МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Компьютерный практикум по курсу «ВВЕДЕНИЕ В ЧИСЛЕННЫЕ МЕТОДЫ» Задание №2 (1)

ОТЧЁТ

о выполненном задании

студента 203 учебной группы факультета ВМК МГУ
Мартынова Олега Павловича

Декабрь 2015

Оглавление

Глава 1

Постановка задачи и её целей

1.1 Цель работы

В данной работе требуется освоить методы Рунге—Кутты второго и четвертого порядка точности и применить их для численного решения задачи Коши для ОДУ первого порядка и системы ОДУ первого порядка, разрешенных относительно производных.

1.2 Постановка задачи

Рассматривается ОДУ первого порядка, разрешенное относительно производной и имеющее вид:

$$\frac{dy}{dx} = f(x, y),\ x_0 < x, \tag{1.1}$$

с дополнительным начальным условием, заданным в точке $x = x_0$:

$$f(x_0) = y_0. \tag{1.2}$$

Предполагается, что правая часть уравнения (1.1) функция такова, что гарантирует существование и единственность решения задачи Коши (1.1)-(1.2).

В том случае, если рассматривается не одно дифференциальное уравнение вида (1.1), а система обыкновенных дифференциальных уравнений первого порядка, разрешенных относительно производных неизвестных функций, то соответствующая задача Коши имеет вид (на примере двух дифференциальных уравнений):

$$\begin{cases} \frac{dy_1}{dx} = f_1(x, y_1, y_2), \\ \frac{dy_2}{dx} = f_2(x, y_1, y_2), \\ x > x_0. \end{cases} \tag{1.3}$$

Дополнительные начальные условия задаются в точке $x = x_0$:

$$y_1(x_0) = y_1^{(0)},\ y_2(x_0) = y_2^{(0)}. \tag{1.4}$$

Также предполагается, что правые части уравнений из (1.3) заданы так, что это гарантирует существование и единственность решения задачи Коши (1.3)-(1.4), но уже для системы обыкновенных дифференциальных уравнений первого порядка в форме, разрешенной относительно производных неизвестных функций. Заметим, что к подобным задачам сводятся многие важные задачи, возникающие в механике (уравнения движения материальной точки), небесной механике, химической кинетике, гидродинамике и т. п.

1.3 Цели и задачи практической работы

1. Решить задачу Коши (1.1)-(1.2) (или (1.3)-(1.4)) наиболее известными и широко используемыми на практике методами Рунге—Кутты второго и четвертого порядка точности, аппроксимировав дифференциальную задачу соответствующей разностной схемой (на равномерной сетке); полученное конечно-разностное уравнение (или уравнения в случае системы), представляющее фактически некоторую рекуррентную формулу, просчитать численно;

2. Найти численное решение задачи и построить его график;

3. Найденное численное решение сравнить с точным решением дифференциального уравнения (подобрать специальные тесты, где аналитические решения находятся в классе элементарных функций, при проверке можно использовать ресурсы on-line системы http://www.wolframalpha.com или пакета Maple и т.п.).

Глава 2

Алгоритм решения

2.1 Семейство прямых методов Рунге—Кутты

Семейство прямых методов Рунге—Кутты задаётся формулами:

$$\vec{y}_{n+1} = \vec{y}_n + h \sum_{i=1}^{s} b_i \vec{k}_i, \tag{2.1}$$

где h — величина шага сетки по x и вычисление нового значения проходит в s этапов:

$$\begin{aligned}
&\vec{k}_1 = \vec{f}(x_n, \vec{y}_n),\\
&\vec{k}_2 = \vec{f}(x_n + c_2 h, \vec{y}_n + a_{21}\vec{k}_1),\\
&\cdots\\
&\vec{k}_s = \vec{f}(x_n + c_s h, \vec{y}_n + a_{s1}\vec{k}_1 + a_{s2}\vec{k}_2 + \ldots + a_{s,s-1}\vec{k}s - 1).
\end{aligned}$$

Конкретный метод определяется числом s и коэффициентами b_i, a_{ij} и c_i. Эти коэффициенты часто упорядочивают в таблицу (называемую таблицей Бутчера):

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

Для коэффициентов метода Рунге—Кутты должны быть выполнены условия $\sum_{j=1}^{i-1} a_{ij} = c_i i = 2, \ldots, s$. Если требуется, чтобы метод имел порядок p, то следует также обеспечить условие:

$$\vec{\mathbf{y}}(h + x_0) - \vec{y}(h + x_0) = O(h^{p+1}),$$

где $\vec{\mathbf{y}}(h + x_0)$ — приближение, полученное по методу Рунге—Кутты. После многократного дифференцирования это условие преобразуется в систему полиномиальных уравнений относительно коэффициентов метода.

2.2 Метод Рунге—Кутты второго порядка

Методу Рунге—Кутты второго порядка соотвествует следующая таблица Бутчера:

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \end{array}$$

что соотвествует следующему уравнению итерации:

$$\vec{y}_{n+1} = \vec{y}_n + h\vec{k}_2. \tag{2.2}$$

где коэффициенты $\vec{k}_1, \vec{k}_2$ находятся следующим образом:

$$\vec{k}_1 = \vec{f}(x_n, \vec{y}_n),$$
$$\vec{k}_2 = \vec{f}(x_n + \frac{h}{2}, \vec{y}_n + \frac{h}{2}\vec{k}_1).$$

2.3 Метод Рунге—Кутты четвертого порядка

Методу Рунге—Кутты четвертого порядка соотвествует следующая таблица Бутчера:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

что соотвествует следующему уравнению итерации:

$$\vec{y}_{n+1} = \vec{y}_n + \frac{h}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4), \tag{2.3}$$

где коэффициенты $\vec{k}_1, \vec{k}_2, \vec{k}_3, \vec{k}_4$ находятся следующим образом:

$$\begin{aligned}
\vec{k}_1 &= \vec{f}(x_n, \vec{y}_n),\\
\vec{k}_2 &= \vec{f}(x_n + \frac{h}{2}, \vec{y}_n + \frac{h}{2}\vec{k}_1),\\
\vec{k}_3 &= \vec{f}(x_n + \frac{h}{2}, \vec{y}_n + \frac{h}{2}\vec{k}_2),\\
\vec{k}_4 &= \vec{f}(x_n + h, \vec{y}_n + h\vec{k}_3).
\end{aligned}$$

Глава 3

Описание программы

3.1 Использование

Программа написана на языке программирования Python и состоит из нескольких модулей, из которых для пользователя представляет интерес только один — модуль *data.py*. Этот модуль содержит в себе единственную переменную — *data*, — представляющую из себя список записей, где каждая запись соответсвует уравнению или системе уравнений, предназначенных для тестирования. После заполнения этой переменной, необходимо запустить модуль *test.py*, который выведет на экран список таблиц, в которых будет отражена информация о сравнении методов решения ОДУ между собой, а также значения погрешностей каждого из методов. Помимо этого, этот модуль генерирует графики для каждой из функций, которые после его запуска могут быть найденны в текущей директории в виде изображений в привычном формате.

3.2 Детали реализации

В модуле *runge_kutta.py* реализованы функции *runge_kutta_2* и *runge_kutta_4*, соответствующие методам Рунге—Кутты решения ОДУ с вторым и четвертым порядками точности. Эти функции принимают в качестве входных данных систему из любого числа уравнений, начальное условие, правую границу отрезка, на котором будет происходить вычисление, и величину шага.

Глава 4

Тестирование

4.1 Примеры из одного уравнения

Пример 1. Тестовый пример 1-2 из оригинального задания.

$$\begin{aligned} f(x,y) &= \sin x - y, \\ y(x) &= -0.5\cos x + 0.5\sin x + \frac{21}{2}e^{-x}, \\ (x_o, y_0) &= (0, 10), \\ [x_0, x_n] &= [0, 10], \\ h &= 1. \end{aligned}$$

Вывод программы:

```
------------------------------------------------------------
| Number of segments:                                   10 |
| Mean Squared Error (rk2):                         0.4959 |
| Mean Squared Error (rk4):                         0.2058 |
------------------------------------------------------------
|       x |     rk2 |     rk4 |   exact | err_rk2 | err_rk4 |
------------------------------------------------------------
|  0.0000 | 10.0000 | 10.0000 | 10.0000 |  0.0000 |  0.0000 |
|  1.0000 |  5.5768 |  4.3522 |  4.0133 |  1.5634 |  0.3389 |
|  2.0000 |  2.9322 |  1.9429 |  2.0837 |  0.8485 |  0.1409 |
|  3.0000 |  1.0448 |  0.4622 |  1.0883 |  0.0436 |  0.6262 |
|  4.0000 | -0.0767 | -0.4253 |  0.1407 |  0.2175 |  0.5661 |
|  5.0000 | -0.2645 | -0.5400 | -0.5505 |  0.2861 |  0.0105 |
|  6.0000 |  0.2226 | -0.0150 | -0.5938 |  0.8164 |  0.5788 |
|  7.0000 |  0.7208 |  0.5775 | -0.0389 |  0.7597 |  0.6164 |
|  8.0000 |  0.6642 |  0.6592 |  0.5710 |  0.0933 |  0.0882 |
|  9.0000 |  0.0509 |  0.1424 |  0.6629 |  0.6120 |  0.5205 |
| 10.0000 | -0.5822 | -0.5025 |  0.1480 |  0.7302 |  0.6505 |
------------------------------------------------------------
```

Пример 2. Упражнение №391 из задачника А.Ф. Филиппова.

$$
\begin{aligned}
f(x,y) &= \frac{y^2 - x}{2y(x+1)},\\
y(x) &= \sqrt{x - (x+1)\ln(e^{-1}(x+1))},\\
(x_o, y_0) &= (0, 1),\\
[x_0, x_n] &= [0, 5],\\
h &= 0.5.
\end{aligned}
$$

Вывод программы:

```
-------------------------------------------------------------
| Number of segments:                                    10 |
| Mean Squared Error (rk2):                          5.4084 |
| Mean Squared Error (rk4):                          0.4015 |
-------------------------------------------------------------
|       x |     rk2 |     rk4 |   exact | err_rk2 | err_rk4 |
-------------------------------------------------------------
|  0.0000 |  1.0000 |  1.0000 |  1.0000 |  0.0000 |  0.0000 |
|  0.5000 |  1.0460 |  1.0447 |  1.1797 |  0.1338 |  0.1350 |
|  1.0000 |  1.0307 |  1.0278 |  1.2703 |  0.2396 |  0.2425 |
|  1.5000 |  0.9649 |  0.9596 |  1.3074 |  0.3425 |  0.3478 |
|  2.0000 |  0.8472 |  0.8375 |  1.3054 |  0.4583 |  0.4680 |
|  2.5000 |  0.6612 |  0.6405 |  1.2710 |  0.6098 |  0.6305 |
|  3.0000 |  0.3389 |  0.2363 |  1.2062 |  0.8672 |  0.9699 |
|  3.5000 | -2.8656 | -0.4625 |  1.1098 |  3.9754 |  1.5723 |
|  4.0000 | -2.9340 |  0.7478 |  0.9761 |  3.9102 |  0.2284 |
|  4.5000 | -2.9933 |  0.4224 |  0.7899 |  3.7832 |  0.3674 |
|  5.0000 | -3.0447 |  0.4496 |  0.4994 |  3.5441 |  0.0498 |
-------------------------------------------------------------
```

Пример 3. Упражнение №315 из задачника А.Ф. Филиппова (немного модифицированное).

$$
\begin{aligned}
f(x,y) &= y - x^2,\\
y(x) &= e^x + x^2 + 2x + 2,\\
(x_o, y_0) &= (0, 3),\\
[x_0, x_n] &= [0, 5],\\
h &= 0.5.
\end{aligned}
$$

Вывод программы:

Number of segments:	10
Mean Squared Error (rk2):	4580.6792
Mean Squared Error (rk4):	4500.8064

x	rk2	rk4	exact	err_rk2	err_rk4
0.0000	3.0000	3.0000	3.0000	0.0000	0.0000
0.5000	4.5625	4.5872	4.8987	0.3362	0.3115
1.0000	6.5078	6.5689	7.7183	1.2105	1.1493
1.5000	8.7627	8.8767	11.7317	2.9690	2.8550
2.0000	11.2081	11.3976	17.3891	6.1809	5.9915
2.5000	13.6507	13.9458	25.4325	11.7818	11.4867
3.0000	15.7762	16.2147	37.0855	21.3094	20.8709
3.5000	17.0738	17.6989	54.3655	37.2917	36.6666
4.0000	16.7136	17.5655	80.5982	63.8846	63.0326
4.5000	13.3471	14.4413	121.2671	107.9200	106.8258
5.0000	4.7828	6.0628	185.4132	180.6304	179.3504

Пример 4. Упражнение №322 из задачника А.Ф. Филиппова.

$$f(x,y) = e^y/(x-2),$$

$$y(x) = -\ln(-\ln(e^{-1}(x-2))),$$

$$(x_o, y_0) = (3,0),$$

$$[x_0, x_n] = [3,4],$$

$$h = 0.1.$$

Вывод программы:

Number of segments:	10
Mean Squared Error (rk2):	0.0054
Mean Squared Error (rk4):	0.0053

x	rk2	rk4	exact	err_rk2	err_rk4
3.0000	0.0000	0.0000	-0.0000	0.0000	0.0000
3.1000	0.0910	0.0910	0.1002	0.0092	0.0091
3.2000	0.1827	0.1828	0.2013	0.0186	0.0185
3.3000	0.2758	0.2760	0.3043	0.0285	0.0283
3.4000	0.3711	0.3713	0.4102	0.0391	0.0389
3.5000	0.4692	0.4695	0.5200	0.0508	0.0505
3.6000	0.5711	0.5715	0.6349	0.0638	0.0634
3.7000	0.6776	0.6782	0.7564	0.0787	0.0781
3.8000	0.7901	0.7909	0.8862	0.0961	0.0954
3.9000	0.9098	0.9109	1.0268	0.1170	0.1159
4.0000	1.0387	1.0402	1.1814	0.1426	0.1412

4.2 Примеры систем из двух уравнений

Пример 5. Тестовый пример 2-8 из оригинального задания.

$$
\begin{aligned}
f_1(x, u, v) &= \cos(x + 1.1v) + u, \\
f_2(x, u, v) &= -v^2 + 2.1u + 1.1, \\
(x_o, y_1^{(0)}, y_2^{(0)}) &= (0, 0.25, 1), \\
[x_0, x_n] &= [0, 1], \\
h &= 0.01.
\end{aligned}
$$

Вывод программы:

```
-------------------------------------------------------------
| Number of segments:                                     10 |
| Mean Squared Error (rk2):                              N/A |
| Mean Squared Error (rk4):                              N/A |
-------------------------------------------------------------
|       x |  rk2_y1 |  rk2_y2 |  rk4_y1 |  rk4_y2 |          |
-------------------------------------------------------------
|  0.0000 |  0.2500 |  1.0000 |  0.2500 |  1.0000 |          |
|  0.1000 |  0.3063 |  1.0626 |  0.3061 |  1.0622 |          |
|  0.2000 |  0.3513 |  1.1227 |  0.3509 |  1.1220 |          |
|  0.3000 |  0.3837 |  1.1779 |  0.3833 |  1.1771 |          |
|  0.4000 |  0.4032 |  1.2263 |  0.4028 |  1.2254 |          |
|  0.5000 |  0.4096 |  1.2665 |  0.4092 |  1.2656 |          |
|  0.6000 |  0.4034 |  1.2976 |  0.4030 |  1.2967 |          |
|  0.7000 |  0.3849 |  1.3192 |  0.3845 |  1.3184 |          |
|  0.8000 |  0.3548 |  1.3312 |  0.3544 |  1.3306 |          |
|  0.9000 |  0.3135 |  1.3337 |  0.3131 |  1.3332 |          |
|  1.0000 |  0.2615 |  1.3271 |  0.2610 |  1.3266 |          |
-------------------------------------------------------------
```

Глава 5

Вывод

В ходе практической работы были реализованы метод Рунге—Кутты второго и четвёртого порядков точности, применительно как к одиночным ОДУ первого порядка, разрешённым относительно производной, так и к соответствующим системам. Тестирование показало, что метод Рунге—Кутты четвёртого порядка точности действительно является более точным, по сравнению с методом второго порядка точности. Эта разница, тем не менее, оказалось слабо заметной на приведённой выборке тестовых примеров, что можно объяснить сравнительной простотой этих примеров.

Глава 6

Графики

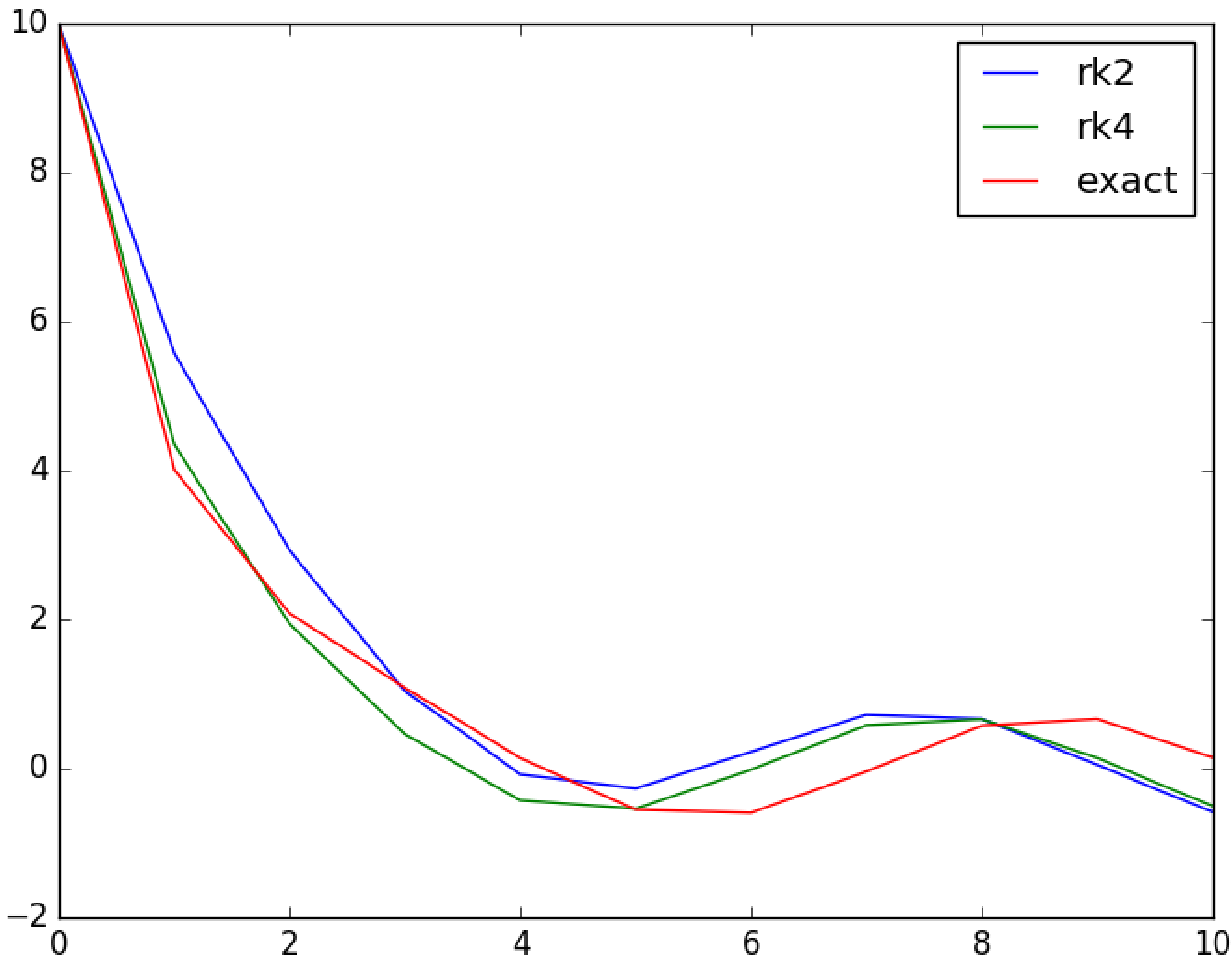

Рис. 6.1: Пример 1.

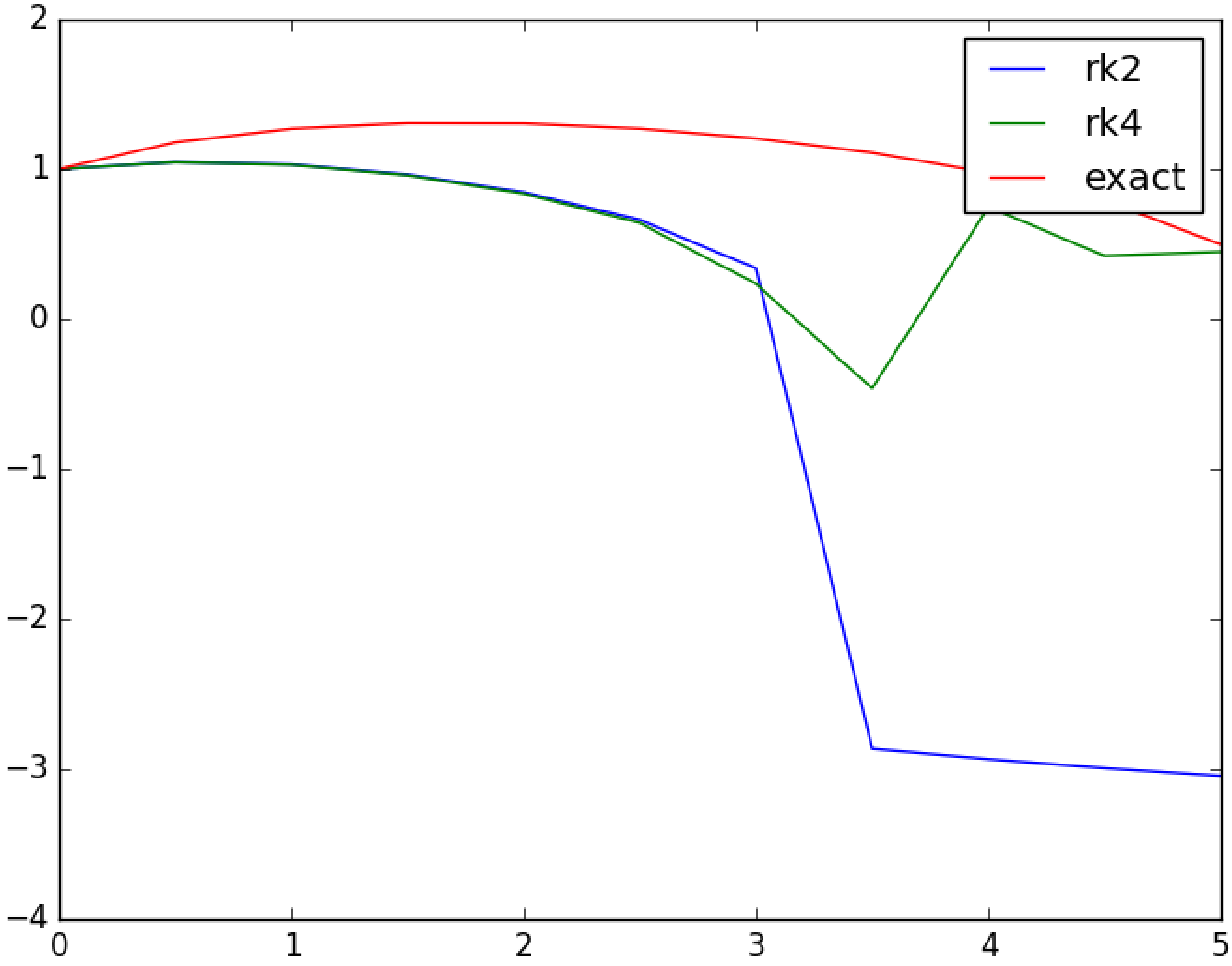

Рис. 6.2: Пример 2.

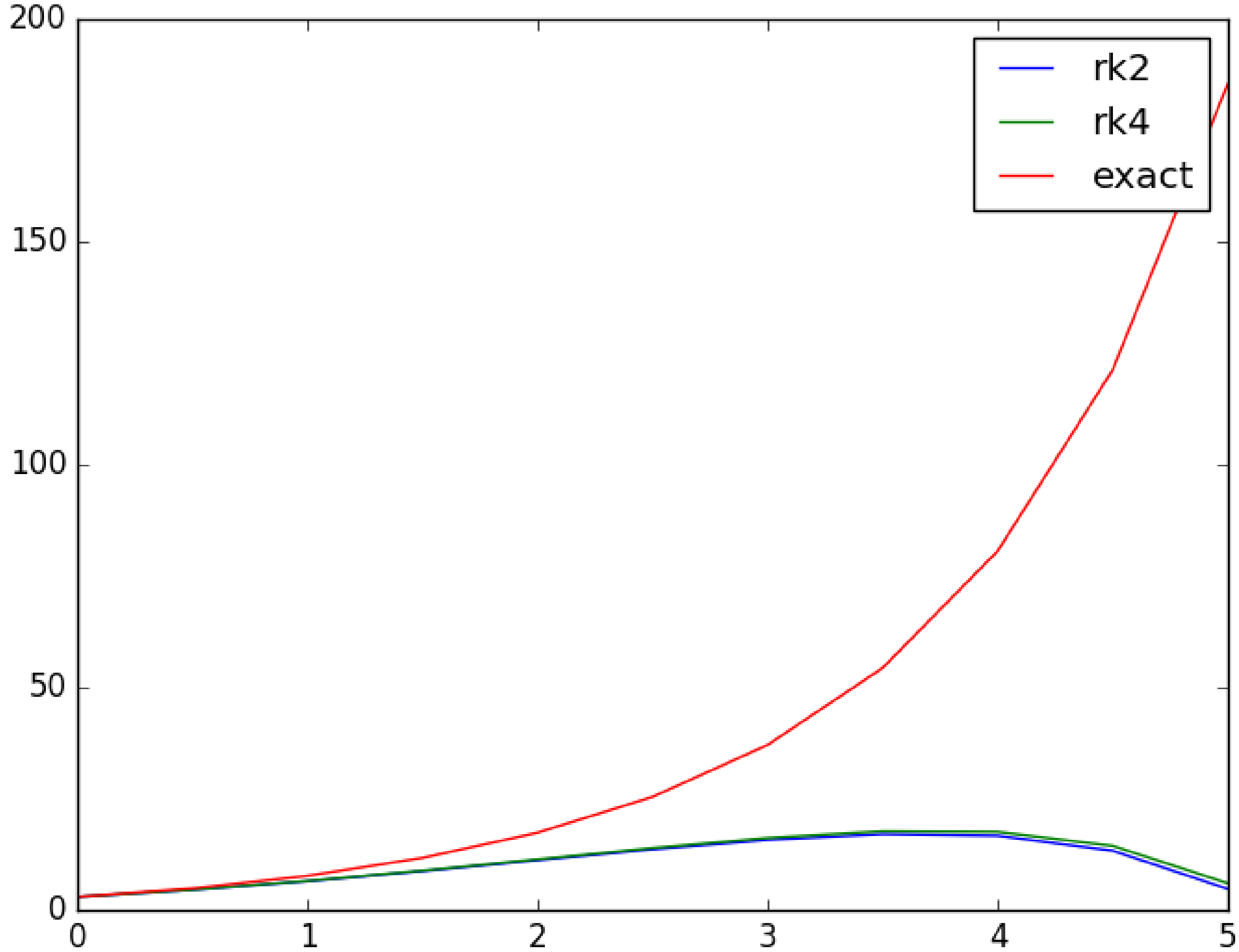

Рис. 6.3: Пример 3.

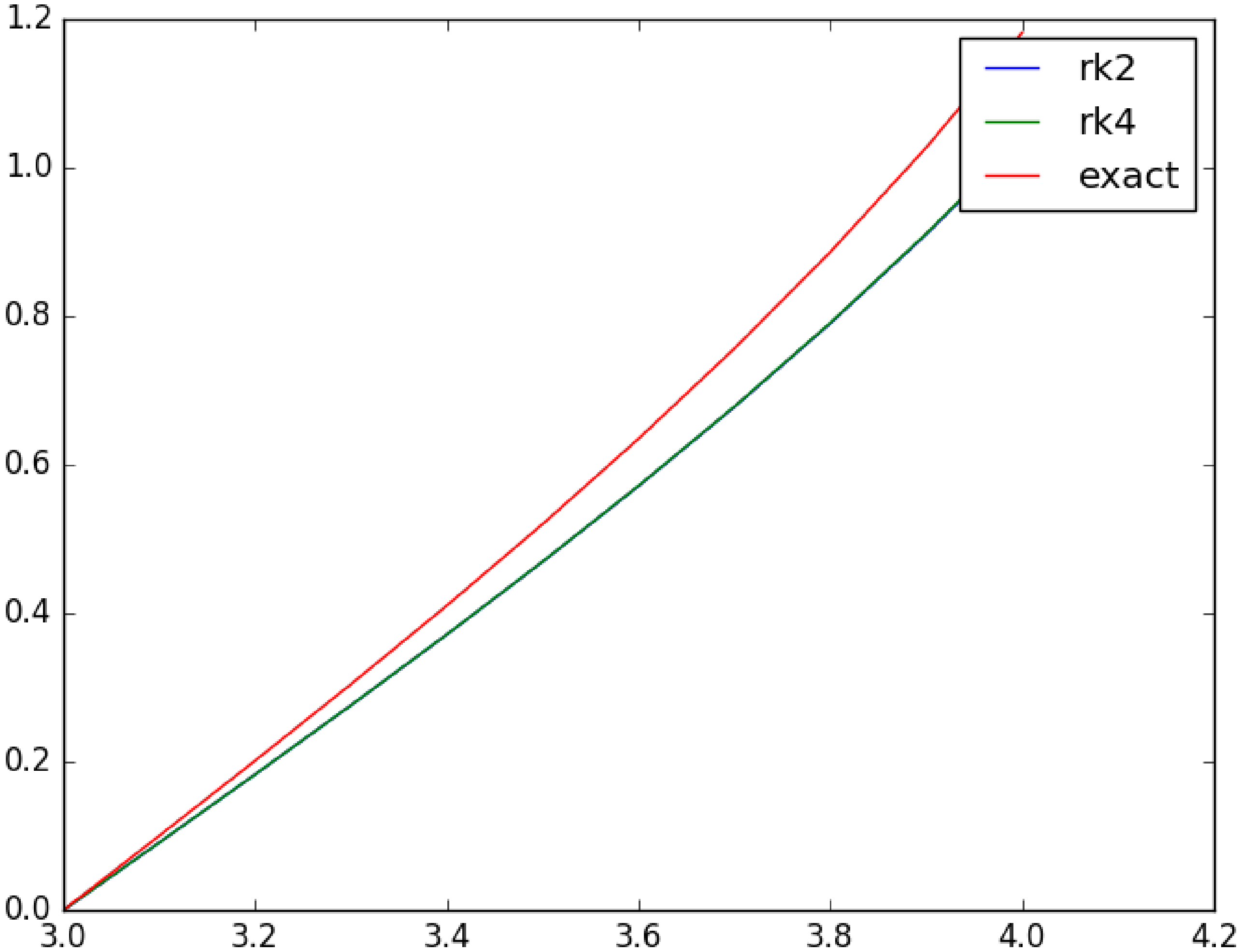

Рис. 6.4: Пример 4.

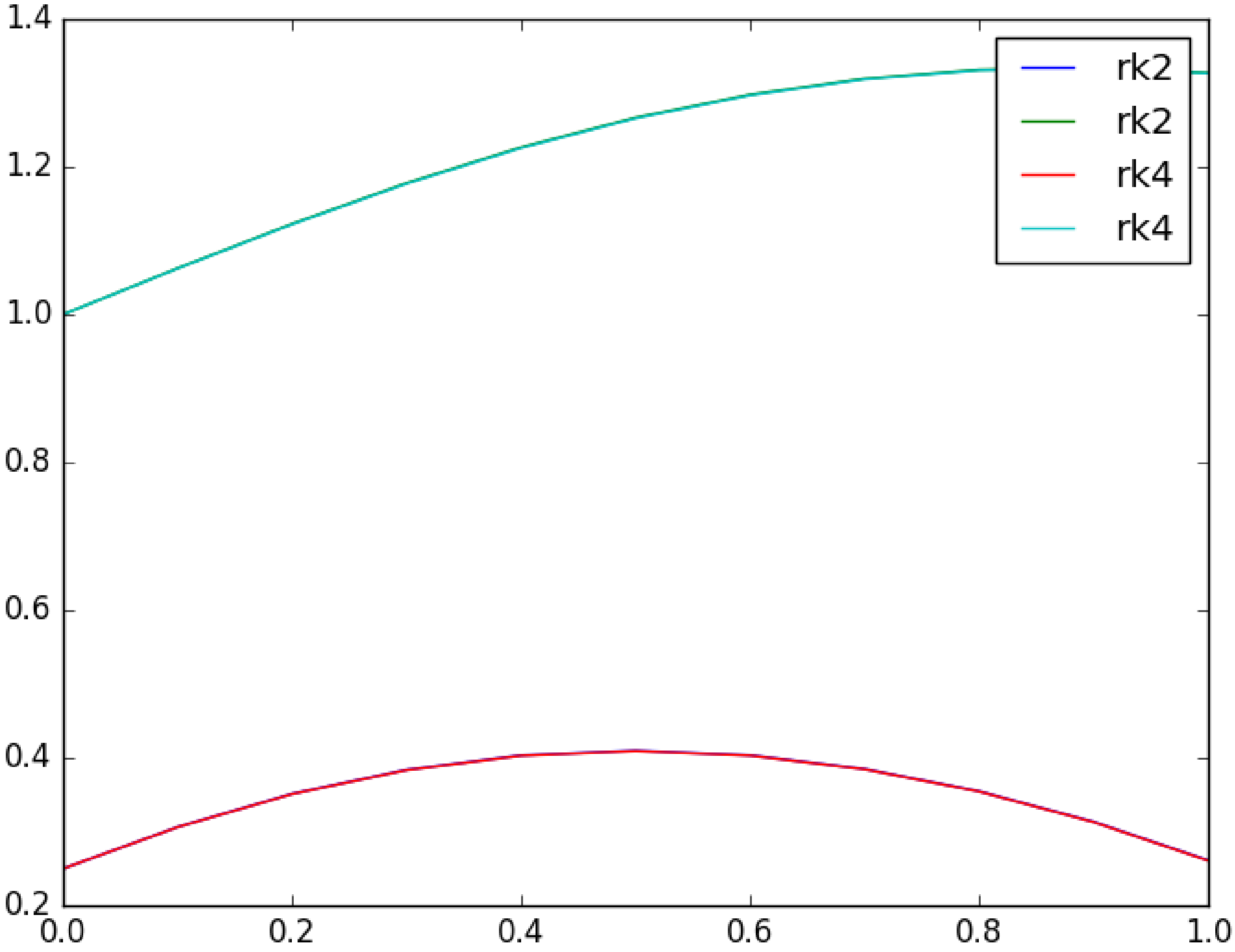

Рис. 6.5: Пример 5.

Глава 7

Исходный код

```
############################################################
#                         test.py                          #
############################################################
from data import data
from plot import plot_data
from runge_kutta import runge_kutta_2, runge_kutta_4, apply_data

# Plot table for single ODEs
for d in data:
    if len(d["f"]) != 1:
        continue

    rk2 = apply_data(d, runge_kutta_2)[0]
    rk4 = apply_data(d, runge_kutta_4)[0]

    x0, xn = d["p0"][0], d["xn"]

    h_raw = d["h"]
    # normalized step size
    n = round((xn - x0) / h_raw)
    h = (xn - x0) / n

    x = [x0 + h * i for i in range(n + 1)]
    exact = list(map(d["sol"][0], x))

    mse_rk2 = sum(map(lambda a: (a[0] - a[1]) ** 2,
                      zip(rk2, exact))) / (n + 1)
    mse_rk4 = sum(map(lambda a: (a[0] - a[1]) ** 2,
                      zip(rk4, exact))) / (n + 1)

    print("-" * 61)
    print("| Number of segments:       {:31} |".format(n))
    print("| Mean Squared Error (rk2): {:31.4f} |".format(mse_rk2))
    print("| Mean Squared Error (rk4): {:31.4f} |".format(mse_rk4))
    print("-" * 61)
    print("|{:>8} |{:>8} |{:>8} |{:>8} |{:>8} |{:>8} |"\
        .format("x", "rk2", "rk4", "exact", "err_rk2", "err_rk4"))
    print("-" * 61)
    for j in range(n + 1):
        print("|{:8.4f} |{:8.4f} |{:8.4f} |{:8.4f} |{:8.4f} |{:8.4f} |".format(x[j], rk2[j], rk4
    print("-" * 61)

# Plot table for systems of 2 ODEs
for d in data:
    if len(d["f"]) != 2:
        continue

    rk2 = apply_data(d, runge_kutta_2)
    rk4 = apply_data(d, runge_kutta_4)
```

```
    x0, xn = d["p0"][0], d["xn"]

    h_raw = d["h"]
    # normalized step size
    n = round((xn - x0) / h_raw)
    h = (xn - x0) / n

    x = [x0 + h * i for i in range(n + 1)]

    print("-" * 61)
    print("| Number of segments:       {:31} |".format(n))
    print("| Mean Squared Error (rk2): {:>31} |".format("N/A"))
    print("| Mean Squared Error (rk4): {:>31} |".format("N/A"))
    print("-" * 61)
    print("|{:>8} |{:>8} |{:>8} |{:>8} |{:>8} |{:>8} |"\
          .format("x", "rk2_y1", "rk2_y2", "rk4_y1", "rk4_y2", ""))
    print("-" * 61)
    for j in range(n + 1):
        print("|{:8.4f} |{:8.4f} |{:8.4f} |{:8.4f} |{:8.4f} |{:>8} |".format(x[j], rk2[0][j], rk
    print("-" * 61)

# Plot graphs for ODEs
for d in data:
    plot_data(d, [(runge_kutta_2, "rk2"), (runge_kutta_4, "rk4")])

############################################################
#                         plot.py                          #
############################################################
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
from random import randint

def plot(f_num, f_ex, x0, xn, h_raw):
    """
    f_num: list of y(x) values, representing the numeric solution;
    f_ex: function y(x), representing the exact solution;
    """

    # normalized step value
    n = round((xn - x0) / h_raw)
    h = (xn - x0) / n

    x = [x0 + h * i  for i in range(n + 1)]
    fig = plt.figure()
    graph = fig.add_subplot(111)
    for fi_num, l in f_num:
        graph.plot(x, fi_num, label=l)
    for fi_ex, l in f_ex:
        graph.plot(x, list(map(fi_ex, x)), label=l)
    graph.legend()
    return fig

def plot_data(data, method):
    sol_num = []
    for m, l in method:
        sol = m(data["f"],
                data["p0"],
                data["xn"],
                data["h"])
        sol_num += list(zip(sol, [l] * len(sol)))
    fig = plot(sol_num,
               list(zip(data["sol"],
                        ["exact"] * len(data["sol"]))),
               data["p0"][0],
               data["xn"],
               data["h"])
    fig.savefig("plot_{:02}.png".format(data["idx"] + 1))
```

```
############################################################
#                       runge_kutta.py                     #
############################################################
def runge_kutta(f, p0, xn, h_raw, order):
    """
    f: list of functions;
    [x0, xn]: segment where to find the solution;
    (x0, y0): initial condition;
    h_raw: step value, to be normalized to fit the segment;
    order: possible values are 2 and 4;
    """

    x0, y0 = p0[0], p0[1:]
    # number of functions
    m = len(f)
    # normalized step value
    n = round((xn - x0) / h_raw)
    h = (xn - x0) / n
    # list of y(x) functions
    y = [[y0[t]] + [0] * n for t in range(m)]
    # main loop
    for i in range(1, n + 1):
        xi = x0 + h * i
        k1 = [f[t](xi, *[y[s][i - 1]
                         for s in range(m)])
              for t in range(m)]
        k2 = [f[t](xi + h / 2, *[y[s][i - 1] + h / 2 * k1[s]
                                 for s in range(m)])
              for t in range(m)]
        if order == 4:
            k3 = [f[t](xi + h / 2, *[y[s][i - 1] + h / 2 * k2[s]
                                     for s in range(m)])
                  for t in range(m)]
            k4 = [f[t](xi + h, *[y[s][i - 1] + h * k3[s]
                                 for s in range(m)])
                  for t in range(m)]
        for t in range(m):
            if order == 2:
                y[t][i] = y[t][i - 1] + h * k2[t]
            elif order == 4:
                y[t][i] = (y[t][i - 1] + h / 6 *
                           (k1[t] + 2 * k2[t] + 2 * k3[t] + k4[t]))
    return y

def runge_kutta_2(f, p0, xn, h_raw):
    return runge_kutta(f, p0, xn, h_raw, order=2)

def runge_kutta_4(f, p0, xn, h_raw):
    return runge_kutta(f, p0, xn, h_raw, order=4)

def apply_data(data, method):
    return method(data["f"],
                  data["p0"],
                  data["xn"],
                  data["h"])

############################################################
#                          data.py                         #
############################################################
from math import *

data = [
    {
    "idx": 0,
    "desc": "Original task, ex. 1-2",
    "f": [lambda x, y: sin(x) - y],
    "p0": (0., 10.),
    "xn": 10.,
    "h": 1.0,
```

```
    "sol": [lambda x: -0.5 * cos(x) + 0.5 * sin(x) + 21 / 2 * exp(-x)]
    },
    {
    "idx": 1,
    "desc": "Filippov, ex. 391",
    "f": [lambda x, y: (y ** 2 - x) / (2 * y * (x + 1))],
    "p0": (0., 1.),
    "xn": 5.,
    "h": 0.5,
    "sol": [lambda x: sqrt(x - (x + 1) * log((1 / e) * (x + 1)))]
    },
    {
    "idx": 2,
    "desc": "Filippov, ex. 315 (modified)",
    "f": [lambda x, y: y - x ** 2],
    "p0": (0., 3.),
    "xn": 5.,
    "h": 0.5,
    "sol": [lambda x: exp(x) + x ** 2 + 2 * x + 2]
    },
    {
    "idx": 3,
    "desc": "Filippov, ex. 322",
    "f": [lambda x, y: exp(y) / (x - 2)],
    "p0": (3., 0.),
    "xn": 4.,
    "h": 0.1,
    "sol": [lambda x: -log(-log((x - 2) / e))]
    },
    {
    "idx": 4,
    "desc": "Original task, ex. 2-8",
    "f": [lambda x, u, v: cos(x + 1.1 * v) + u,
          lambda x, u, v: -(v ** 2) + 2.1 * u + 1.1],
    "p0": (0., 0.25, 1.),
    "xn": 1.,
    "h": 0.1,
    "sol": []
    }
]
```