МИНОБРНАУКИ РОССИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «ВГУ»)**

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

**Численное решение задачи Коши для обыкновенных дифференциальных уравнений 1-го порядка**

**методами Рунге-Кутта**

Отчёт по лабораторной работе

Студент 3 курса                                                 \_\_\_\_\_\_\_ М.О. Курченков

Преподаватель                                                   \_\_\_\_\_\_\_     О.А. Махинова

Воронеж 2023

**Содержание**

[Постановка задачи 3](#_Toc151595658)

[Анализ задачи 4](#_Toc151595659)

Этапы решения [5](#_Toc151595659)

Задача 1 [6](#_Toc151595664)

Тестирование алгоритма [6](#_Toc151595665)

Вычислительный эксперимент11

Задача 217

Тестирование алгоритма17

Вычислительный эксперимент22

[Приложение 1](#_Toc151595664) 25

EulerMethod.java25

RungeKuttMethod[.java](#_Toc151595666) 27

DESolvingMethod[.java](#_Toc151595667) 30

[Приложение 2](#_Toc151595664) 31

EulerMethod.java31

RungeKuttMethod[.java](#_Toc151595666) 34

DESolvingMethod[.java](#_Toc151595667) 37

# Постановка задачи

1. Реализовать классы, производящие вычисление решения задачи Коши следующими методами: Эйлера и Рунге-Кутта;
2. Протестировать корректность работы классов;
3. Провести вычислительные эксперименты;
4. Проанализировать полученные результаты.

# Анализ задачи

Входные данные представлены аналитической функцией правой части (см рис. 1): f(x) ∈ R→R, отрезком разбиения [a, b] ∈ R, n ∈ N — количеством подотрезков разбиения, требуемой точностью ε, и начальным условием .

Рис. 1 — система для задачи 1

В случае с решением системы дифференциальных уравнений 1 порядка входные данные идентичны за исключением того, что вместо функции f и начального условия представлены: функции начальные условия

Рис. 2 — система для задачи 2

**Этапы решения**

Решение представлено нахождением соответсвующих методу интерполирования коэффициентов:

1. Методы Рунге-Кутта требуют нахождения значений решения по следующим формулам:

1.1) Для 2-го порядка:

Рис. 3 — Формулы для метода Рунге-Кутта 2 порядка

1.2) Для 5-го порядка:

Рис. 4 — Формулы для метода Рунге-Кутта 5 порядка

1. Метод Эйлера требует нахождения коэффициентов следующим способом:

Рис. 5 — Формулы для метода Эйлера

# Задача 1

**Тестирование алгоритма**

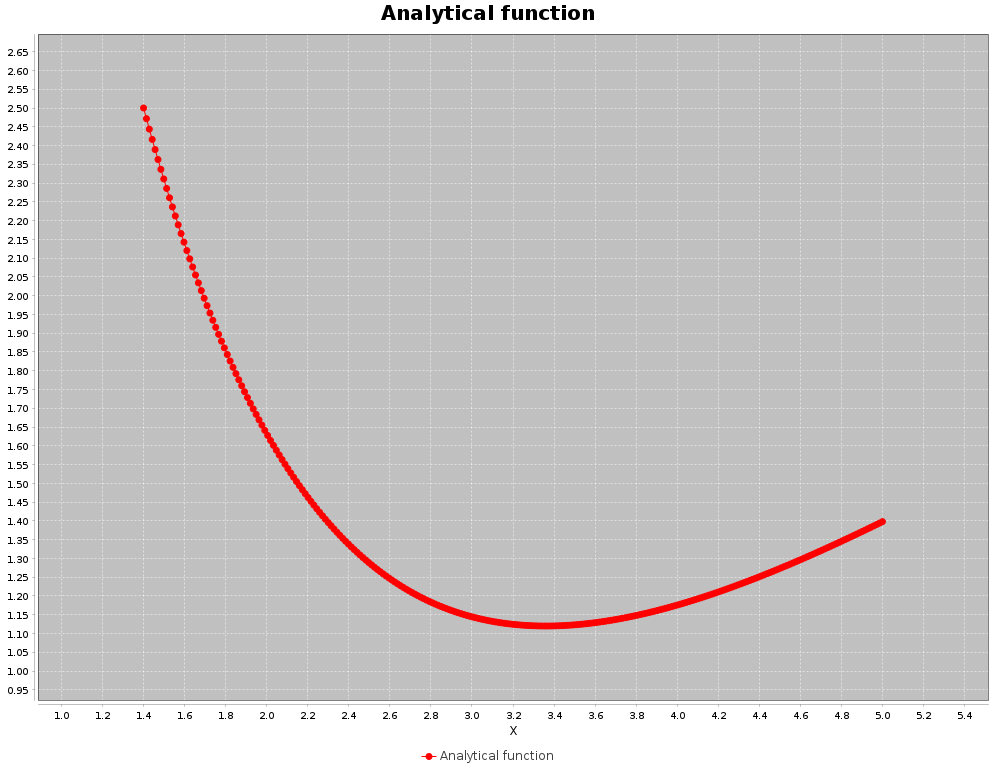
Возьмём тестовую задачу и проверим, действительно ли находится решение, сходное с аналитическим:

Рис. 6 — тестовая задача

Аналитическим решением этой задачи является следующая функция:

Рис. 7 — Аналитическое решение тестовой задачи

Так, графики принимают следующий вид:

Рис. 8 — График тестовой функции

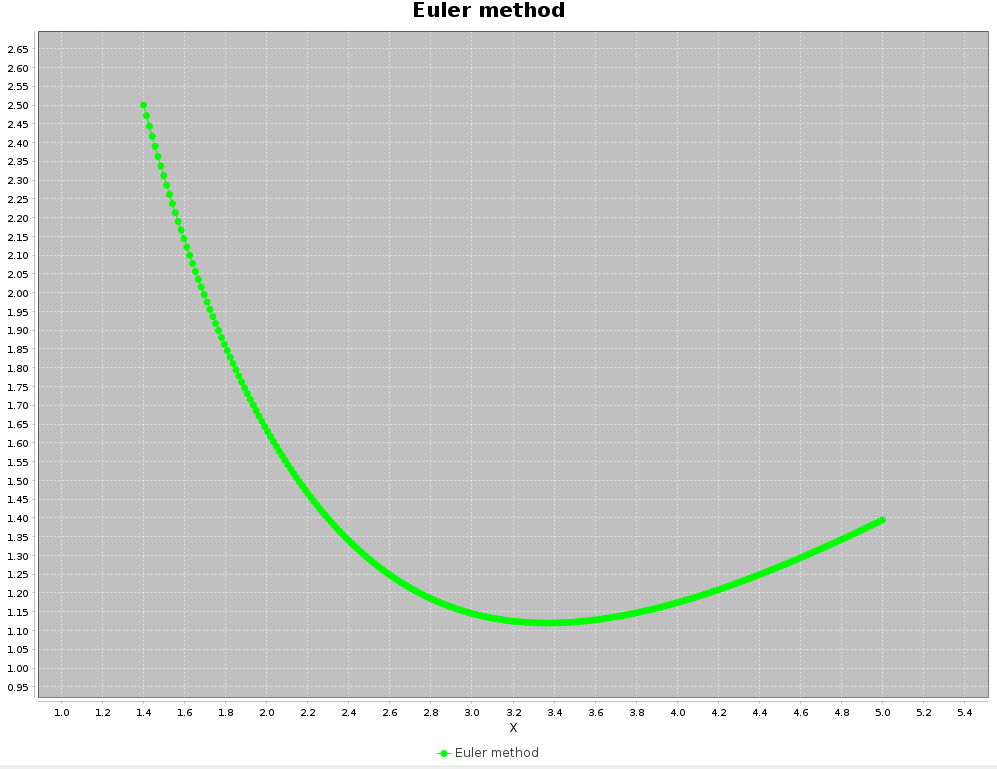
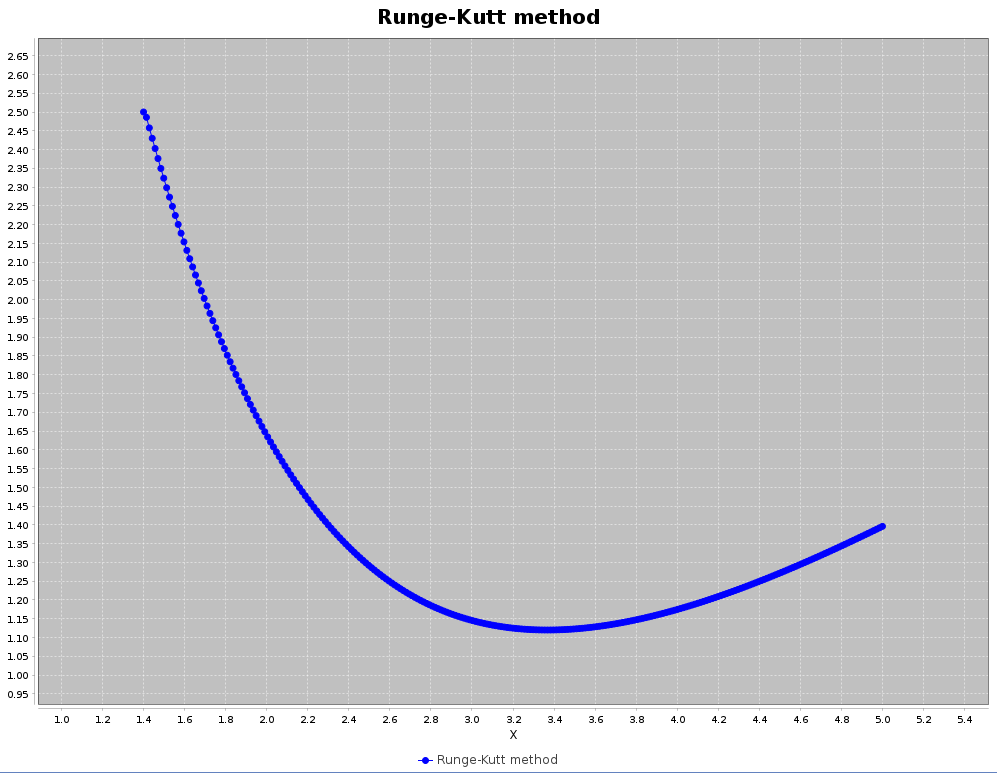


Рис. 9 — График решения методом Эйлера

Рис. 10 — График решения методом Рунге-Кутта

Как видно из графиков, решения, составленные программой согласно обоим методам, соответствуют аналитическому, а написанные классы содержат правильную реализацию методов решения задачи Коши 1 порядка. Удостоверимся в этом, составив таблицу погрешностей (рис. 11). Как можно увидеть, количество разбиений, которые требует метод, корректно отображает работу программы, т. к. порядок метода Рунге-Кутта сильно превосходит порядок метода Эйлера.

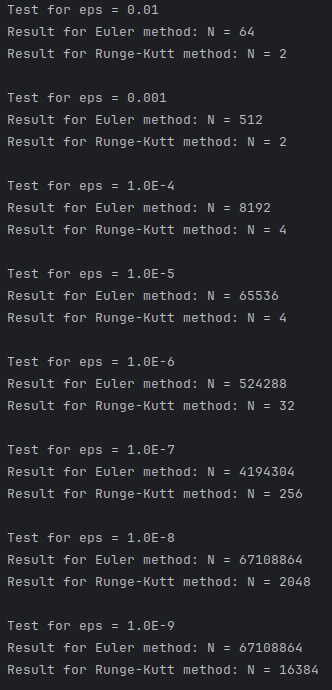
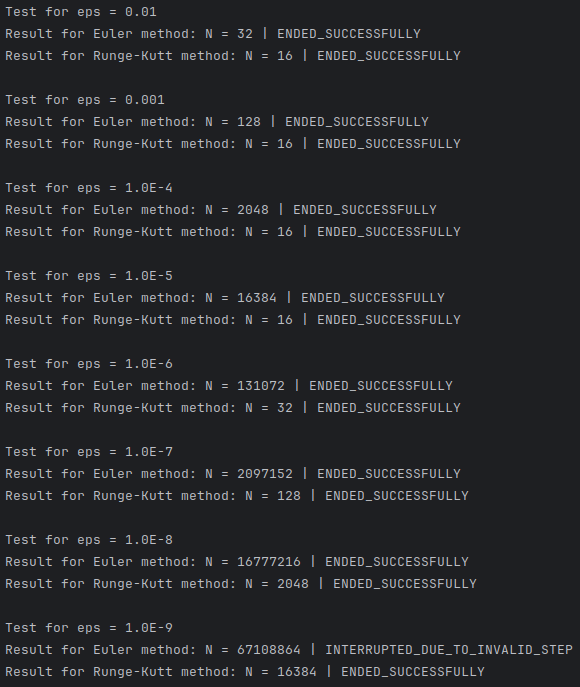


Рис. 11 — Таблица погрешностей тестовой задачи

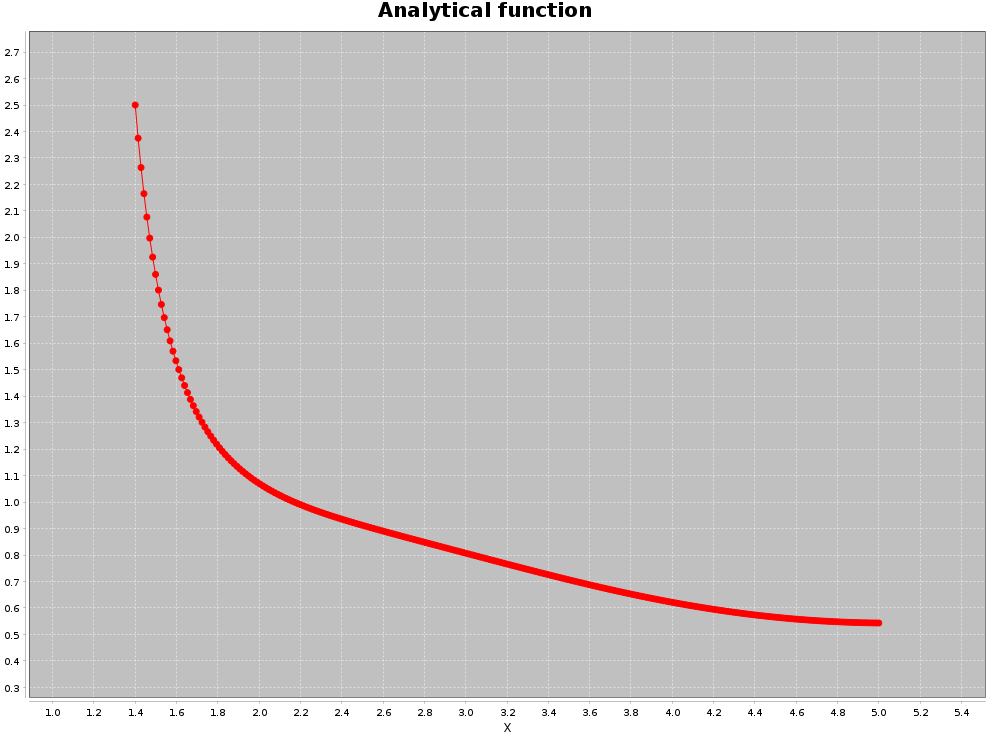
# Вычислительный эксперимент

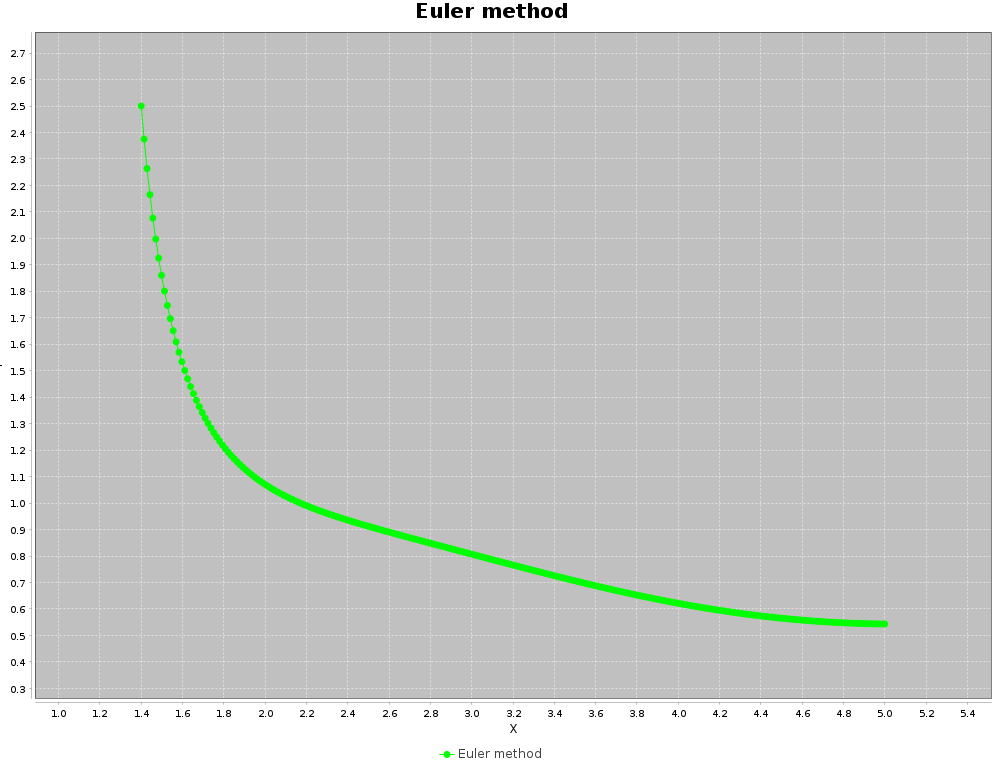
В ходе вычислительного эксперимента возьмём следующую задачу и составим схожую таблицу погрешностей:

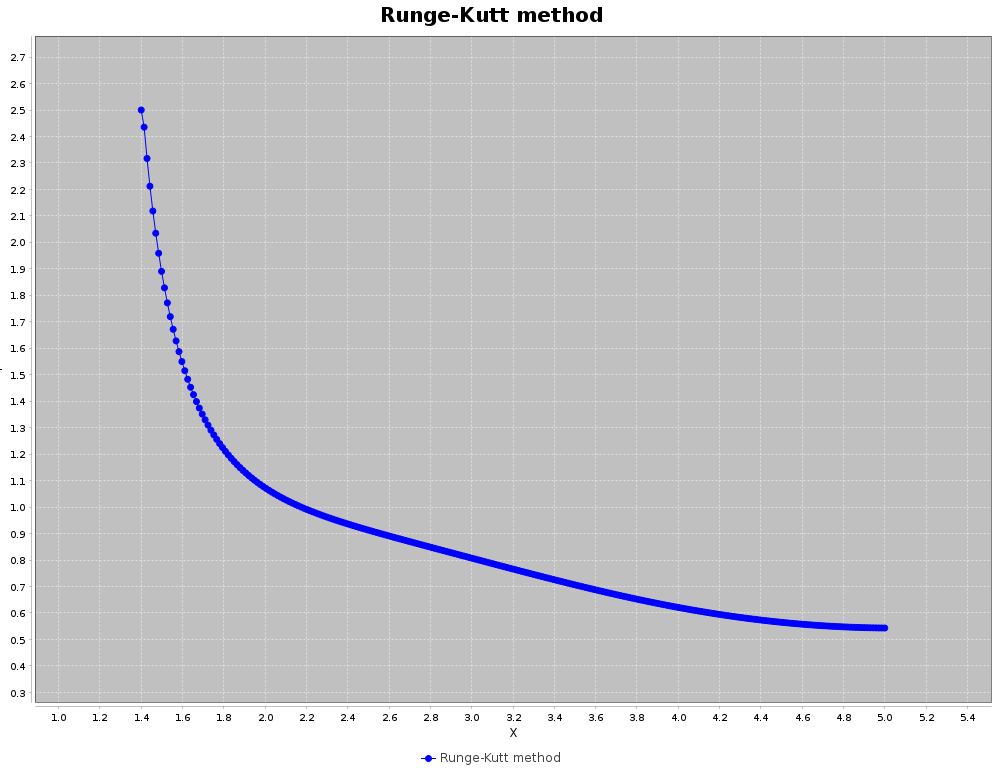
Рис. 12 — Задача для вычислительного эксперимента

Рис. 13 — таблица погрешностей вычислительного эксперимента

Обратим внимание на то, что в случае слишком большого количества итераций значение шага разбиения становится некорретно малым, а вычислительная нагрузка на систему, наоборот, — слишком большим. В таком случае алгоритм прекращает свою работу с соответствующим кодом ошибки. Выведем полученные графики:

Рис. 14 — Аналитическое решение задачи (рис. 12)

Рис. 15 — Решение задачи (рис. 12) методом Эйлера

Рис. 16 — Решение задачи (рис. 12) методом Рунге-Кутта

Зафиксируем сетку с количеством отрезков разбиения, равным 16 и выведем полученные графики:

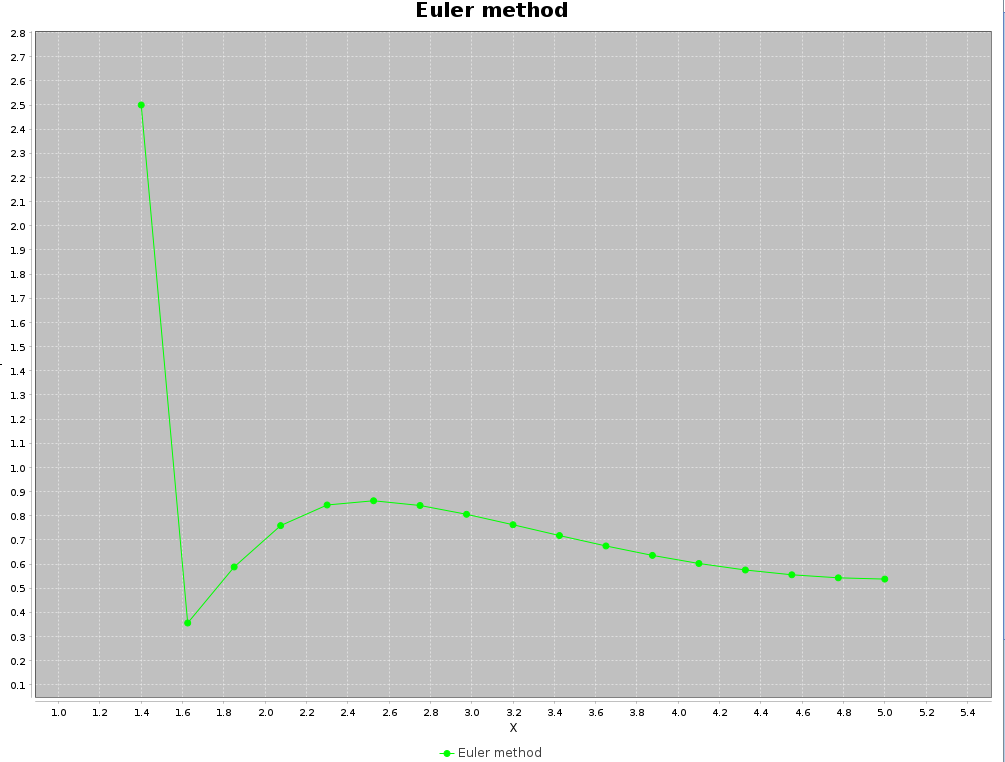
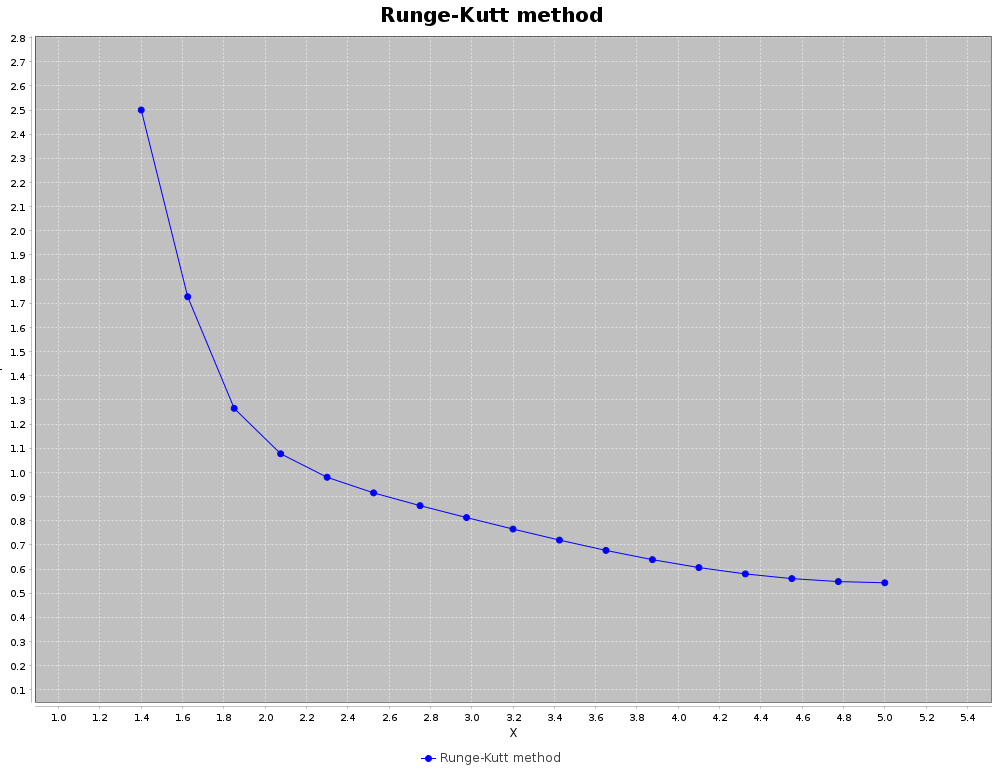


Рис. 17 — Решение задачи (рис. 12) методом Эйлера с начальным количеством отрезков разбиения

Рис. 18 — Решение задачи (рис. 12) методом Рунге-Кутта с начальным количеством отрезков разбиения

Из полученных результатов можно сделать вывод о том, что оба метода на начальной итерации далеки от требуемых результатов, однако метод Рунге-Кутта 5 порядка сильно превосходит метод Эйлера, представляющий собой метод Рунге-Кутта 1 порядка.

# Задача 2

**Тестирование алгоритма**

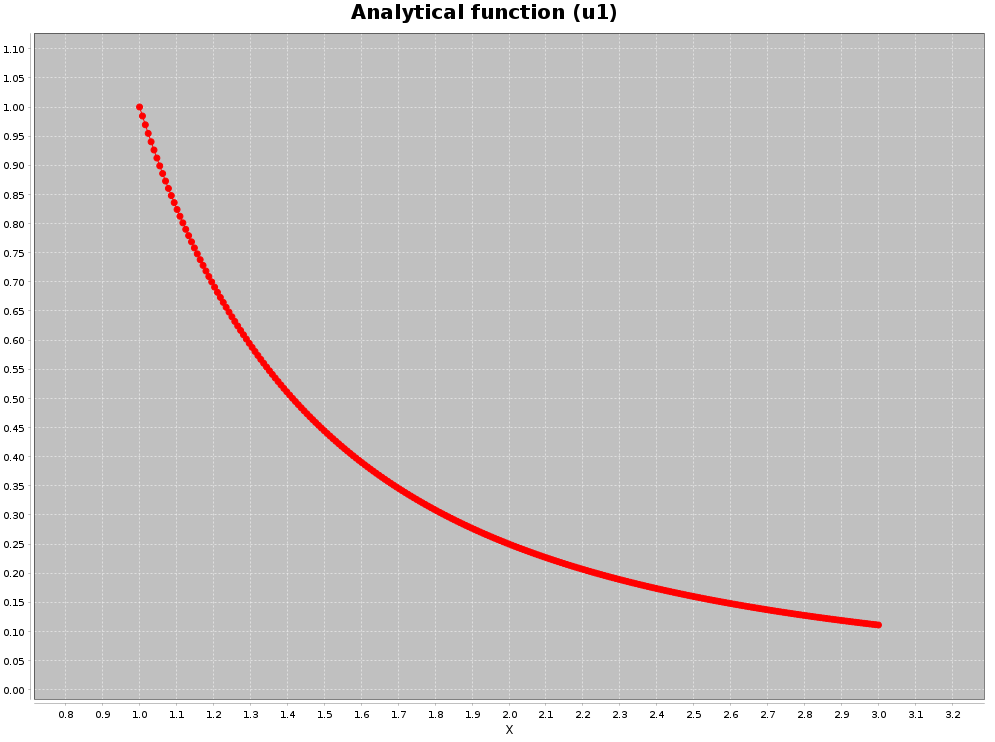
Возьмём тестовую задачу и проверим, действительно ли находится решение, сходное с аналитическим:

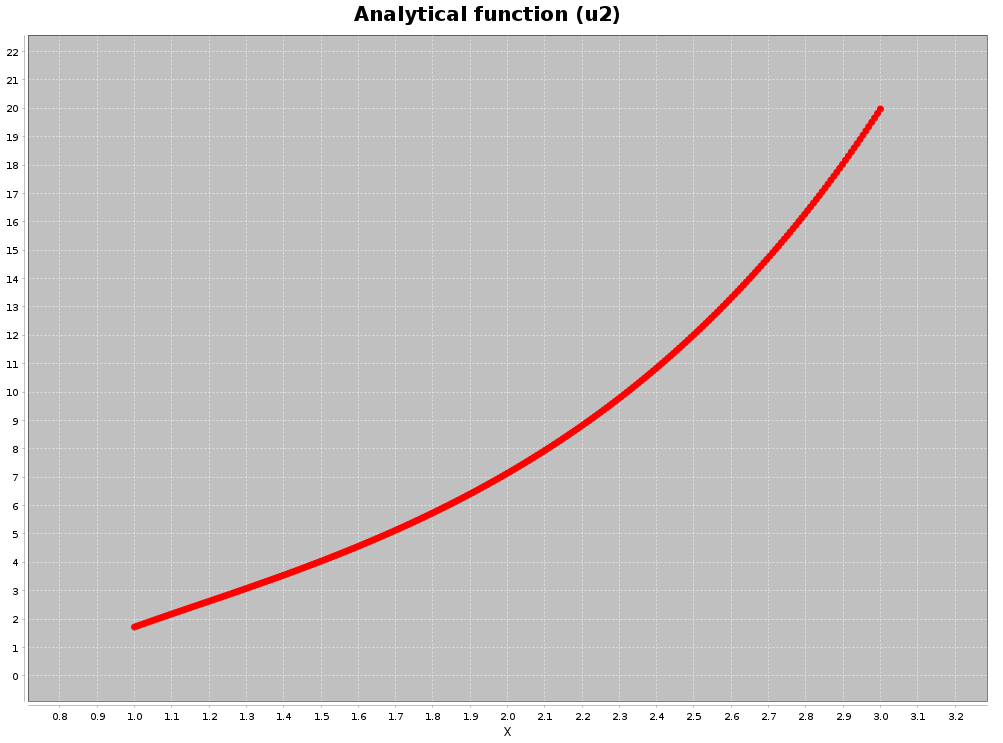
Рис. 19 — Тестовая система

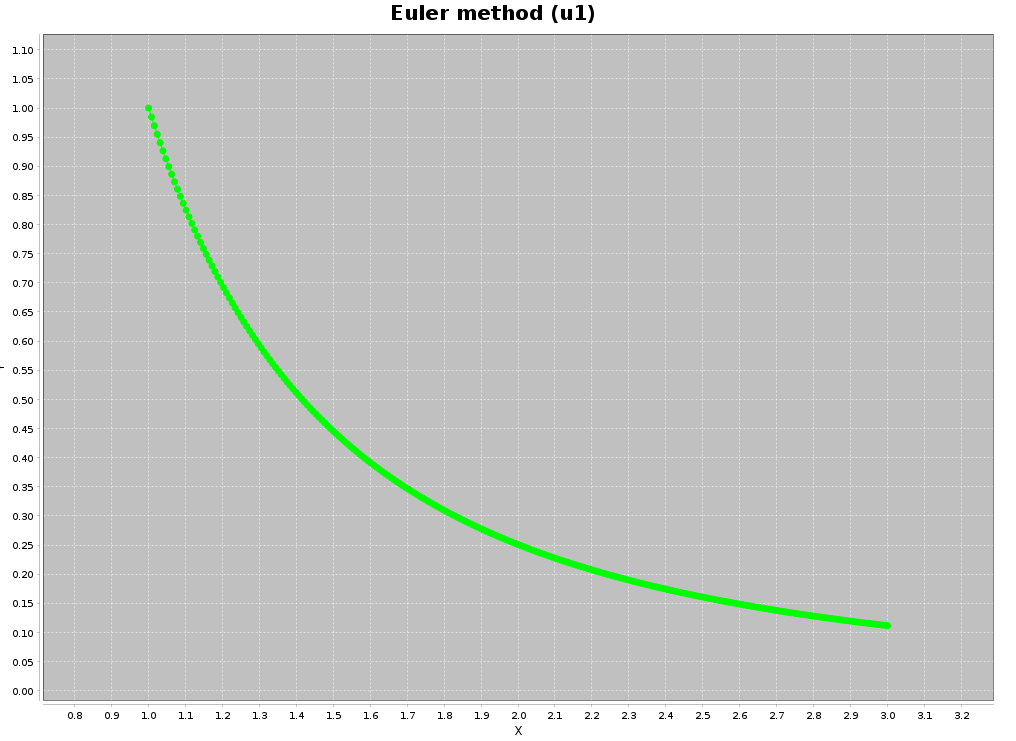
Аналитическим решением этой задачи являются следующие функции:

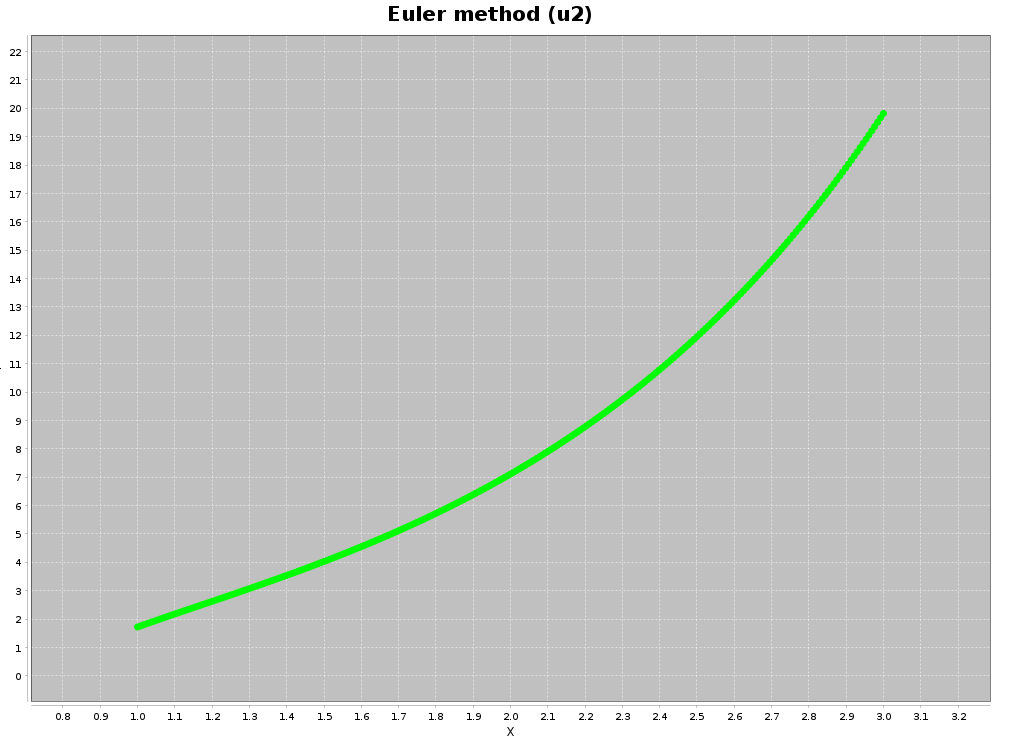
Рис. 20— Аналитическое решение тестовой системы

Так, графики принимают следующий вид:

Рис. 21 — График первой тестовой функции

Рис. 21 — График второй тестовой функции

Рис. 22 — График решения методом Эйлера (первая функция)

Рис. 23 — График решения методом Эйлера (вторая функция)

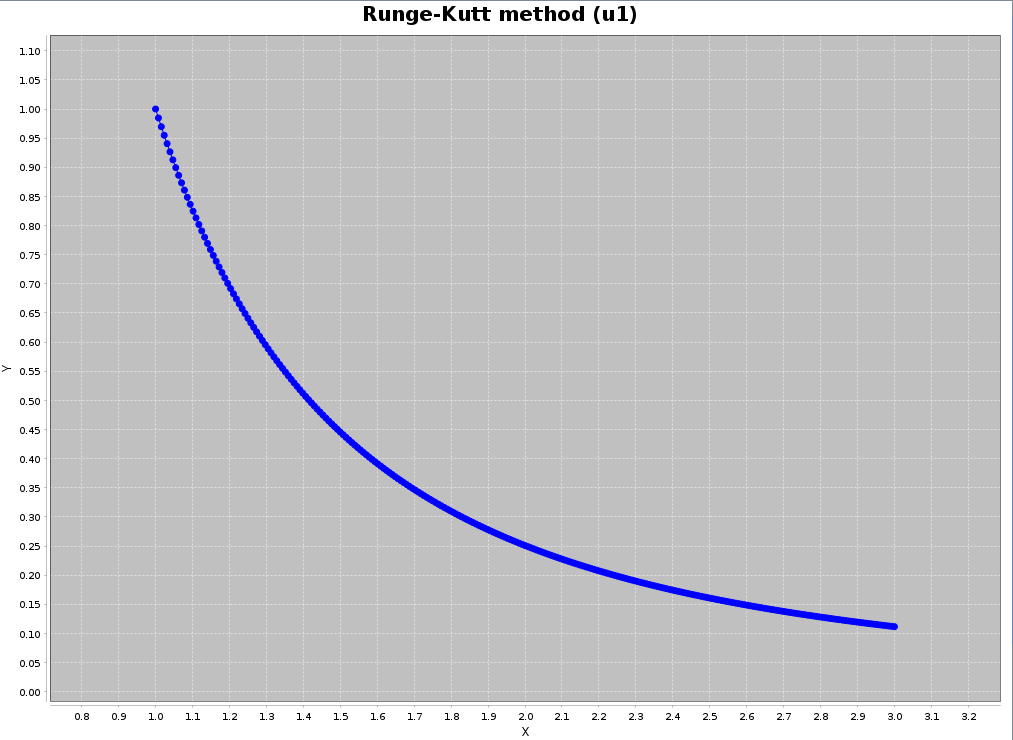


Рис. 23 — График решения методом Рунге-Кутта (первая функция)

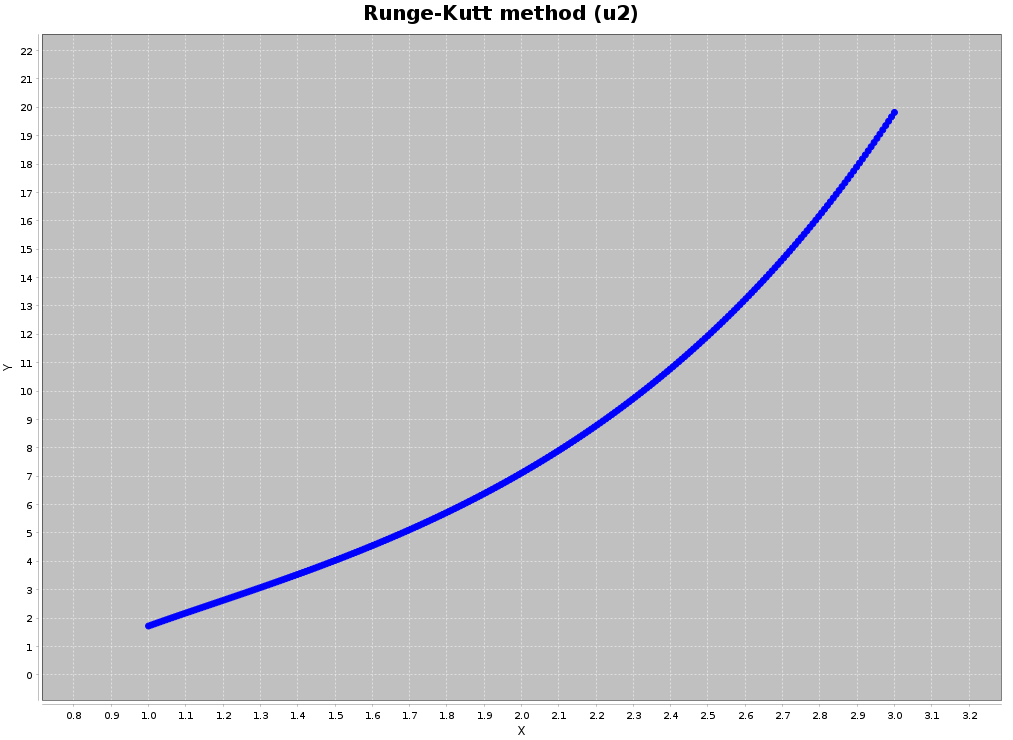
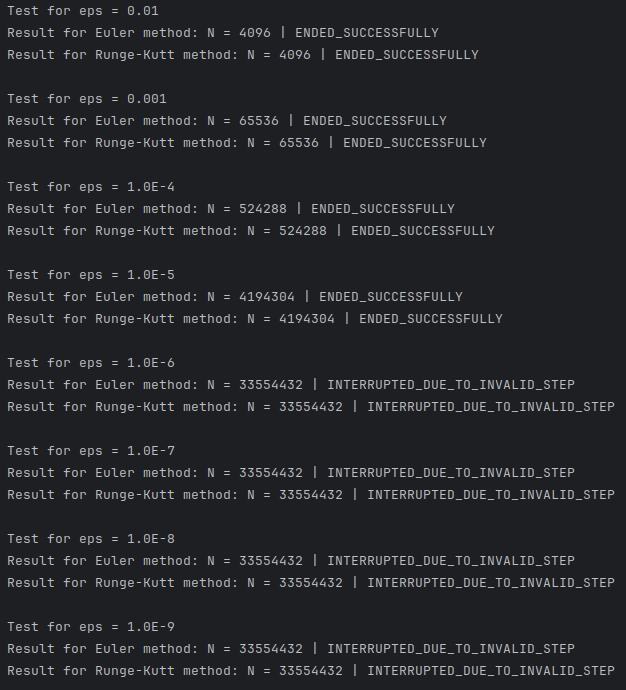


Рис. 24 — График решения методом Рунге-Кутта (вторая функция)

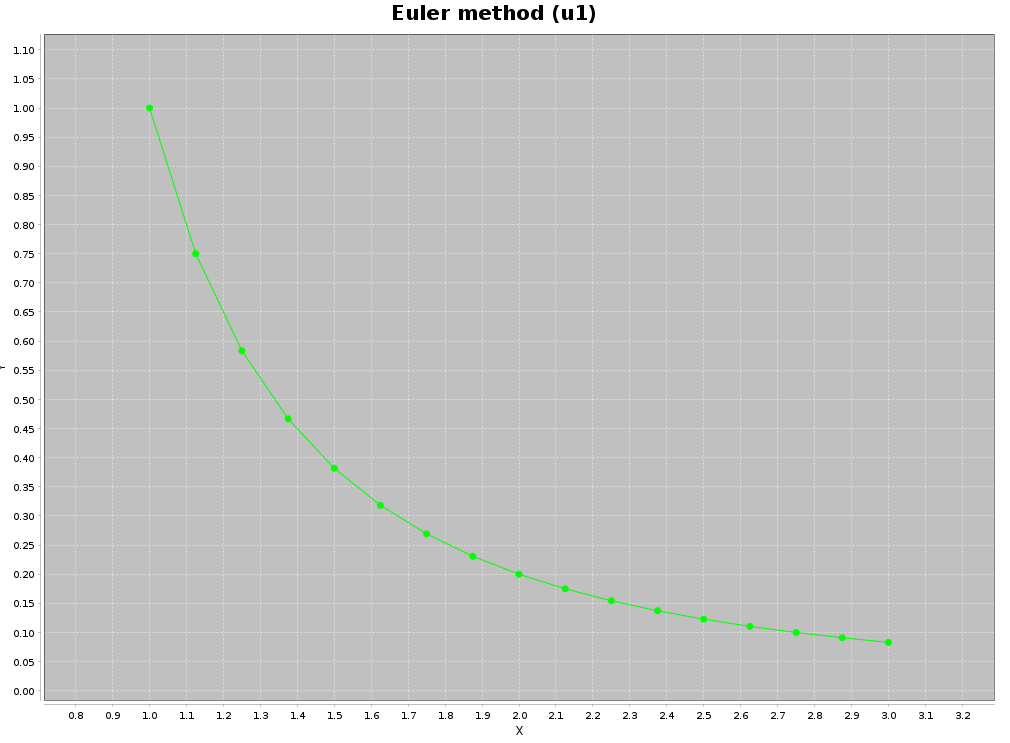
Как видно из графиков, решения, составленные программой согласно обоим методам, соответствуют аналитическому, а написанные классы содержат правильную реализацию методов решения задачи Коши 1 порядка, представленной в виде системы. Удостоверимся в этом, составив таблицу погрешностей (рис. 25). Как можно увидеть, количество разбиений, которые требует метод, корректно отображает работу программы, т. к. реализованный в данной задаче порядок метода Рунге-Кутта лишь на 1 больше, чем метод Эйлера. Помимо этого, в отличие от задачи 1, система требует больших вычислительных мощностей, поэтому и количество итераций тоже больше.

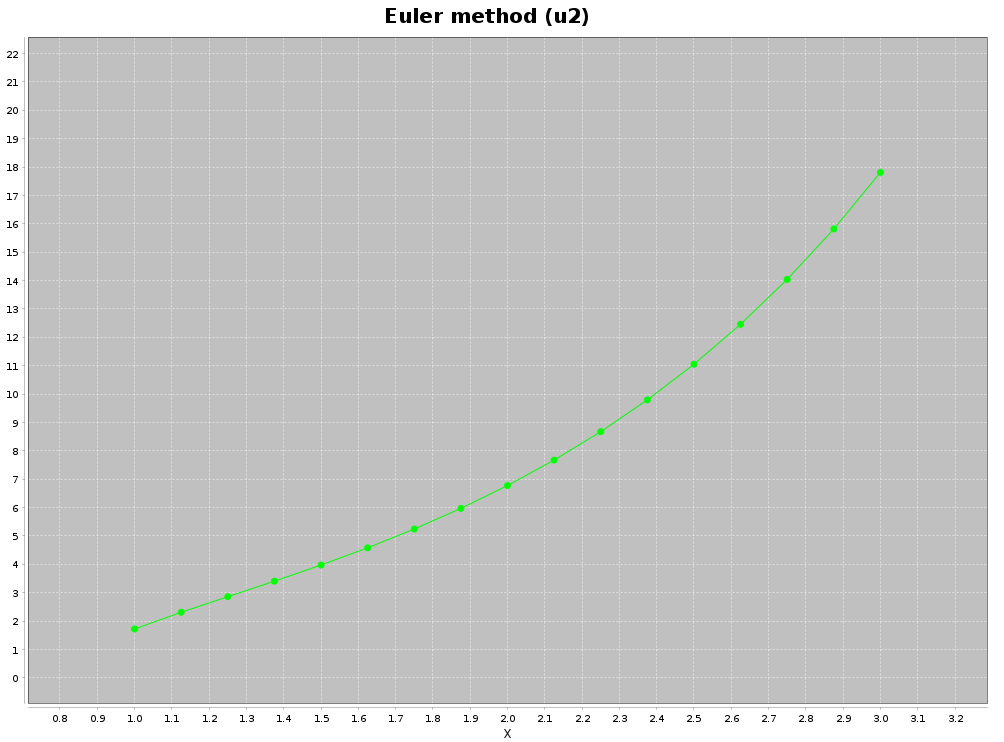
Рис. 25 — Таблица погрешностей для тестовой системы

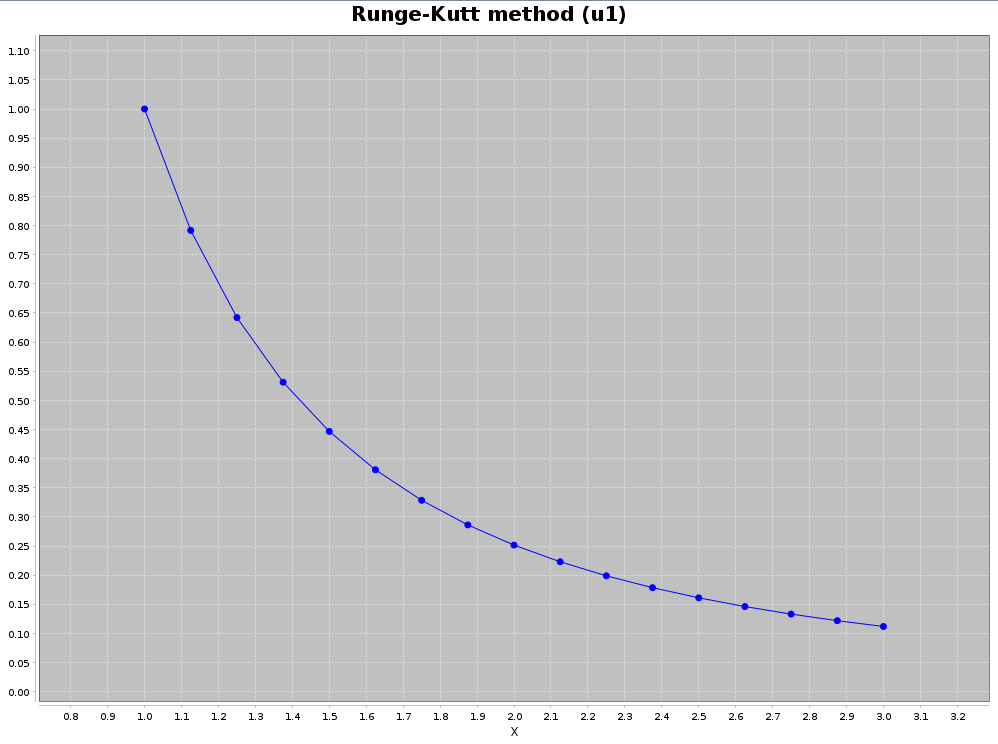
# Вычислительный эксперимент

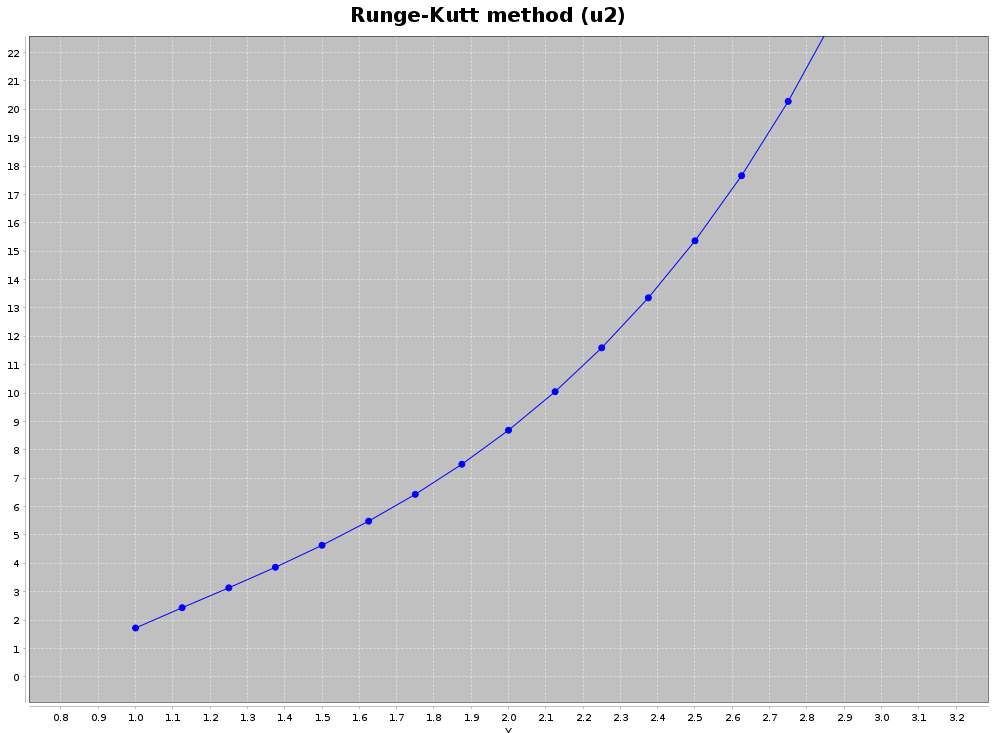
В ходе вычислительного эксперимента будем использовать ту же систему (рис. 19).

Рассмотрим рис. 25. Имеем, что уже при погрешности равной в отличие от случая, исследованного в задаче 1, происходит программное прерывание алогритма. Это связано с высокой вычислительной нагрузкой на алгоритм: в случае системы она больше. Так, достигается слишком маленький шаг и программа прерывается. Сравним графики функций на первой (N = 16) и последней итерации увеличения разбиения:

Рис. 26 — Решение задачи (рис. 19) методом Эйлера с начальным количеством отрезков разбиения (первая функция)

Рис. 27 — Решение задачи (рис. 19) методом Эйлера с начальным количеством отрезков разбиения (вторая функция)

Рис. 28 — Решение задачи (рис. 19) методом Рунге-Кутта с начальным количеством отрезков разбиения (первая функция)

Рис. 29 — Решение задачи (рис. 19) методом Рунге-Кутта с начальным количеством отрезков разбиения (вторая функция)

Полученные графики говорят о том, что оба метода на начальной итерации далеки от требуемых результатов, притом метод Рунге-Кутта 2 порядка и метод Эйлера, представляющий собой метод Рунге-Кутта 1 порядка, выдают сравнимые погрешности.

# **Приложение 1**

**EulerMethod.java**

package lab.numeric.methods.core.models.methods.impl;  
  
import lab.numeric.methods.core.models.Function;  
import lab.numeric.methods.core.models.Section;  
import lab.numeric.methods.core.models.enums.SeparationType;  
import lab.numeric.methods.core.models.methods.DESolvingMethod;  
  
public class EulerMethod extends DESolvingMethod {  
  
 @Override  
 protected void calculateNextY(int i) {  
 y[i + 1] = y[i] + h \* f.calculateExpression(x[i], y[i]);  
 }  
  
 public EulerMethod(Function f, double[] args, double y0, double eps) {  
 this.eps = eps;  
 double previousY = Double.*MAX\_VALUE*;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = y == null ? eps : Math.*abs*(y[y.length - 1] - previousY);  
 previousY = y == null ? Double.*MAX\_VALUE* : y[y.length - 1];  
  
 this.f = f;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 y = new double[x.length];  
  
 y[0] = y0;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*abs*(y[y.length - 1] - previousY);  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (Math.*abs*(y[y.length - 1] - previousY) <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
  
 */\*\**  
 *\* Конструктор для тестировки*  
 *\*/*  
public EulerMethod(Function f, double[] args, double y0, double eps, double yPrecise) {  
 this.eps = eps;  
 double previousY = Double.*MAX\_VALUE*;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = y == null ? eps : Math.*abs*(y[y.length - 1] - previousY);  
 previousY = y == null ? Double.*MAX\_VALUE* : y[y.length - 1];  
  
 this.f = f;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 y = new double[x.length];  
  
 y[0] = y0;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*abs*(y[y.length - 1] - yPrecise);  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (Math.*abs*(y[y.length - 1] - previousY) <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
}

**RungeKuttMethod.java**

package lab.numeric.methods.core.models.methods.impl;  
  
import lab.numeric.methods.core.models.Function;  
import lab.numeric.methods.core.models.Section;  
import lab.numeric.methods.core.models.enums.SeparationType;  
import lab.numeric.methods.core.models.methods.DESolvingMethod;  
  
public class RungeKuttMethod extends DESolvingMethod {  
  
 @Override  
 protected void calculateNextY(int i) {  
  
 var k1 = h \* f.calculateExpression(  
 x[i],  
 y[i]  
 );  
 var k2 = h \* f.calculateExpression(  
 x[i] + 0.25 \* h,  
 y[i] + 0.25 \* k1  
 );  
 var k3 = h \* f.calculateExpression(  
 x[i] + 0.375 \* h,  
 y[i] + 3.0 / 32 \* k1 + 9.0 / 32 \* k2  
 );  
 var k4 = h \* f.calculateExpression(  
 x[i] + 12.0 / 13 \* h,  
 y[i] + 1932.0 / 2197 \* k1 - 7200.0 / 2197 \* k2 + 7296.0 / 2197 \* k3  
 );  
 var k5 = h \* f.calculateExpression(  
 x[i] + h,  
 y[i] + 439.0 / 216 \* k1 - 8 \* k2 + 3680.0 / 513 \* k3  
 - 815.0 / 4104 \* k4  
 );  
 var k6 = h \* f.calculateExpression(  
 x[i] + 0.5 \* h,  
 y[i] - 8.0 / 27 \* k1 + 2 \* k2 - 3544.0 / 2565 \* k3  
 + 1859.0 / 4104 \* k4 - 0.275 \* k5  
 );  
  
 var deltaYn = 16.0 / 135 \* k1 + 6656.0 / 12825 \* k3 + 28561.0 / 56430 \* k4  
 - 0.18 \* k5 + 2.0 / 55 \* k6;  
  
 y[i + 1] = y[i] + deltaYn;  
 }  
  
 public RungeKuttMethod(Function f, double[] args, double y0, double eps) {  
 this.eps = eps;  
 double previousY = Double.*MAX\_VALUE*;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = y == null ? eps : Math.*abs*(y[y.length - 1] - previousY);  
 previousY = y == null ? Double.*MAX\_VALUE* : y[y.length - 1];  
  
 this.f = f;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 y = new double[x.length];  
  
 y[0] = y0;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*abs*(y[y.length - 1] - previousY) / 31;  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (diff <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
  
 */\*\**  
 *\* Конструктор для тестировки*  
 *\*/*  
public RungeKuttMethod(Function f, double[] args, double y0, double eps, double yPrecise) {  
 this.eps = eps;  
 double previousY = Double.*MAX\_VALUE*;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = y == null ? eps : Math.*abs*(y[y.length - 1] - previousY);  
 previousY = y == null ? Double.*MAX\_VALUE* : y[y.length - 1];  
  
 this.f = f;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 y = new double[x.length];  
  
 y[0] = y0;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*abs*(y[y.length - 1] - yPrecise) / 31;  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (Math.*abs*(y[y.length - 1] - previousY) <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
}

**DESolvingMethod.java**

package lab.numeric.methods.core.models.methods;  
  
import lab.numeric.methods.core.models.Function;  
import lombok.Getter;  
import lombok.NoArgsConstructor;  
  
@NoArgsConstructor  
@Getter  
public abstract class DESolvingMethod {  
  
 protected static final int *SPLIT\_RESTRICTION* = 67108864;  
  
 protected Function f;  
 protected double[] y;  
 protected double[] x;  
 protected double h;  
 protected double eps;  
 protected ErrorIndicator errorIndicator;  
  
 protected abstract void calculateNextY(int i);  
  
 protected enum ErrorIndicator {  
 *ENDED\_SUCCESSFULLY*,  
 *INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*,  
 *INTERRUPTED\_DUE\_TO\_INVALID\_STEP*,  
 *UNKNOWN*  
}  
}

# **Приложение 2**

**EulerMethod.java**

package lab.numeric.methods.core.models.methods.impl;  
  
import lab.numeric.methods.core.models.Function;  
import lab.numeric.methods.core.models.Section;  
import lab.numeric.methods.core.models.enums.SeparationType;  
import lab.numeric.methods.core.models.methods.DESolvingMethod;  
  
public class EulerMethod extends DESolvingMethod {  
  
 @Override  
 protected void calculateNextY(int i) {  
 u1[i + 1] = u1[i] + h \* f1.calculateExpression(x[i], u1[i], u2[i]);  
 u2[i + 1] = u2[i] + h \* f2.calculateExpression(x[i], u1[i], u2[i]);  
 }  
  
 public EulerMethod(Function f1, Function f2, double[] args, double u10, double u20, double eps) {  
 this.eps = eps;  
 double previousU1 = Double.*MAX\_VALUE*;  
 double previousU2 = Double.*MAX\_VALUE*;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = u1 == null || u2 == null ?  
 eps : Math.*max*(Math.*abs*(u1[u1.length - 1] - previousU1), Math.*abs*(u2[u2.length - 1] - previousU2));  
 previousU1 = u1 == null ? Double.*MAX\_VALUE* : u1[u1.length - 1];  
 previousU2 = u2 == null ? Double.*MAX\_VALUE* : u2[u2.length - 1];  
  
 this.f1 = f1;  
 this.f2 = f2;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 u1 = new double[x.length];  
 u2 = new double[x.length];  
  
 u1[0] = u10;  
 u2[0] = u20;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*max*(Math.*abs*(u1[u1.length - 1] - previousU1), Math.*abs*(u2[u2.length - 1] - previousU2));  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (diff <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
  
 */\*\**  
 *\* Конструктор для тестировки*  
 *\*/*  
public EulerMethod(  
 Function f1,  
 Function f2,  
 double[] args,  
 double u10,  
 double u20,  
 double eps,  
 double u1Precise,  
 double u2Precise  
 ) {  
 this.eps = eps;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = u1 == null || u2 == null ?  
 eps : Math.*max*(Math.*abs*(u1[u1.length - 1] - u1Precise), Math.*abs*(u2[u2.length - 1] - u2Precise));  
  
 this.f1 = f1;  
 this.f2 = f2;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 u1 = new double[x.length];  
 u2 = new double[x.length];  
  
 u1[0] = u10;  
 u2[0] = u20;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*max*(Math.*abs*(u1[u1.length - 1] - u1Precise), Math.*abs*(u2[u2.length - 1] - u2Precise));  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (diff <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
}

**RungeKuttMethod.java**

package lab.numeric.methods.core.models.methods.impl;  
  
import lab.numeric.methods.core.models.Function;  
import lab.numeric.methods.core.models.Section;  
import lab.numeric.methods.core.models.enums.SeparationType;  
import lab.numeric.methods.core.models.methods.DESolvingMethod;  
  
public class RungeKuttMethod extends DESolvingMethod {  
  
 @Override  
 protected void calculateNextY(int i) {  
  
 var k11 = h \* f1.calculateExpression(  
 x[i],  
 u1[i],  
 u2[i]  
 );  
 var k12 = h \* f1.calculateExpression(  
 x[i] + h,  
 u1[i] + k11,  
 u2[i] + k11  
 );  
 var k21 = h \* f2.calculateExpression(  
 x[i],  
 u1[i],  
 u2[i]  
 );  
 var k22 = h \* f2.calculateExpression(  
 x[i] + h,  
 u1[i] + k21,  
 u2[i] + k21  
 );  
  
 var deltaU1 = (k11 + k12) / 2;  
 var deltaU2 = (k21 + k22) / 2;  
  
 u1[i + 1] = u1[i] + deltaU1;  
 u2[i + 1] = u2[i] + deltaU2;  
 }  
  
 public RungeKuttMethod(Function f1, Function f2, double[] args, double u10, double u20, double eps) {  
 this.eps = eps;  
 double previousU1 = Double.*MAX\_VALUE*;  
 double previousU2 = Double.*MAX\_VALUE*;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = u1 == null || u2 == null ? eps :  
 Math.*max*(Math.*abs*(u1[u1.length - 1] - previousU1) / 3, Math.*abs*(u2[u2.length - 1] - previousU2) / 3);  
 previousU1 = u1 == null ? Double.*MAX\_VALUE* : u1[u1.length - 1];  
 previousU2 = u2 == null ? Double.*MAX\_VALUE* : u2[u2.length - 1];  
  
 this.f1 = f1;  
 this.f2 = f2;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 u1 = new double[x.length];  
 u2 = new double[x.length];  
  
 u1[0] = u10;  
 u2[0] = u20;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*max*(Math.*abs*(u1[u1.length - 1] - previousU1) / 3, Math.*abs*(u2[u2.length - 1] - previousU2) / 3);  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (diff <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
  
 */\*\**  
 *\* Конструктор для тестировки*  
 *\*/*  
public RungeKuttMethod(  
 Function f1,  
 Function f2,  
 double[] args,  
 double u10,  
 double u20,  
 double eps,  
 double u1Precise,  
 double u2Precise  
 ) {  
 this.eps = eps;  
 double previousDiff;  
 double diff;  
 do {  
 previousDiff = u1 == null || u2 == null ? eps :  
 Math.*max*(Math.*abs*(u1[u1.length - 1] - u1Precise) / 3, Math.*abs*(u2[u2.length - 1] - u2Precise) / 3);  
  
 this.f1 = f1;  
 this.f2 = f2;  
 x = x == null ? args : new Section(  
 x[0],  
 x[x.length - 1],  
 (x.length - 1) \* 2,  
 SeparationType.*UNIFORM*  
).getSeparation();  
 h = x[1] - x[0];  
  
 u1 = new double[x.length];  
 u2 = new double[x.length];  
  
 u1[0] = u10;  
 u2[0] = u20;  
 for (int i = 0; i < x.length - 1; ++i) {  
 calculateNextY(i);  
 }  
 diff = Math.*max*(Math.*abs*(u1[u1.length - 1] - u1Precise) / 3, Math.*abs*(u2[u2.length - 1] - u2Precise) / 3);  
 } while (diff > eps && x.length < *SPLIT\_RESTRICTION* && diff != previousDiff);  
  
 if (diff <= eps) {  
 errorIndicator = ErrorIndicator.*ENDED\_SUCCESSFULLY*;  
 } else if (x.length >= *SPLIT\_RESTRICTION*) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_INVALID\_STEP*;  
 } else if (diff == previousDiff) {  
 errorIndicator = ErrorIndicator.*INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*;  
 } else {  
 errorIndicator = ErrorIndicator.*UNKNOWN*;  
 }  
 }  
}

**DESolvingMethod.java**

package lab.numeric.methods.core.models.methods;  
  
import lab.numeric.methods.core.models.Function;  
import lombok.Getter;  
import lombok.NoArgsConstructor;  
  
@NoArgsConstructor  
@Getter  
public abstract class DESolvingMethod {  
  
 protected static final int *SPLIT\_RESTRICTION* = 16;  
  
 protected Function f1;  
 protected Function f2;  
 protected double[] u1;  
 protected double[] u2;  
 protected double[] x;  
 protected double h;  
 protected double eps;  
 protected ErrorIndicator errorIndicator;  
  
 protected abstract void calculateNextY(int i);  
  
 protected enum ErrorIndicator {  
 *ENDED\_SUCCESSFULLY*,  
 *INTERRUPTED\_DUE\_TO\_CONSTANT\_DIFFERENCE*,  
 *INTERRUPTED\_DUE\_TO\_INVALID\_STEP*,  
 *UNKNOWN*  
}  
}