Описание метода, расчетные формулы

Метод Адамса для решения задачи Коши

Для решения уравнения $y' = f(x, y)$ по *методу Адамса*, исходя из началных условий $y(x_0) = y_0$ находим *методом Рунге-Кутта* следующие значения $y(x)$:

$$y_1 = y(x_1) = y(x_0 + h), \quad y_2 = y(x_2) = y(x_0 + 2h), \quad y_3 = y(x_3) = y(x_0 + 3h)$$

где h – *шаг вычисления*, что $h^4 < \varepsilon$(*точность вычисления*).

Находим далее величины

$$q_0 = h \cdot y_0' = h \cdot f(x_0, y_0), \quad q_1 = h \cdot y_1' = h \cdot f(x_1, y_1),$$

$$q_2 = h \cdot y_2' = h \cdot f(x_2, y_2), \quad q_3 = h \cdot y_3' = h \cdot f(x_3, y_3)$$

Метод Адамса заключается в предложении *диагональной таблицы разностей* с помощью *формулы Адамса.*

$$\tilde{y}_i^{\textit{прог}} = \frac{1}{24}\left(55q_i - 59q_{i-1} + 37q_{i-2} - 9q_{i-3}\right),$$

$$\tilde{y}_i^{\textit{прог}} = h \cdot f(x_{i+1}, \tilde{y}_i^{\textit{прог}}),$$

$$\Delta y_i = \frac{1}{24}\left(9\tilde{y}_i^{\textit{прог}} + 19q_i - 5q_{i-1} + q_{i-2}\right).$$

$$\boxed{x_{i+1} = x_0 + (i+1) \cdot h, \quad y_{i+1} = y_i + \Delta y_i}$$

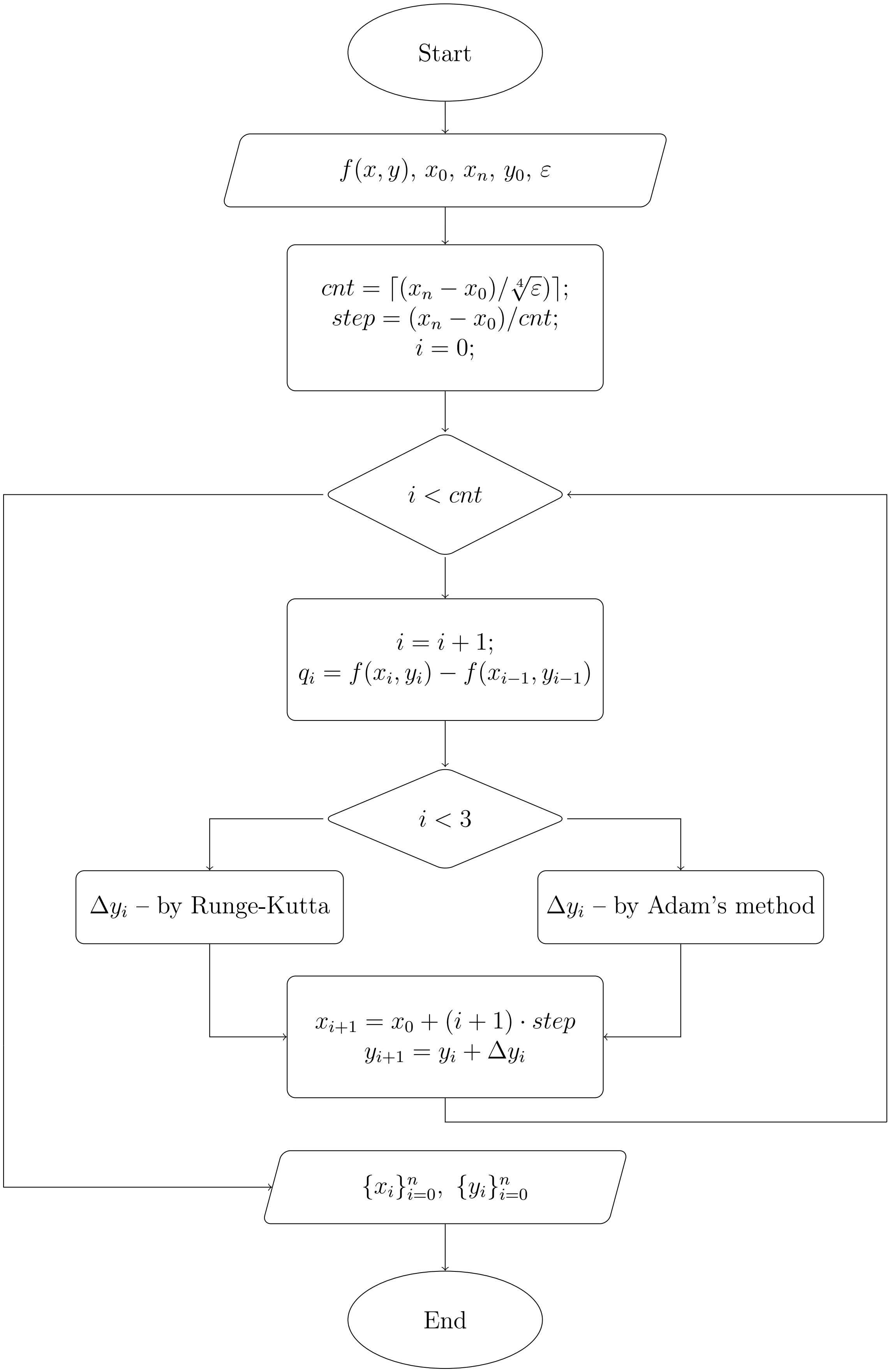
Формула метода Рунге-Кутта четвертого порядка

$$\Delta y_i = \frac{1}{6}\left(k_1^{(i)} + 2k_2^{(i)} + 2k_3^{(i)} + k_4^{(i)}\right), \; i = 0, 1, 2, .., n,$$

$$k_1^{(i)} = f(x_i, y_i) \cdot h, \qquad k_3^{(i)} = f(x_i + \frac{h}{2}, y_i + \frac{k_2^{(i)}}{2}) \cdot h,$$

$$k_2^{(i)} = f(x_i + \frac{h}{2}, y_i + \frac{k_1^{(i)}}{2}) \cdot h, \quad k_4^{(i)} = f(x_i + h, y_i + k_3^{(i)}) \cdot h$$

Блок-схема численного метода



Листинг реализованного численного метода программы

```java
import lombok.NonNull;
import ru.ifmo.cmath.utils.Point;

import java.util.ArrayList;
import java.util.List;
import static java.lang.Math.*;

public class DifferentialEquationSolver {
    public List<Point> solveByAdams(@NonNull Function function, @NonNull
                                    Point initialPoint, double end, double epsilon) {

        List<Point> points = new ArrayList<>();
        points.add(initialPoint);

        if (0 >= epsilon || epsilon >= 1) {
            throw new IllegalArgumentException("accuracy: allowed only decimal point
            ↪  numbers between 0 and 1");
        }
        int count = (int) ceil((end - initialPoint.getX()) / pow(epsilon, 0.25));
        double step = (end - initialPoint.getX()) / count;

        List<Double> values = new ArrayList<>();
        for (int idx = 0; idx < count; idx++) {
            /* Calculate function value at every point */
            values.add(function.apply(points.get(idx).getX(), points.get(idx).getY()));
            if (idx < 3)
                points.add(pointByRungeKutta(function, points, idx + 1, step));
            else
                points.add(pointByAdams(function, values, points, idx + 1, step));
        }
        return points;
    }

    private Point pointByRungeKutta(Function f, List<Point> pnts, int i, double h) {
        Point point = pnts.get(i - 1);
        double delta, k[] = new double[4];

        k[0] = h * func.apply(point.getX(), point.getX());
        k[1] = h * func.apply(point.getX() + 0.5 * h, point.getY() + 0.5 * k[0]);
        k[2] = h * func.apply(point.getX() + 0.5 * h, point.getY() + 0.5 * k[1]);
        k[3] = h * func.apply(point.getX() + h, point.getY() + k[2]);

        delta = (k[0] + 2 * k[1] + 2 * k[2] + k[3]) / 6;

        return new Point(pnts.get(0).getX() + i * h, point.getY() + delta);
    }
    ...
```

```java
    ...
    private Point pointByAdams(Function f, List<Double> values, List<Point> pnts, int i,
    ↪   double h) {
        Point start = pnts.get(0), end = pnts.get(i - 1);

        double delta, q[] = new double[4];

        q[0] = h * values.get(i - 1);
        q[1] = h * values.get(i - 2);
        q[2] = h * values.get(i - 3);
        q[3] = h * values.get(i - 4);

        /* Forecast */
        delta = (55 * q[0] - 59 * q[1] + 37 * q[2] - 9 * q[3]) / 24;
        delta = h * f.apply(start.getX() + i * step, delta);
        /* Correction */
        delta = (9 * delta + 19 * q[0] - 5 * q[1] + q[2]) / 24;

        return new Point(start.getX() + i * h, end.getY() + delta);
    }
}
```

```java
import net.objecthunter.exp4j.Expression;
import net.objecthunter.exp4j.ExpressionBuilder;

public class Function {
    private final Expression expression;

    public Function(String exp, String... variables) {
        this.expression = new ExpressionBuilder(exp).variables(variables).build();
    }

    public Double apply(Double... values) {
        try {
            int index = 0;
            for (String variable : expression.getVariableNames()) {
                this.expression.setVariable(variable, values[index++]);
            }
            return this.expression.evaluate();
        } catch (RuntimeException e) {
            return Double.NaN;
        }
    }
}
```

```java
import ru.ifmo.cmath.utils.Point;

import java.util.List;

public class LagrangianPolynomialBuilder {
    private final StringBuilder lagrangianPolynomial;
    private List<Point> axisData;

    public LagrangianPolynomialBuilder() {
        this.lagrangianPolynomial = new StringBuilder();
    }

    public LagrangianPolynomialBuilder setAxisData(List<Point> axisData) {
        if (axisData.size() < 2) {
            throw new IllegalArgumentException("lagrangian polynomial: points can not be less than 2");
        }
        this.axisData = axisData;
        return this;
    }

    public Function build() {
        if (axisData.isEmpty()) {
            throw new IllegalArgumentException("lagrangian polynomial: experimental data is empty!");
        }
        for (int i = 0; i < axisData.size(); i++) {
            if (axisData.get(i).getY().isNaN() || axisData.get(i).getY().isInfinite()) {
                throw new IllegalArgumentException("lagrangian polynomial: function undefined at x=" + axisData.get(i).getX());
            }
            /* Create a Lagrangian Polynomial */
            lagrangianPolynomial.append("+").append(axisData.get(i).getY())
                    .append("(")
                    .append(lagrangianMultiplier(i))
                    .append(")");
        }
        return new Function(lagrangianPolynomial.toString(), "x");
    }
    ...
```

```java
    ...
    private StringBuilder lagrangianMultiplier(int i) {
        StringBuilder numerator = new StringBuilder();
        Double dominator = 1.0D;

        for (int j = 0; j < axisData.size(); j++) {
            if (i == j) continue;
            /* Create a numerator of lagrangian multiplier */
            numerator.append("(x-").append(axisData.get(j).getX()).append(")");
            /* Calculate a dominator value */
            dominator *= (axisData.get(i).getX() - axisData.get(j).getX());

            if (-1E-9 < dominator && dominator < 1E-9) {
                throw new RuntimeException("lagrangian polynomial: too small steps!");
            }
        }
        return numerator.append("/").append(dominator);
    }
}
```

Примеры и результаты работы программы на разных данных

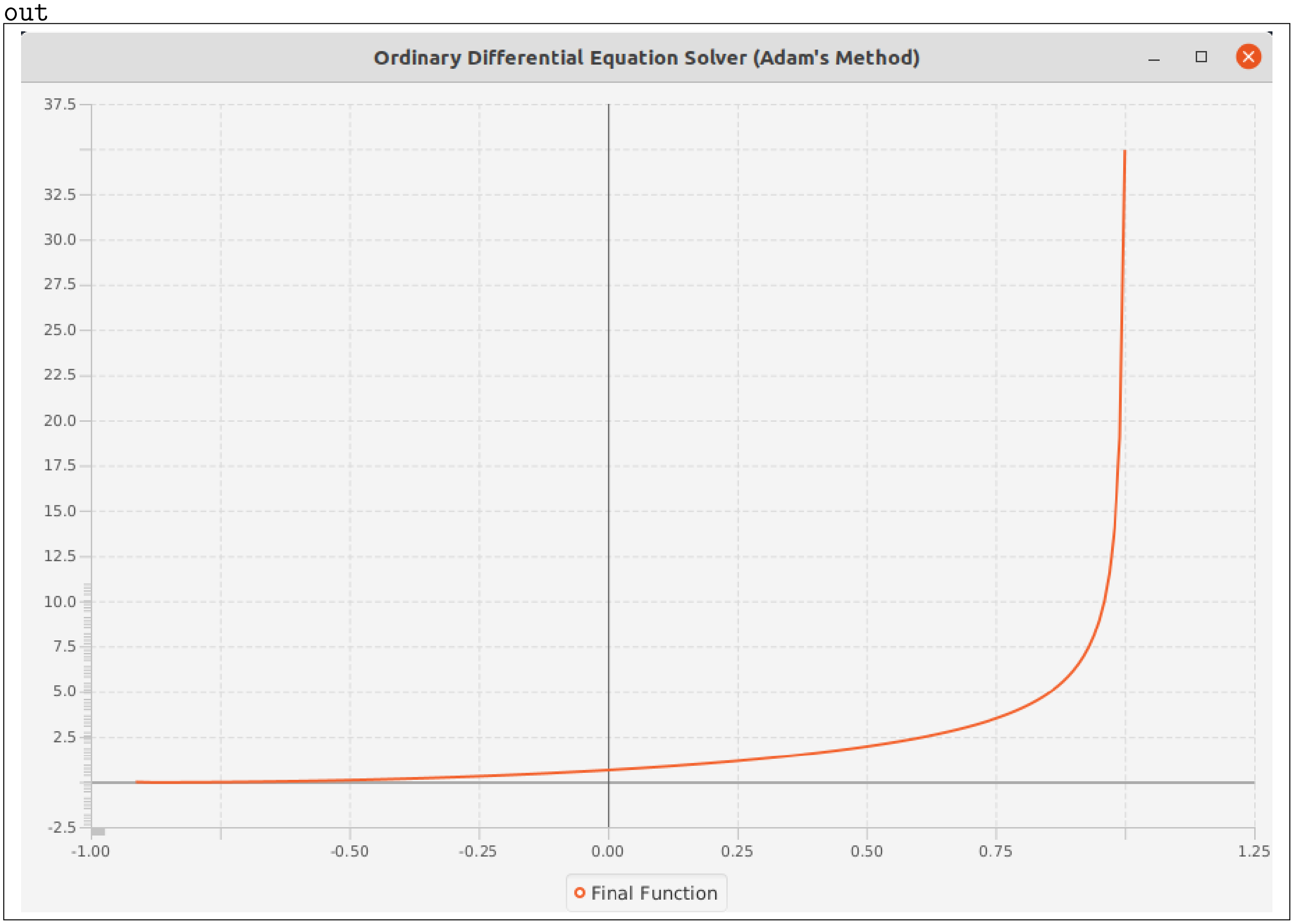
in

differential.properties

```
# ================================
# = ORDINARY DIFFERENTIAL EQUATION
# ================================
# Left hand side of ordinary differential equation
differential.function=1+x+y/(1-x^2)
# The interval, where searches answer of equation
differential.equation.x0=-0.91
differential.equation.xN=1
# Value of function at point x0
differential.equation.y0=0
# Specified accuracy
differential.equation.epsilon=0.00000001

```

out



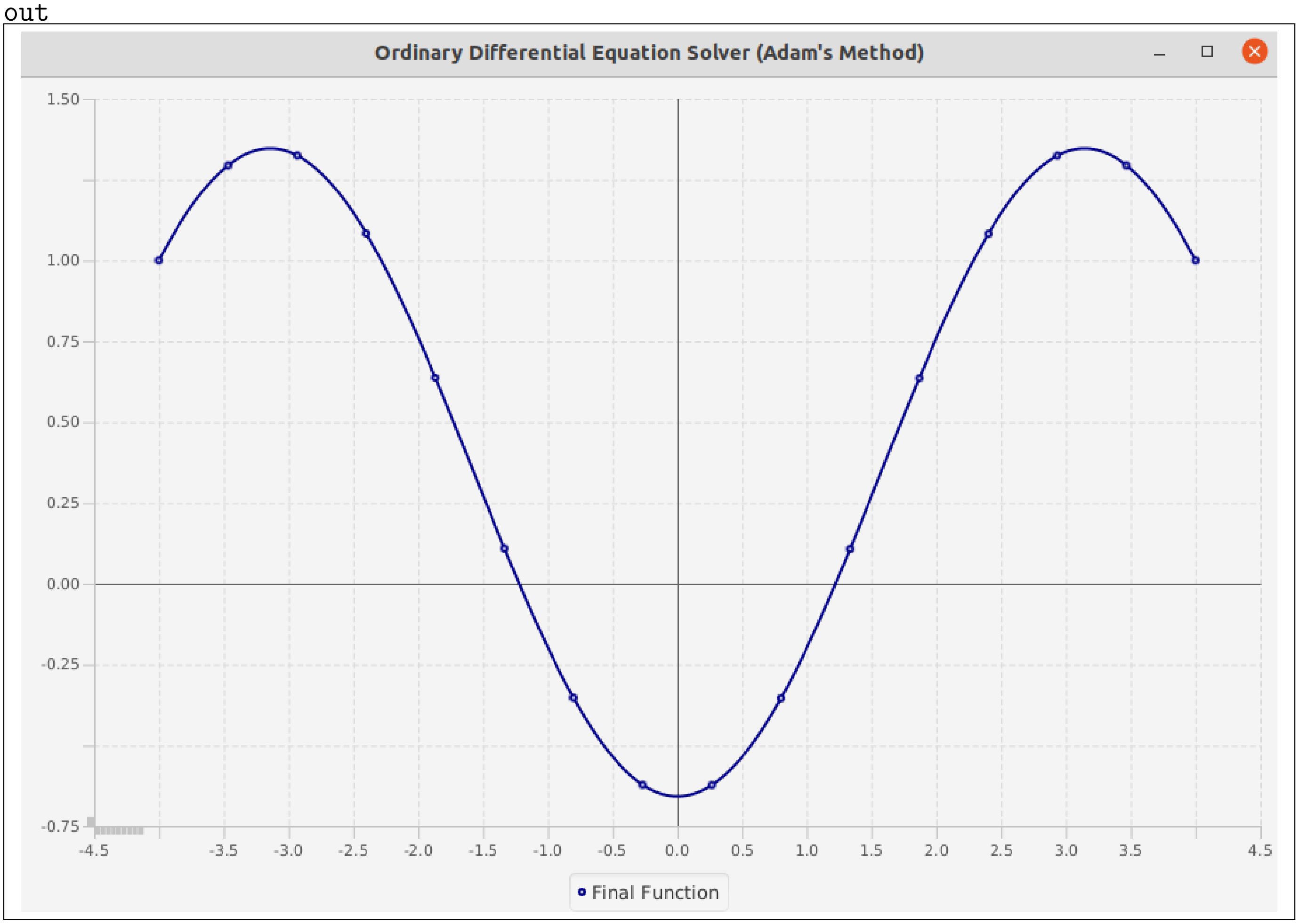
.

in

differential.properties

```
# ================================
# = ORDINARY DIFFERENTIAL EQUATION
# ================================
# Left hand side of ordinary differential equation
differential.function=sin(x)
# The interval, where searches answer of equation
differential.equation.x0=-4
differential.equation.xN=4
# Value of function at point x0
differential.equation.y0=1
# Specified accuracy
differential.equation.epsilon=0.1
```

out



Вывод

Метод Эйлер является самым простым способом решения задачи Коши, но недостаточно точным. Для исползования данного метода нужно выбирать шаг интегрирования h очень маленьким.

Модификационный метод Эйлер на порядок точнее обычного.

Все методы Эйлера являются частными случиями метода Рунге-Кутта k-го порядка. Данный метод позволяет проводить вычисления с большим шагом.

Многощаговые методы построены на том, что для вычисления значения x_{k+1} применяются несколько предыдущих точек. При этом предыдещие несколько точек должны быть вычислены одношаговым методом, т.е. многошаговые методы зависят от одношаговых.

Для вычисления каждой следующей приближений, в методе Адамса, исползуется интерполяционный полином Лагранжа, а в методе Милна – интерполяционный полином Ньютона.