorder : Math.min(x + step, rightBorder);
        double y1 = function.getFunctionValue(x1);
        double y2 = function.getFunctionValue(x2);
        double S = (y1+y2)*(x2-x1)/2.0;
        integral += S;
        x=x2;
    }
    return integral;
}

}
```

Фото 1

Была проведена проверка метода интегрирования:

```
public static void main(String[] args)
{
    Function expFunction = new Exp();
    System.out.println("Тест 1");
    double accuracy = 1e-7;
    double left = 0;
    double right = 1;
    double equivalentValue = Math.E - 1;
    double step = 0.00001;

    double integral = Functions.integrate(expFunction, left, right, step);

    System.out.printf("Вычисленный интеграл: %.10f%n", integral);
    System.out.printf("Теоретическое значение: %.10f%n", equivalentValue);
    System.out.printf("Разница: %.2e%n", Math.abs(integral - equivalentValue));

    if(Math.abs(integral - equivalentValue) <= accuracy)
    {
        System.out.println("Необходимая точность достигнута, интеграл найден верно");
    }
    else
    {
        System.out.println("Необходимая точность не достигнута");
    }
```

Фото 2

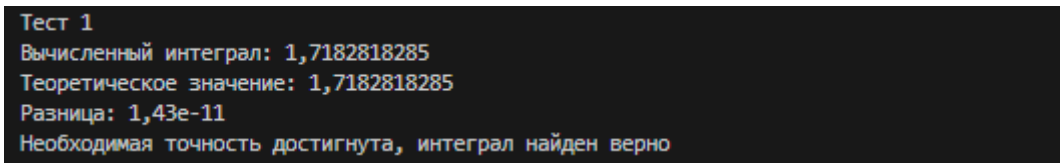


Фото 3

Оптимальный шаг для достижения необходимой точности равен 0.0001

Следовательно метод работает корректно и может быть использован в дальнейшей работе.

Задание 2

Создал пакет threads, в котором размещены классы, связанные с потоками.

1)В пакете threads описа класс Task, объект которого должен хранить ссылку на объект интегрируемой функции, границы области интегрирования, шаг дискретизации, а также целочисленное поле, хранящее количество выполняемых заданий.

```java
public class Task
{
    private Function function;
    private double leftBorder;
    private double rightBorder;
    private double step;
    private int taskCount;

    public Task()
    {
        this.taskCount = 0;
    }

    public Task(int taskCount)
    {
        this.taskCount = taskCount;
    }
     public Function getFunction() {
        return function;
    }

    public double getLeftBorder() {
        return leftBorder;
    }

    public double getRightBorder() {
        return rightBorder;
    }

    public double getStep() {
        return step;
    }

    public int getTaskCount() {
        return taskCount;
    }

    public void setFunction(Function function) {
        this.function = function;
    }

    public void setLeftBorder(double leftBorder) {
        this.leftBorder = leftBorder;
    }

    public void setRightBorder(double rightBorder) {
        this.rightBorder = rightBorder;
    }

    public void setStep(double step) {
        this.step = step;
    }

    public void setTaskCount(int taskCount) {
        this.taskCount = taskCount:
```

Фото 4

2)В main создал метод nonThread() реализующий последовательную (без применения потоков инструкций) версию программы:

Random-генерирует случайные числа
Для основания воспользовался циклом do while, чтобы избежать основания равному 1

Метод nextDouble() возвращает псевдослучайное число от 0.0 до 1.0. поэтому мы умножаем результат на 100

```
public static void nonThread()
{
    Random random = new Random();
    Task task = new Task();
    task.setTaskCount(taskCount: 100);

    System.out.println("ПОСЛЕДОВАТЕЛЬНАЯ ОБРАБОТКА ЗАДАНИЙ");


    for (int i = 0; i < task.getTaskCount(); i++) {
        try {
            double base;
            do {
                base = 1 + random.nextDouble() * 9; // [1, 10]
            } while (Math.abs(base - 1.0) < 1e-7); // избегаем основание = 1
            Function logFunction = new Log(base);
            task.setFunction(logFunction);


            double leftBorder = random.nextDouble() * 100; // [0, 100]
            task.setLeftBorder(leftBorder);


            double rightBorder = 100 + random.nextDouble() * 100; // [100, 200]
            task.setRightBorder(rightBorder);

            //Шаг дискретизации от 0 до 1
            double step;
            do {
                step = random.nextDouble(); // [0, 1]
            } while (step == 0.0); // избегаем нулевой шаг
            task.setStep(step);

            //Вывод информации о задании
            System.out.printf("Source %d: %.6f %.6f %.6f%n", i + 1, task.getLeftBorder(), task.getRightBorder(), task.getStep());
            double integral = Functions.integrate(task.getFunction(), task.getLeftBorder(), task.getRightBorder(), task.getStep());
            System.out.printf("Result %d: %.6f %.6f %.6f %.6f%n",i + 1, task.getLeftBorder(), task.getRightBorder(), task.getStep(), integral);
            System.out.println("_________");

        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка в задании " + (i + 1) + ": " + e.getMessage());
            System.out.println("Повторяем задание...");
            i--; // Повторяем это задание
        }
    }

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

Фото 5

Была проведена проверка, метод работает верно, все задачи были решены

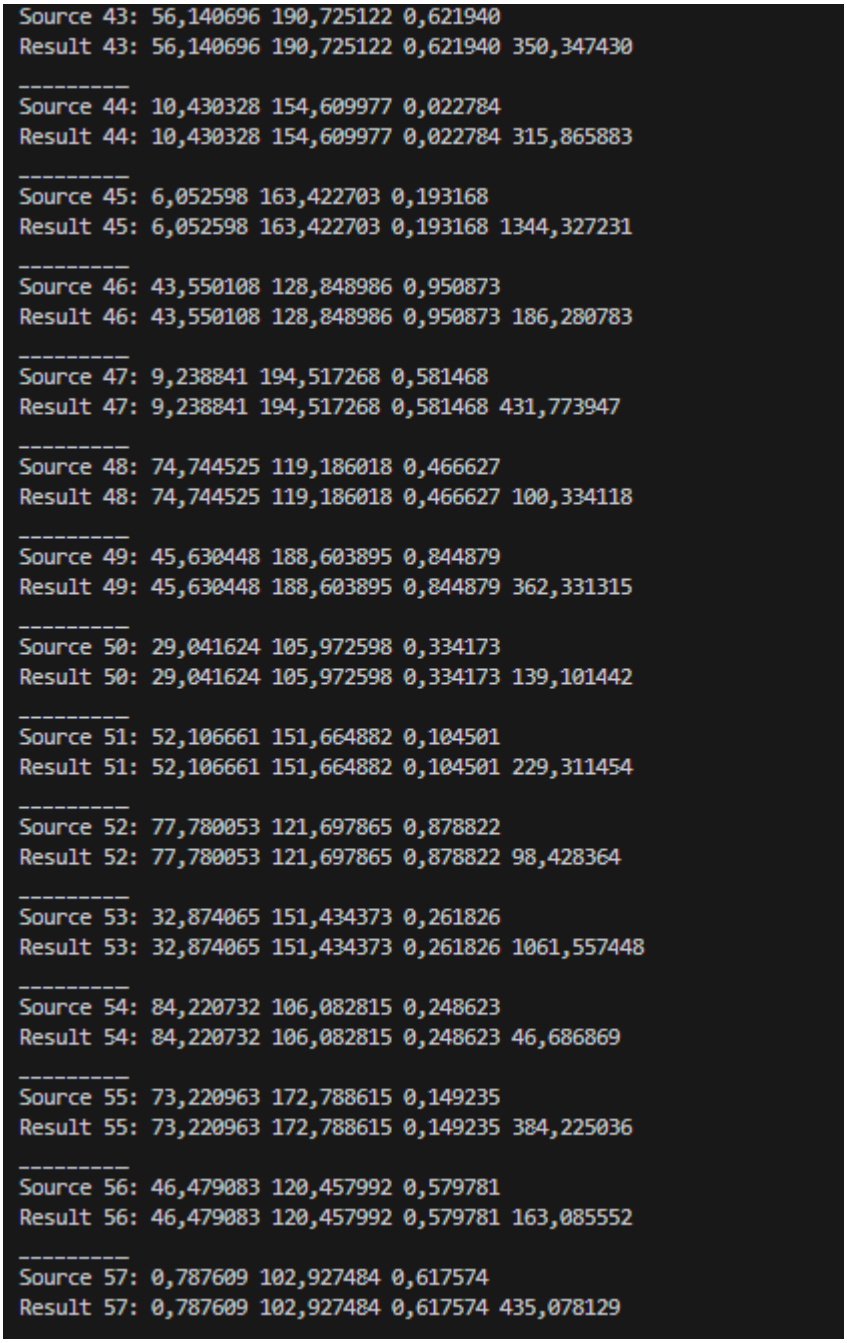


Фото 6

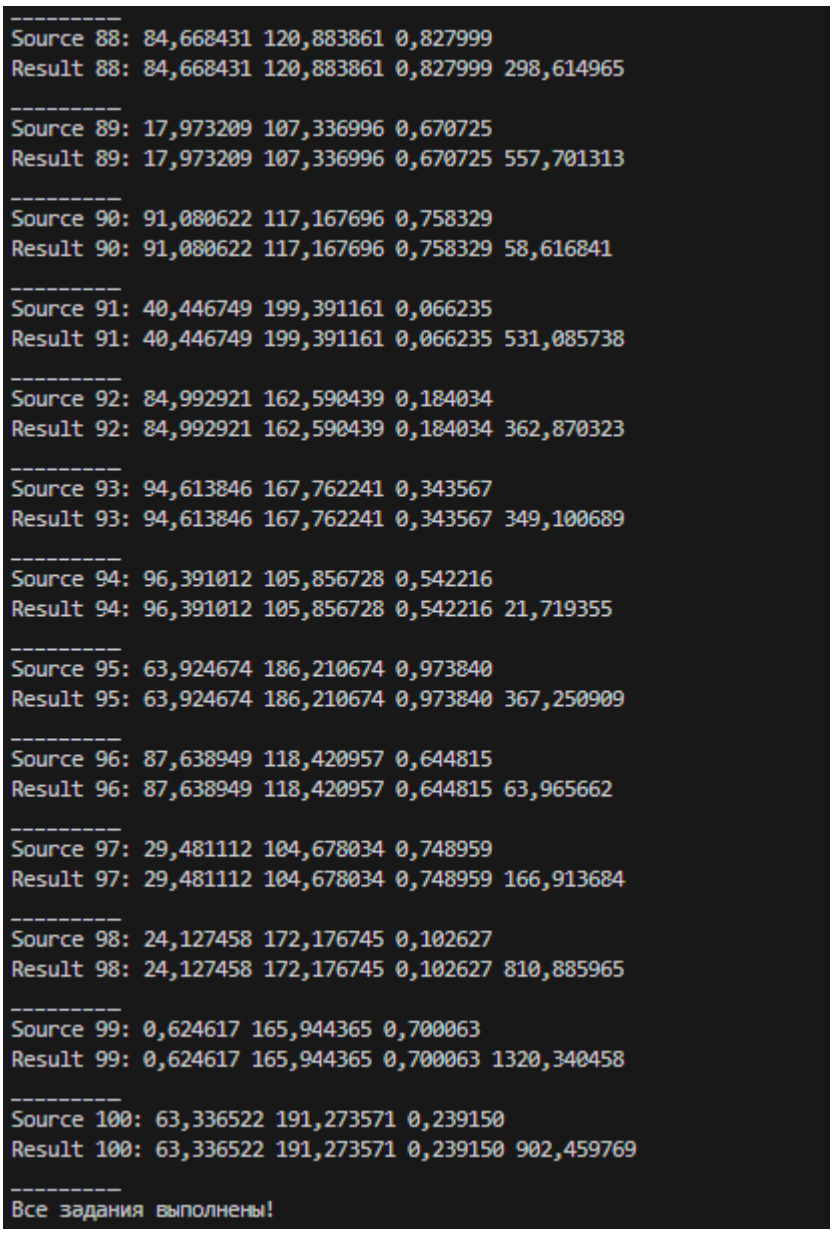
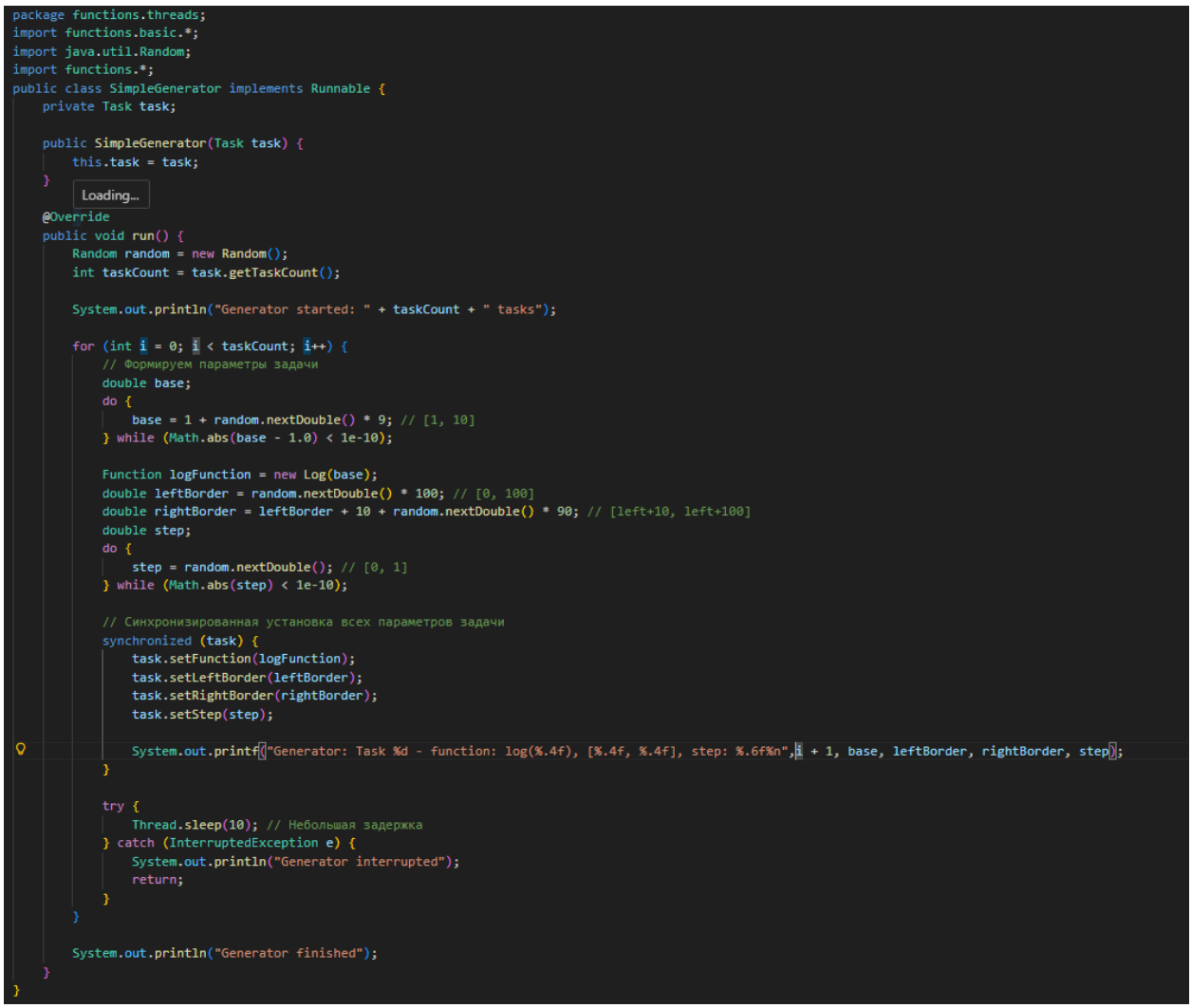


Фото 7

Задание 3

В пакете threads создал два следующих класса.

1)Класс SimpleGenerator реализовывает интерфейс Runnable, получает в конструкторе и сохраняет в своём поле ссылку на объект типа Task, а в методе run() в цикле формируются задачи и заносятся в полученный объект задания, а также выводятся сообщения в консоль.



```java
package functions.threads;
import functions.basic.*;
import java.util.Random;
import functions.*;
public class SimpleGenerator implements Runnable {
    private Task task;

    public SimpleGenerator(Task task) {
        this.task = task;
    }
    Loading...
    @Override
    public void run() {
        Random random = new Random();
        int taskCount = task.getTaskCount();

        System.out.println("Generator started: " + taskCount + " tasks");

        for (int i = 0; i < taskCount; i++) {
            // Формируем параметры задачи
            double base;
            do {
                base = 1 + random.nextDouble() * 9; // [1, 10]
            } while (Math.abs(base - 1.0) < 1e-10);

            Function logFunction = new Log(base);
            double leftBorder = random.nextDouble() * 100; // [0, 100]
            double rightBorder = leftBorder + 10 + random.nextDouble() * 90; // [left+10, left+100]
            double step;
            do {
                step = random.nextDouble(); // [0, 1]
            } while (Math.abs(step) < 1e-10);

            // Синхронизированная установка всех параметров задачи
            synchronized (task) {
                task.setFunction(logFunction);
                task.setLeftBorder(leftBorder);
                task.setRightBorder(rightBorder);
                task.setStep(step);

                System.out.printf("Generator: Task %d - function: log(%.4f), [%.4f, %.4f], step: %.6f%n",i + 1, base, leftBorder, rightBorder, step);
            }

            try {
                Thread.sleep(10); // Небольшая задержка
            } catch (InterruptedException e) {
                System.out.println("Generator interrupted");
                return;
            }
        }

        System.out.println("Generator finished");
    }
}
```

Фото 8

2)Класс SimpleIntegrator реализовывает интерфейс Runnable, получает в конструкторе и сохраняет в своём поле ссылку на объект типа Task, а в методе run() в цикле решаются задачи, данные для которых берутся из полученного объекта задания, а также выводиться сообщения в консоль.

```java
package functions.threads;

import functions.Function;
import functions.Functions;

public class SimpleIntegrator implements Runnable {
    private Task task;

    public SimpleIntegrator(Task task) {
        this.task = task;
    }

    @Override
    public void run() {
        int tasksProcessed = 0;
        int taskCount = task.getTaskCount();

        System.out.println("Integrator started");

        while (tasksProcessed < taskCount) {
            Function function = null;
            double leftBorder, rightBorder, step;

            // Синхронизированное чтение всех параметров задачи
            synchronized (task) {
                function = task.getFunction();
                leftBorder = task.getLeftBorder();
                rightBorder = task.getRightBorder();
                step = task.getStep();
            }

            // Проверка на null для избежания NullPointerException
            if (function == null) {
                try {
                    Thread.sleep(1); // Короткая пауза если данные еще не готовы
                } catch (InterruptedException e) {
                    System.out.println("Integrator interrupted");
                    return;
                }
                continue;
            }

            try {
                // Используем метод integrate из класса Functions
                double integral = Functions.integrate(function, leftBorder, rightBorder, step);

                System.out.printf("Integrator: Task %d - [%.4f, %.4f], step: %.6f, result: %.6f%n",
                    tasksProcessed + 1, leftBorder, rightBorder, step, integral);

                tasksProcessed++;

            } catch (IllegalArgumentException e) {
                System.out.printf("Integrator: Invalid task %d - %s%n",
                    tasksProcessed + 1, e.getMessage());
                tasksProcessed++; // Все равно считаем задачу обработанной
            }

            try {
                Thread.sleep(10); // Небольшая задержка
            } catch (InterruptedException e) {
                System.out.println("Integrator interrupted");
                return;
            }
        }

        System.out.println("Integrator finished: " + tasksProcessed + " tasks processed");
    }
```

Фото 9

В main создал метод simpleThreads() для проверки.

В нём создал объект задания, указал количество выполняемых заданий в размере 100 штук, создал и запустил два потока вычислений, основанных на описанных

классах SimpleGenerator и SimpleIntegrator.

```java
public static void simpleThreads() {
    Task task = new Task(taskCount: 100);
    // Создаем потоки
    Thread generatorThread = new Thread(new SimpleGenerator(task));
    Thread integratorThread = new Thread(new SimpleIntegrator(task));
    // Эксперименты с приоритетами
    //одинаковые приоритеты
    generatorThread.setPriority(Thread.NORM_PRIORITY);
    integratorThread.setPriority(Thread.NORM_PRIORITY);
    //разные приоритеты_1
    // generatorThread.setPriority(Thread.MAX_PRIORITY);
    // integratorThread.setPriority(Thread.MIN_PRIORITY);
    //разные приоритеты_2
    // generatorThread.setPriority(Thread.MIN_PRIORITY);
    // integratorThread.setPriority(Thread.MAX_PRIORITY);

    System.out.println("Starting threads with priorities - " +"Generator: " + generatorThread.getPriority() +", Integrator: " + integratorThread.getPriority());

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

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

    System.out.println("All threads completed");
}
```

Фото 10

Результаты запуска:

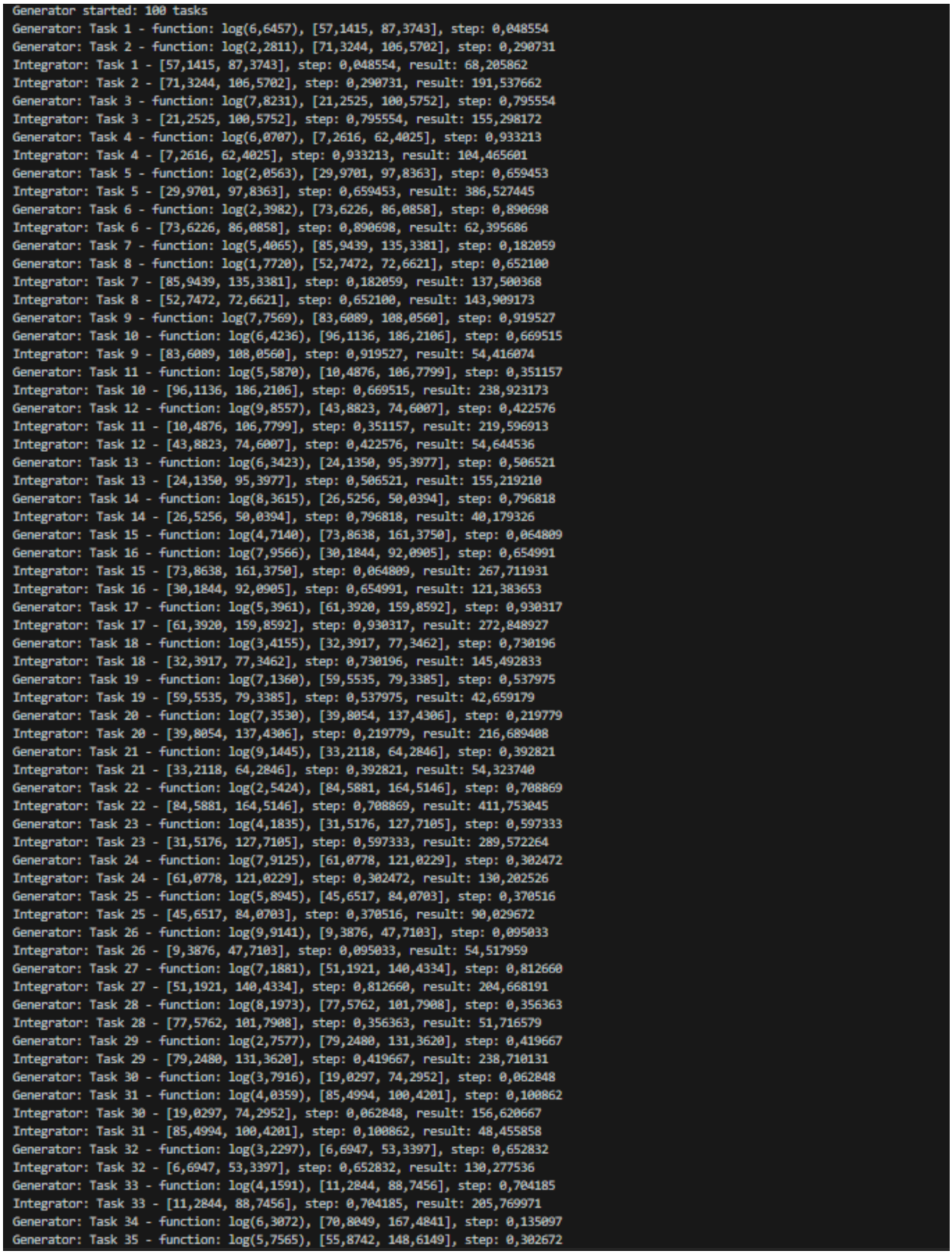


Фото 11

Экспериментально было найдено лучшее соотношение:

Максимальный приоритет на интегратор и минимальный на генератор
Предварительно в интеграторе повысил время до Thread.sleep(10);

Задание 4

Определите причину того, что не все сгенерированные задания оказываются выполнены интегрирующим потоком:
генератор записывает данные в Task, интегратор читает эти данные ,но нет механизма подтверждения, что он читает именно этот номер задания. То есть интегратор может решить одну и ту же задачу дважды, при этом пропустив другую.

Есть несколько вариантов исправления, для примера можно сделать флажки(сколько задач создано и сколько прочитано) или семафор.

Семафор работает как светофор:
Семафор:

- Красный (writers > 0): "Идет запись" - все читатели ждут

- Желтый (writeRequests > 0): "Очередь на запись" - новые читатели ждут

- Зеленый (readers > 0): "Идет чтение" - новые писатели ждут

(Было решено использовать стандартную реализацию семафора)

В пакете threads создал два следующих класса. При их реализации воспользовался фрагментами классов из предыдущего задания.

Класс Generator расширяет класс Thread, получает в конструкторе и сохраняет в свои поля ссылки на объект типа Task и на объект семафора, а в методе run() выполняются те же действия, что и в предыдущей версии генерирующего класса.

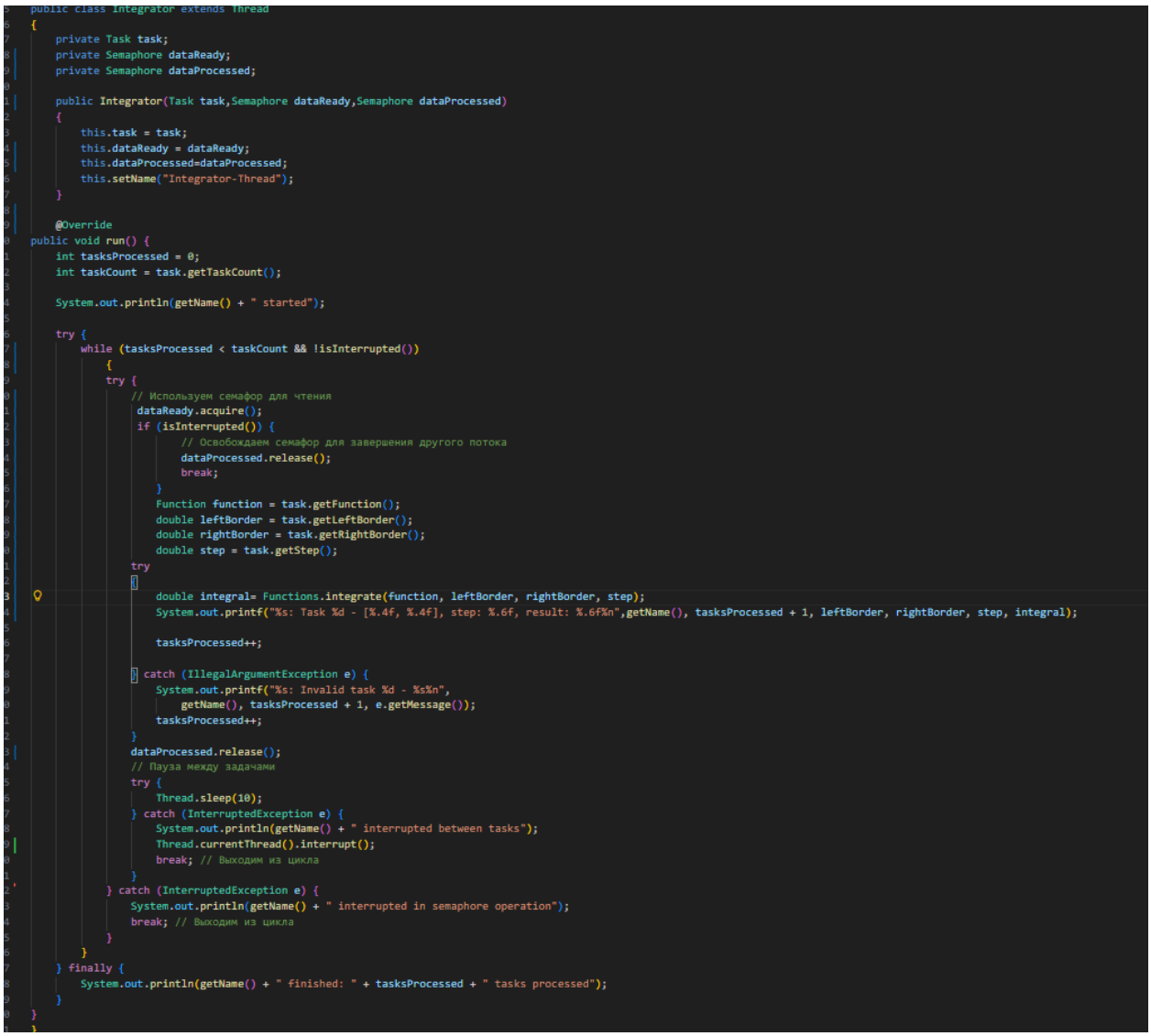
```java
package functions.threads;
import functions.basic.*;
import java.util.Random;
import java.util.concurrent.Semaphore;
public class Generator extends Thread {
private Task task;
Semaphore dataReady;
Semaphore dataProcessed;
    public Generator(Task task, Semaphore dataReady, Semaphore dataProcessed)
    {
        this.task = task;
        this.dataReady = dataReady;
        this.dataProcessed = dataProcessed;
        this.setName("Generator-Thread");
    }

 @Override
public void run() {
    Random random = new Random();
    int taskCount = task.getTaskCount();

    System.out.println(getName() + " started: " + taskCount + " tasks");

    try {
        for (int i = 0; i < taskCount && !isInterrupted(); i++) {
            try {
                dataProcessed.acquire();

                if (isInterrupted())
                    {
                        // Освобождаем семафор для завершения другого потока
                        dataReady.release();
                        break;
                    }
                // Генерируем параметры
                double base;
                do {
                    base = 1 + random.nextDouble() * 9;
                } while (Math.abs(base - 1.0) < 1e-10);

                Log logFunc = new Log(base);

                double leftBorder = random.nextDouble() * 100;
                double rightBorder;
                do {
                    rightBorder = leftBorder + 1 + random.nextDouble() * 100;
                } while (rightBorder <= leftBorder);

                double step;
                do {
                    step = random.nextDouble();
                } while (Math.abs(step) < 1e-10);

                if (leftBorder <= 0) {
                    leftBorder = 0.1;
                    rightBorder = Math.max(rightBorder, leftBorder + 1);
                }

                try {
                    task.setFunction(logFunc);
                    task.setLeftBorder(leftBorder);
                    task.setRightBorder(rightBorder);
                    task.setStep(step);

                    System.out.printf("%s %d: Source %.6f %.6f %.6f (base=%.4f)%n",
                                      getName(), i + 1,
                                      leftBorder, rightBorder, step, base);
```

Фото 13

```
                } finally {

                    dataReady.release();                }

                // Пауза
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    System.out.println(getName() + " interrupted during sleep");
                    Thread.currentThread().interrupt();
                    break; // Выходим из цикла
                }

            } catch (InterruptedException e) {
                System.out.println(getName() + " interrupted in semaphore operation");
                break; // Выходим из цикла
            } catch (Exception e) {
                System.out.println(getName() + " error: " + e.getMessage());
                dataProcessed.release();
                i--; // Повторяем итерацию
            }
        }
    } finally {
        System.out.println(getName() + " finished");
    }
}
}
```

Фото 14

Аналогично сделал Integrator



```
public class Integrator extends Thread
{
    private Task task;
    private Semaphore dataReady;
    private Semaphore dataProcessed;

    public Integrator(Task task,Semaphore dataReady,Semaphore dataProcessed)
    {
        this.task = task;
        this.dataReady = dataReady;
        this.dataProcessed=dataProcessed;
        this.setName("Integrator-Thread");
    }

    @Override
public void run() {
    int tasksProcessed = 0;
    int taskCount = task.getTaskCount();

    System.out.println(getName() + " started");

    try {
        while (tasksProcessed < taskCount && !isInterrupted())
            {
            try {
                // Используем семафор для чтения
                dataReady.acquire();
                if (isInterrupted()) {
                        // Освобождаем семафор для завершения другого потока
                        dataProcessed.release();
                        break;
                    }
                    Function function = task.getFunction();
                    double leftBorder = task.getLeftBorder();
                    double rightBorder = task.getRightBorder();
                    double step = task.getStep();
                try
                {
                    double integral= Functions.integrate(function, leftBorder, rightBorder, step);
                    System.out.printf("%s: Task %d - [%.4f, %.4f], step: %.6f, result: %.6f%n",getName(), tasksProcessed + 1, leftBorder, rightBorder, step, integral);

                    tasksProcessed++;

                } catch (IllegalArgumentException e) {
                    System.out.printf("%s: Invalid task %d - %s%n",
                        getName(), tasksProcessed + 1, e.getMessage());
                    tasksProcessed++;
                }
                dataProcessed.release();
                // Пауза между задачами
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    System.out.println(getName() + " interrupted between tasks");
                    Thread.currentThread().interrupt();
                    break; // Выходим из цикла
                }
            } catch (InterruptedException e) {
                System.out.println(getName() + " interrupted in semaphore operation");
                break; // Выходим из цикла
            }
        }
    } finally {
        System.out.println(getName() + " finished: " + tasksProcessed + " tasks processed");
    }
}
}
```

Фото 15

Отличие этих классов от предыдущих версий заключается в том, что вместо средств синхронизации в методах run() должны использоваться возможности семафора.

В главном классе программы создал метод complicatedThreads(). В нём создал объект задания

Регулярная проверка: if (isInterrupted()) или while (!isInterrupted())

Обработка InterruptedException: Немедленный выход из метода

Восстановление статуса (опционально):

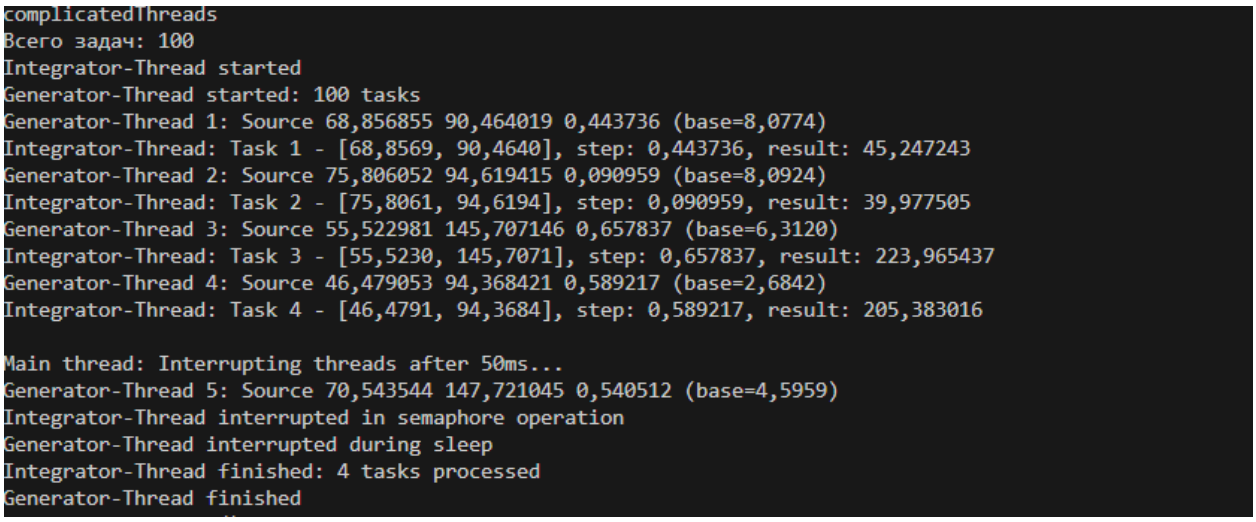


Фото 17(Thread.sleep(50))

Thread.sleep(2000)

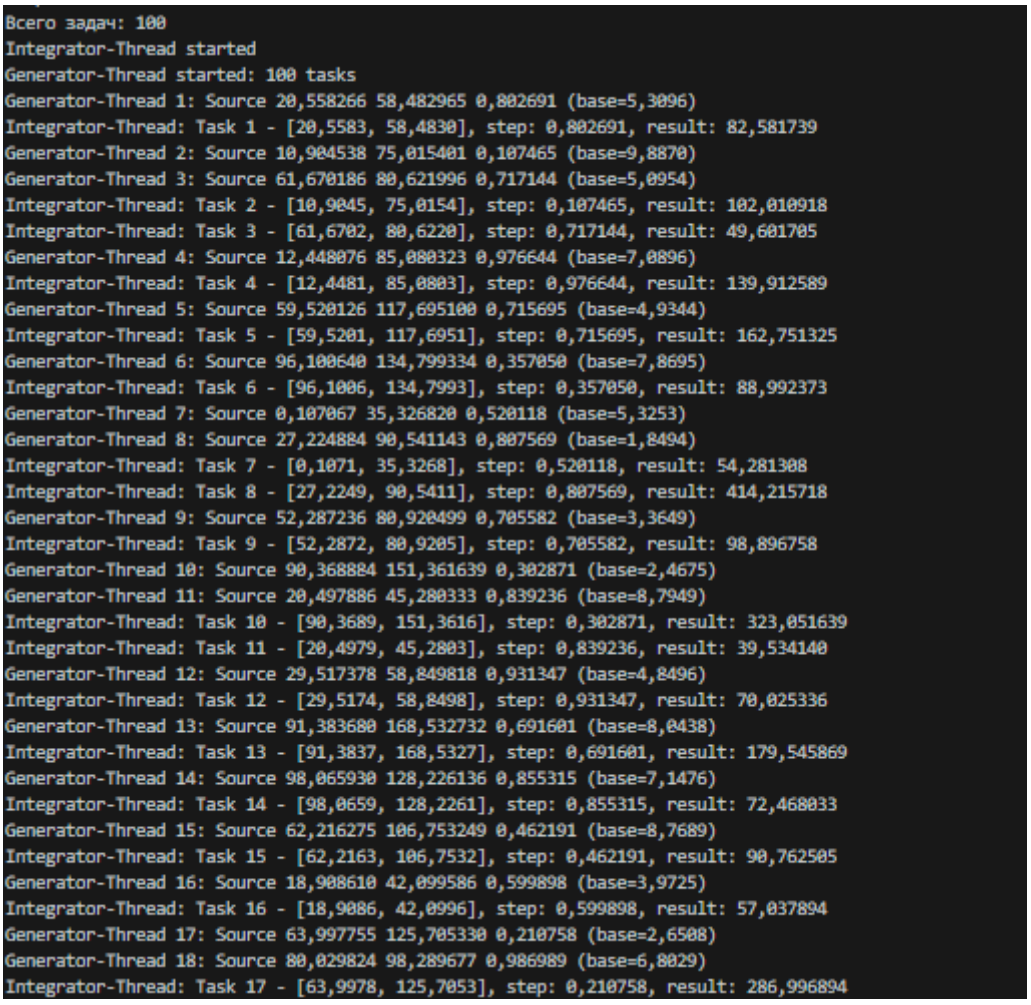


Фото 18

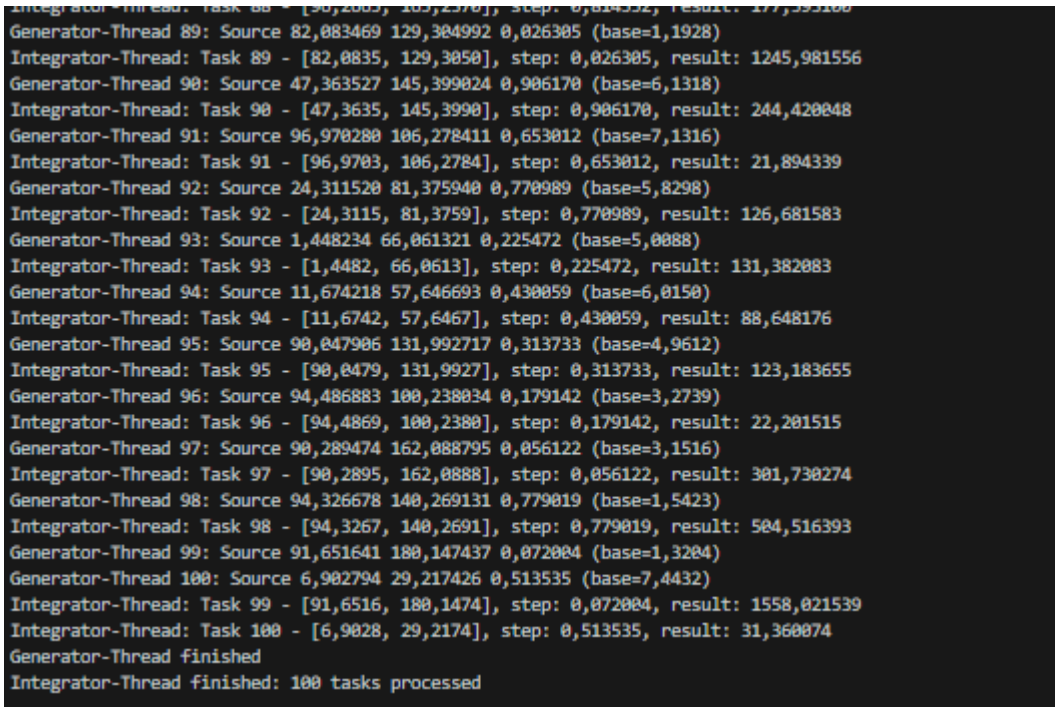

Generator-Thread 89: Source 82,083469 129,304992 0,026305 (base=1,1928)
Integrator-Thread: Task 89 - [82,0835, 129,3050], step: 0,026305, result: 1245,981556
Generator-Thread 90: Source 47,363527 145,399024 0,906170 (base=6,1318)
Integrator-Thread: Task 90 - [47,3635, 145,3990], step: 0,906170, result: 244,420048
Generator-Thread 91: Source 96,970280 106,278411 0,653012 (base=7,1316)
Integrator-Thread: Task 91 - [96,9703, 106,2784], step: 0,653012, result: 21,894339
Generator-Thread 92: Source 24,311520 81,375940 0,770989 (base=5,8298)
Integrator-Thread: Task 92 - [24,3115, 81,3759], step: 0,770989, result: 126,681583
Generator-Thread 93: Source 1,448234 66,061321 0,225472 (base=5,0088)
Integrator-Thread: Task 93 - [1,4482, 66,0613], step: 0,225472, result: 131,382083
Generator-Thread 94: Source 11,674218 57,646693 0,430059 (base=6,0150)
Integrator-Thread: Task 94 - [11,6742, 57,6467], step: 0,430059, result: 88,648176
Generator-Thread 95: Source 90,047906 131,992717 0,313733 (base=4,9612)
Integrator-Thread: Task 95 - [90,0479, 131,9927], step: 0,313733, result: 123,183655
Generator-Thread 96: Source 94,486883 100,238034 0,179142 (base=3,2739)
Integrator-Thread: Task 96 - [94,4869, 100,2380], step: 0,179142, result: 22,201515
Generator-Thread 97: Source 90,289474 162,088795 0,056122 (base=3,1516)
Integrator-Thread: Task 97 - [90,2895, 162,0888], step: 0,056122, result: 301,730274
Generator-Thread 98: Source 94,326678 140,269131 0,779019 (base=1,5423)
Integrator-Thread: Task 98 - [94,3267, 140,2691], step: 0,779019, result: 504,516393
Generator-Thread 99: Source 91,651641 180,147437 0,072004 (base=1,3204)
Generator-Thread 100: Source 6,902794 29,217426 0,513535 (base=7,4432)
Integrator-Thread: Task 99 - [91,6516, 180,1474], step: 0,072004, result: 1558,021539
Integrator-Thread: Task 100 - [6,9028, 29,2174], step: 0,513535, result: 31,360074
Generator-Thread finished
Integrator-Thread finished: 100 tasks processed

Фото 19