Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа №3

по курсу «Численные методы»

«Методы Рунге-Кутта численного решения задачи Коши для системы обыкновенных дифференциальных уравнений»

Студент:

Группа: ИУ9-61Б

Преподаватель: Домрачева А.Б.

Москва 2025

1 Постановка задачи

Дано: Задача Коши для дифференциального уравнения второго порядка (вариант 4):

$$y'' - y' = 2(1 - x), \quad y(0) = 1, \quad y'(0) = 1$$

на отрезке $[0; 1]$.

Найти:

1. Численное решение с погрешностью $\varepsilon = 0.001$
2. Точное аналитическое решение
3. Сравнение приближенного и точного решений на каждом шаге

2 Основные теоретические сведения

2.1 Методы Рунге-Кутта для решения СОДУ

Методы Рунге-Кутта применяются для решения задачи Коши для системы обыкновенных дифференциальных уравнений первого порядка:

$$\begin{cases} y_1' = f_1(x, y_1, y_2, \ldots, y_n) \\ y_2' = f_2(x, y_1, y_2, \ldots, y_n) \\ \vdots \\ y_n' = f_n(x, y_1, y_2, \ldots, y_n) \end{cases}$$

на отрезке $[x_0, x_{end}]$ с начальными условиями:

$$y_1(x_0) = y_{01}, \ldots, y_n(x_0) = y_{0n}$$

В векторной форме:

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}), \quad \mathbf{y}(x_0) = \mathbf{y}_0$$

где:

- $\mathbf{y} = (y_1(x), \ldots, y_n(x))$ - вектор искомых функций
- $\mathbf{f} = (f_1(x, \mathbf{y}), \ldots, f_n(x, \mathbf{y}))$ - вектор правых частей
- $\mathbf{y}_0 = (y_{01}, \ldots, y_{0n})$ - вектор начальных условий

2.2 Классический метод Рунге-Кутты 4-го порядка

Метод заключается в последовательном вычислении коэффициентов:

$$\begin{cases} K_1 = \mathbf{f}(x, \mathbf{y}) \\ K_2 = \mathbf{f}\left(x + \frac{h}{2}, \mathbf{y} + \frac{h}{2}K_1\right) \\ K_3 = \mathbf{f}\left(x + \frac{h}{2}, \mathbf{y} + \frac{h}{2}K_2\right) \\ K_4 = \mathbf{f}(x + h, \mathbf{y} + hK_3) \end{cases}$$

Приближенное решение в точке $x + h$:

$$\mathbf{y}(x + h) \approx \mathbf{y}_h = \mathbf{y} + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

Метод имеет четвертый порядок точности:

$$\|\mathbf{y}(x + h) - \mathbf{y}_h\| \leq Ch^5$$

где C - константа, зависящая от правых частей системы.

2.3 Автоматический выбор шага

Для контроля точности используется следующий алгоритм:

1. Вычисляют два приближения:
   - y_h - два шага длины h
   - y_{2h} - один шаг длины $2h$
2. Оценивают погрешность по правилу Рунге:

$$err = \frac{1}{2^p - 1}\|y_h - y_{2h}\|$$

где $p = 4$ - порядок точности метода

3. Вычисляют оптимальный шаг:

$$h_{opt} = h \left(\frac{\epsilon}{err} \right)^{\frac{1}{p+1}}$$

4. Если $err \leq \epsilon$, то:

   - Принимают решение y_h или уточненное значение $y_h + \frac{y_h - y_{2h}}{2^p - 1}$
   - Новый шаг $h_{new} = 0.9 h_{opt}$

5. Если $err > \epsilon$, шаг уменьшают и повторяют вычисления

2.4 Приведение к системе ОДУ первого порядка

Введем замену переменных:

$$\begin{cases} y_1 = y \\ y_2 = y' \end{cases}$$

Получаем систему:

$$\begin{cases} y_1' = y_2 \\ y_2' = 2(1 - x) + y_2 \end{cases}$$

с начальными условиями $y_1(0) = 1$, $y_2(0) = 1$.

3 Реализация

```go
package main

import (
  "fmt"
  "math"
)

// вариант 4: "y - y' = 2(1 - x)
func fd(x float64, y []float64) []float64 {
  // y[0] = y, y[1] = y'
```

```go
  // u' = v
  // v' = v + 2(1 - x)
  return []float64{
    y[1],              // u' = v
    y[1] + 2.0*(1.0-x), // v' = v + 2(1 - x)
  }
}

// точное решение вариант( 4: y = e^x + x^2)
func exactSolution(x float64) float64 {
  return math.Exp(x) + x*x
}

func rungeKutta4(x0, xEnd float64, y0 []float64, eps float64, startH float64) ([]float64, []float64, []float64, []float64, []float64) {
  var xPoints, yPoints, exactPoints, runge, hArr []float64
  x := x0
  y := make([]float64, len(y0))
  copy(y, y0)
  h := startH
  k := 0.9

  hArr = append(hArr, h)
  xPoints = append(xPoints, x)
  yPoints = append(yPoints, y[0])
  exactPoints = append(exactPoints, exactSolution(x))
  runge = append(runge, 0.0)

  for x < xEnd {
    if x+h > xEnd {
      h = xEnd - x
    }

    y1 := stepRK4(x, y, h)
    y2 := stepRK4(x+h, y1, h)
    y2h := stepRK4(x+h, y, 2*h)

    err := math.Abs(y2[0]-y2h[0]) / 15.0
    hOpt := h * math.Pow(eps/err, 1.0/5.0)

    if err < eps {
      xPoints = append(xPoints, x+h)
      yPoints = append(yPoints, y1[0])
      exactPoints = append(exactPoints, exactSolution(x+h))
      runge = append(runge, err)
      hArr = append(hArr, h)
```

```

      if x+2*h <= xEnd {
        xPoints = append(xPoints, x+2*h)
        yPoints = append(yPoints, y2[0])
        exactPoints = append(exactPoints, exactSolution(x+2*h))
        runge = append(runge, err)
        hArr = append(hArr, h)
      }

      x += 2 * h
      y = y2
      h = math.Min(k*hOpt, xEnd-x)
    } else {
      h = k * hOpt
    }
  }

  return xPoints, yPoints, exactPoints, runge, hArr
}

func stepRK4(x float64, y []float64, h float64) []float64 {
  k1 := fd(x, y)

  k2y := make([]float64, len(y))
  for i := range y {
    k2y[i] = y[i] + h/2*k1[i]
  }
  k2 := fd(x+h/2, k2y)

  k3y := make([]float64, len(y))
  for i := range y {
    k3y[i] = y[i] + h/2*k2[i]
  }
  k3 := fd(x+h/2, k3y)

  k4y := make([]float64, len(y))
  for i := range y {
    k4y[i] = y[i] + h*k3[i]
  }
  k4 := fd(x+h, k4y)

  yNew := make([]float64, len(y))
  for i := range y {
    yNew[i] = y[i] + h/6*(k1[i]+2*k2[i]+2*k3[i]+k4[i])
  }

```

```
	return yNew
}

func main() {
	// Начальные условия вариант( 4)
	x0 := 0.0
	xEnd := 1.0
	y0 := []float64{1.0, 1.0} // y(0) = 1, y'(0) = 1
	eps := 0.001
	h := 1.0

	xPoints, yPoints, exactPoints, runge, hArr := rungeKutta4(x0, xEnd, y0, eps, h)

	fmt.Println("Автоматический подбор шага")
	fmt.Printf("%-10s %-20s %-20s %-12s %-12s %-6s\n", "x", "Приближенное y(x)", "Точное y(x)", "Ошибка", "Погрешность", "Шаг")
	for i := range xPoints {
		err := math.Abs(yPoints[i] - exactPoints[i])
		fmt.Printf("%-10.5f %-20.7f %-20.7f %-12.7f %-12.7f %-6.5f\n",
			xPoints[i],
			yPoints[i],
			exactPoints[i],
			err,
			runge[i],
			hArr[i])
	}

	constH(h)
}
```

Листинг 1: Метод Рунге-Кутта с адаптивным шагом

```
package main

import (
	"fmt"
	"math"
)

func rungeKutta4ConstH(x0, xEnd float64, y0 []float64, eps float64, startH float64) ([][]float64, [][]float64, [][]float64, [][]float64, [][]float64) {
	var xPointsArr, yPointsArr, exactPointsArr, rungeArr, hArrArr [][]float64
	var xPoints, yPoints, exactPoints, runge, hArr []float64
	h := startH

	for {
```

```
    if len(xPoints) != 0 {
      xPointsArr = append(xPointsArr, xPoints)
      yPointsArr = append(yPointsArr, yPoints)
      exactPointsArr = append(exactPointsArr, exactPoints)
      hArrArr = append(hArrArr, hArr)
      rungeArr = append(rungeArr, runge)
    }

    xPoints = []float64{}
    yPoints = []float64{}
    exactPoints = []float64{}
    hArr = []float64{}
    runge = []float64{}

    localMax := 0.0

    x := x0
    y := make([]float64, len(y0))
    copy(y, y0)

    hArr = append(hArr, h)
    xPoints = append(xPoints, x)
    yPoints = append(yPoints, y[0])
    exactPoints = append(exactPoints, exactSolution(x))
    runge = append(runge, 0.0)

    for x < xEnd {

      y1 := stepRK4(x, y, h)
      y2 := stepRK4(x+h, y1, h)
      y2h := stepRK4(x+h, y, 2*h)

      err := math.Abs(y2[0]-y2h[0]) / 15.0
      localMax = math.Max(err, localMax)

      xPoints = append(xPoints, x+h)
      yPoints = append(yPoints, y1[0])
      exactPoints = append(exactPoints, exactSolution(x+h))
      runge = append(runge, err)
      hArr = append(hArr, h)

      if x+2*h <= xEnd {
        xPoints = append(xPoints, x+2*h)
        yPoints = append(yPoints, y2[0])
        exactPoints = append(exactPoints, exactSolution(x+2*h))
        runge = append(runge, err)
```

```go
				hArr = append(hArr, h)
			}

			x += 2 * h
			y = y2
		}

		if localMax < eps {
			xPointsArr = append(xPointsArr, xPoints)
			yPointsArr = append(yPointsArr, yPoints)
			exactPointsArr = append(exactPointsArr, exactPoints)
			hArrArr = append(hArrArr, hArr)
			rungeArr = append(rungeArr, runge)
			break
		} else {
			h = h / 2
		}
	}

	return xPointsArr, yPointsArr, exactPointsArr, rungeArr, hArrArr
}

func constH(startH float64) {
	// Начальные условия вариант( 4)
	x0 := 0.0
	xEnd := 1.0
	y0 := []float64{1.0, 1.0} // y(0) = 1, y'(0) = 1
	eps := 0.001

	xPoints, yPoints, exactPoints, runge, hArr := rungeKutta4ConstH(x0, xEnd, y0,
		eps, startH)
	l := len(xPoints)

	fmt.Println("Константный шаг")
	for j := 0; j < l; j++ {
		fmt.Println("
		----------------------------------------------------------------------------------
		")
		fmt.Printf("Проход %d с шагом %0.7f\n", j, hArr[j][0])
		if j == l-1 {
			fmt.Println("Нужный шаг найден!!!!!")
		}

		resX, resY, exP, rungRes, hArrRes := xPoints[j], yPoints[j], exactPoints[j],
		 runge[j], hArr[j]

```

```
    fmt.Printf("%-10s %-20s %-20s %-12s %-12s %-6s\n", "x", "Приближенное y(x)",
   "Точное y(x)", "Ошибка", "Погрешность", "Шаг")

    for i := range resX {
      err := math.Abs(resY[i] - exP[i])
      fmt.Printf("%-10.5f %-20.7f %-20.7f %-12.7f %-12.7f %-6.5f\n",
        resX[i],
        resY[i],
        exP[i],
        err,
        rungRes[i],
        hArrRes[i])
    }
  }
}
```

Листинг 2: Метод Рунге-Кутта с постоянным шагом

4 Результаты

Таблица 1: Автоматический подбор шага метода Рунге-Кутта

x	Приближенное $y(x)$	Точное $y(x)$	Ошибка	Погрешность	Шаг
0.00000	1.0000000	1.0000000	0.0000000	0.0000000	1.00000
0.13978	1.1695586	1.1695591	0.0000005	0.0008018	0.13978
0.27956	1.4007001	1.4007012	0.0000010	0.0008018	0.13978
0.41104	1.6773458	1.6773475	0.0000016	0.0006637	0.13148
0.54253	2.0146819	2.0146842	0.0000024	0.0006637	0.13148
0.67097	2.4063431	2.4063463	0.0000032	0.0006177	0.12845
0.79942	2.8633189	2.8633232	0.0000042	0.0006177	0.12845
0.92672	3.3849988	3.3850042	0.0000054	0.0006010	0.12730

5 Вывод

В ходе выполнения лабораторной работы была решена задача Коши для дифференциального уравнения второго порядка методом Рунге–Кутты четвёртого порядка. Уравнение было приведено к системе первого порядка, после чего были реализованы два подхода: с постоянным шагом и с автоматической адаптацией шага в зависимости от заданной точности. Полученные численные решения сравнивались с точным аналитическим решением, что позволило

Таблица 2: Метод Рунге-Кутта с постоянным шагом

Проход	x	Приближенное $y(x)$	Точное $y(x)$	Ошибка	Погрешность	Шаг
0	0.00000	1.0000000	1.0000000	0.0000000	0.0000000	1.00000
	1.00000	3.7083333	3.7182818	0.0099485	0.5556713	1.00000
1	0.00000	1.0000000	1.0000000	0.0000000	0.0000000	0.50000
	0.50000	1.8984375	1.8987213	0.0002838	0.0478231	0.50000
	1.00000	3.7173462	3.7182818	0.0009356	0.0478231	0.50000
2	0.00000	1.0000000	1.0000000	0.0000000	0.0000000	0.25000
	0.25000	1.3465169	1.3465254	0.0000085	0.0049654	0.25000
	0.50000	1.8986995	1.8987213	0.0000218	0.0049654	0.25000
	0.75000	2.6794580	2.6795000	0.0000420	0.0049767	0.25000
	1.00000	3.7182099	3.7182818	0.0000719	0.0049767	0.25000
3	0.00000	1.0000000	1.0000000	0.0000000	0.0000000	0.12500
	0.12500	1.1487732	1.1487735	0.0000003	0.0005675	0.12500
	0.25000	1.3465248	1.3465254	0.0000006	0.0005675	0.12500
	0.37500	1.5956154	1.5956164	0.0000010	0.0005676	0.12500
	0.50000	1.8987198	1.8987213	0.0000015	0.0005676	0.12500
	0.62500	2.2588688	2.2588710	0.0000021	0.0005678	0.12500
	0.75000	2.6794971	2.6795000	0.0000029	0.0005678	0.12500
	0.87500	3.1644964	3.1645003	0.0000038	0.0005681	0.12500
	1.00000	3.7182768	3.7182818	0.0000050	0.0005681	0.12500

наглядно оценить точность каждого метода. Использование адаптивного шага позволило достичь требуемой точности $\varepsilon = 0.001$ при меньшем числе итераций по сравнению с методом с постоянным шагом, обеспечив более эффективное использование вычислительных ресурсов.