Национальный исследовательский университет
“Московский энергетический институт”

Отчет по курсовой работе
по дисциплине: «Численные методы»

Тема курсовой работы
«Модель Вольтерра-Лотки».

Группа: А-13-18
Студент: Штыкова А.А.
Преподаватель: Амосова О.А.

Москва 2020

Оглавление:

1. Постановка задачи

> Динамика численности двух видов, один из которых является пищевым ресурсом другого (хищников и жертв), описывается следующей системой дифференциальных уравнений

$$\begin{cases} \frac{dx}{dt} = x(a - by) \\ \frac{dy}{dt} = -y(c - dx) \end{cases}$$

$x(t)$ – численность жертв

$y(t)$ – численность хищников

Коэффициента a и c характеризуют скорость прироста жертв и убыли хищников при отсутствии межвидовых контактов (при изолированном проживании). Предполагается, что корм для популяции жертв имеется в неограниченном количестве, поэтому каждое следующее поколение в идеальных условиях будет в а раз больше предыдущего. Хищники наоборот, будут лишены корма при изолированном проживании и даже размножение не спасёт их от голодной смерти (предполагается, что жертвы - их единственно возможная пища). *Поэтому слагаемое ax входит в правую часть со знаком плюс, а cy - со знаком минус*.

Вторые слагаемые в правых частях моделируют процесс встречи жертвы с хищником. С одной стороны, встреча с хищником не всегда заканчивается трапезой. С другой, для выживания хищника может быть недостаточно одной такой встречи. За соответствующую убыть жертв и прирост хищников отвечают коэффициент b и d. Все коэффициента положительны.

2. Проверка работоспособности алгоритма

Для того, чтобы проверить работу написанного алгоритма составим тестовый пример. Рассморим систему:

$$\begin{cases} \frac{dx}{dt} = -2x + 4y \\ \frac{dy}{dt} = -x + 3y \end{cases}$$
$$x(0) = 3, y(0) = 0$$

Частное решение:

$$\begin{cases} x(t) = 4e^{-t} - e^{2t} \ (1) \\ y(t) = e^{-t} - e^{2t} \ (2) \end{cases}$$

Для того, чтобы программа работала с разными начальные даннные и разными функциями (правых частей), создадим класс $Solution$. Этот класс инициализируется коэффициентами a, b, c, d при переменных x, y и начальными данными x_0, y_0.

▼ Инициализация класса

```
class Solution(object):

    def __init__(self, a, b, c, d, x0, y0):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

        self.x0 = x0
        self.y0 = y0

        self.T0 = 0
        self.T = 1
```

Для решения задачи рассмотрим 3 метода, выбрав из них оптимальный. Все методы реализованы в классе $Solution$

▼ Поиск с заданной точностью

Алгоритм поиска решения с заданной точностью есть воплощение правила Рунге.

$$r_1^h = \frac{y_2^h - y_1^{2h}}{2^p - 1}$$

Чтобы в программе избежать двойного пересчёта, изменим формулу

$$r_1^{h/2} = \frac{y_2^{h/2} - y_1^h}{2^p - 1}$$

Для каждого метода алгоритм является одинаковым за исключением порядка точности p, который будет разниться от метода к методу. Именно поэтому общая функция поиска с заданной точностью - acc одна, из которой вызывается $solver$ - метод, которым мы решаем систему.

```
def acc(self, solver, eps):
    k = 0
    h = 0.2
    t1, x_1, y_1, p = solver(h/2)
    y1 = y_1[::2]
    x1 = x_1[::2]
    t1, x2, y2 = solver(h)[0], solver(h)[1], solver(h)[2]

    e = self.find_e(y1, y2, x1, x2, p)
    h = h/2

    while(e >= eps):
        k += 1
        h /= 2
        y2 = y_1
        x2 = x_1
        t1, x_1, y_1 = solver(h)[0], solver(h)[1], solver(h)[2]
        y1 = y_1[::2]
        x1 = x_1[::2]
        e = self.find_e(y1, y2, x1, x2, p)
    print("k", k, "h = ", h)
    return t1, x_1, y_1
```

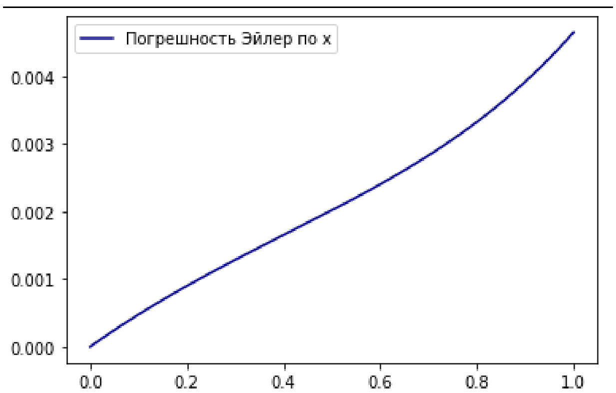
где функция $find\_e$ - это

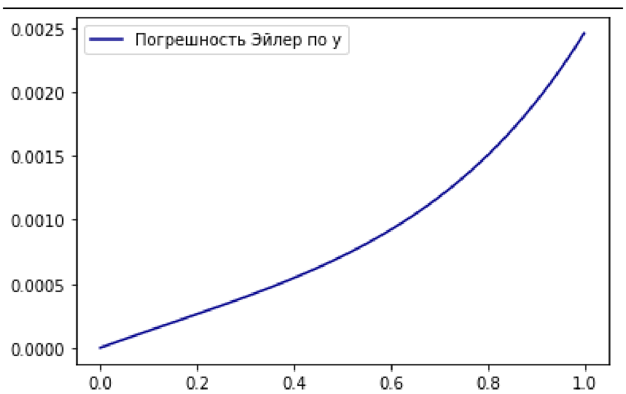
```
@staticmethod
def find_e(y1, y2, x1, x2, p):
    r1 = np.array(abs(y1 - y2)/(2**p -1))
    e1 = max(r1)
    r2 = np.array(abs(x1 - x2)/(2**p -1))
    e2 = max(r2)
    return max(e1, e2)
```

▼ Метод Эйлера

```
def euler_solver(self, h):
    t = np.arange(self.T0, self.T, h)
    y = np.zeros(len(t))
    x = np.zeros(len(t))
    y[0] = self.y0
    x[0] = self.x0
    for i in range(len(y) - 1):
        x[i + 1] = x[i] + h*(self.a*x[i] + self.b*y[i])
        y[i + 1] = y[i] + h*(self.c*x[i] + self.d*y[i])
    return t, x, y, 1
```

Графики погрешностей

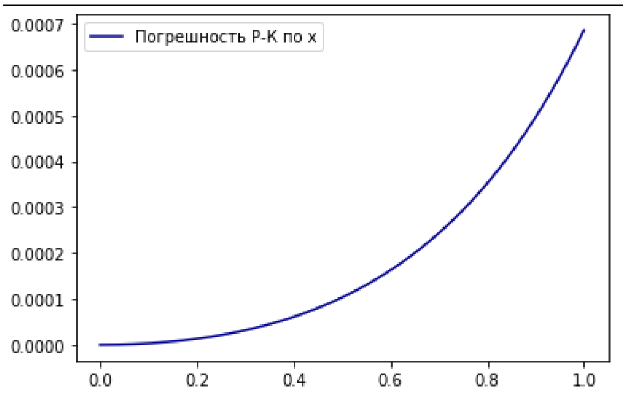


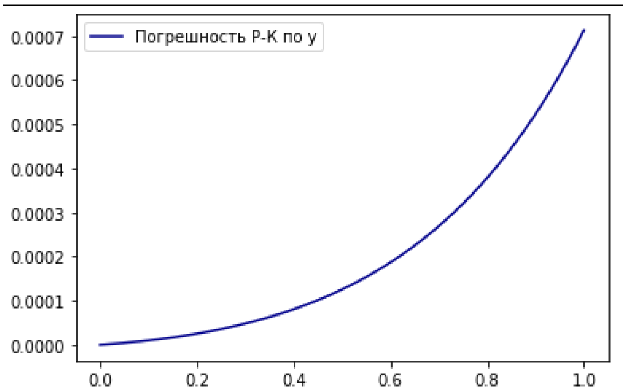


▼ Метод Рунге-Кутты 3 порядка

```
@staticmethod
def iteration(h, f, x, y, a, b):
    k1 = h*f(x, y, a, b)
    k2 = h*f(x + h/2, y + k1/2, a, b)
    k3 = h*f(x + 3/4*h, y + 3/4*k2, a, b)
    k = 1/9*(2*k1 + 3*k2 + 4*k3)
```

```
    return k

def runge_kutta_solver(self, h = 0.1):
    t = np.arange(self.T0, self.T, h)
    x = np.zeros(len(t))
    y = np.zeros(len(t))
    x[0] = self.x0
    y[0] = self.y0
    for i in range(len(t) - 1):
        x[i + 1] = x[i] + self.iteration(h, f1_test, x[i], y[i], self.a, self.b)
        y[i + 1] = y[i] + self.iteration(h, f2_test, x[i], y[i], self.c, self.d)
    return t, x, y, 3
```





▼ Метод прогноза(модифицированным методом Эйлера) и коррекции(методом Адамса-Моултона)

```
#ПРОГНОЗ
def predicator_1(self, h, x_pred, y_pred):
    half_y = x_pred + h/2 * f1(x_pred, y_pred, self.a, self.b)
    half_x = y_pred + h/2 * f2(x_pred, y_pred, self.c, self.d)
    x = x_pred + h*f1(half_x, half_y, self.a, self.b)
    y = y_pred + h*f2(half_x, half_y, self.c, self.d)
    return x, y

#КОРРЕКЦИЯ
def corrector(self, h, x_pred, y_pred, x_next, y_next):
    x = x_pred + h/2*(self.a*x_pred + self.b*y_pred + self.a*x_next + self.b*y_next)
    y = y_pred + h/2*(self.c*x_pred + self.d*y_pred + self.c*x_next + self.d*y_next)
    return x, y

def adam_solver(self, h):
    t = np.arange(self.T0, self.T, h)
    y = np.zeros(len(t))
    x = np.zeros(len(t))
    y[0] = self.y0
```

```
    x[0] = self.x0
    for i in range(len(y) - 1):
        x_next, y_next = self.predicator(h, x[i], y[i])
        x[i + 1], y[i + 1] = self.corrector(h, x[i], y[i], x_next, y_next)
    return t, x, y, 2
```

Графики погрешностей:

Сравнительная таблица

Для $eps = 0.001$

Метод	Шаг	Количество итераций
Эйлера	0.000390625	8
Рунге – Кутты	$4.8e-05$	11
Адамс – Моултон	0.003125	5

Делаем вывод, что для данной задачи лучше всего использовать неявный метод Адамса-Моултона.

▼ Вся программа:

```
import numpy as np
import matplotlib.pyplot as plt


def f1_test(x, y, a, b):
    return a*x + b*y
```

```
def f2_test(x, y, c, d):
    return c*x + d*y

class Solution(object):

    def __init__(self, a, b, c, d, x0, y0):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

        self.x0 = x0
        self.y0 = y0

        self.T0 = 0
        self.T = 1

    @staticmethod
    def iteration(h, f, x, y, a, b):
        k1 = h*f(x, y, a, b)
        k2 = h*f(x + h/2, y + k1/2, a, b)
        k3 = h*f(x + 3/4*h, y + 3/4*k2, a, b)
        k = 1/9*(2*k1 + 3*k2 + 4*k3)
        return k

    @staticmethod
    def find_e(y1, y2, x1, x2, p):
        r1 = np.array(abs(y1 - y2)/(2**p -1))
        e1 = max(r1)
        r2 = np.array(abs(x1 - x2)/(2**p -1))
        e2 = max(r2)
        return max(e1, e2)

    def adam_solver(self, h):
        t = np.arange(self.T0, self.T, h)
        y = np.zeros(len(t))
        x = np.zeros(len(t))
        y[0] = self.y0
        x[0] = self.x0
        for i in range(len(y) - 1):
            x_next, y_next = self.predicator_kutta(h, x[i], y[i])
            x[i + 1], y[i + 1] = self.corrector(h, x[i], y[i], x_next, y_next)
        return t, x, y, 2

    def euler_solver(self, h):
        t = np.arange(self.T0, self.T, h)
        y = np.zeros(len(t))
        x = np.zeros(len(t))
        y[0] = self.y0
        x[0] = self.x0
        for i in range(len(y) - 1):
            x[i + 1] = x[i] + h*(self.a*x[i] + self.b*y[i])
            y[i + 1] = y[i] + h*(self.c*x[i] + self.d*y[i])
        return t, x, y, 1

    def runge_kutta_solver(self, h = 0.1):
        t = np.arange(self.T0, self.T, h)
        x = np.zeros(len(t))
        y = np.zeros(len(t))
        x[0] = self.x0
        y[0] = self.y0
        for i in range(len(t) - 1):
            x[i + 1] = x[i] + self.iteration(h, f1_test, x[i], y[i], self.a, self.b)
            y[i + 1] = y[i] + self.iteration(h, f2_test, x[i], y[i], self.c, self.d)
        return t, x, y, 3

    def acc(self, solver, eps):
        k = 0
        h = 0.2
        t1, x_1, y_1, p = solver(h/2)
        y1 = y_1[::2]
        x1 = x_1[::2]
        t1, x2, y2 = solver(h)[0], solver(h)[1], solver(h)[2]
```

```
        e = self.find_e(y1, y2, x1, x2, p)
        h = h/2

        while(e >= eps):
            k += 1
            h /= 2
            y2 = y_1
            x2 = x_1
            t1, x_1, y_1 = solver(h)[0], solver(h)[1], solver(h)[2]
            y1 = y_1[::2]
            x1 = x_1[::2]
            e = self.find_e(y1, y2, x1, x2, p)
        print("k", k, "h = ", h)
        return t1, x_1, y_1

    def predicator(self, h, x_pred, y_pred):
        x = x_pred + h*(self.a*x_pred + self.b*y_pred)
        y = y_pred + h*(self.c*x_pred + self.d*y_pred)
        return x, y

    def corrector(self, h, x_pred, y_pred, x_next, y_next):
        x = x_pred + h/2*(self.a*x_pred + self.b*y_pred + self.a*x_next + self.b*y_next)
        y = y_pred + h/2*(self.c*x_pred + self.d*y_pred + self.c*x_next + self.d*y_next)
        return x, y

    def predicator_kutta(self, h, x_pred, y_pred):
        x = x_pred + self.iteration(h, f1_test, x_pred, y_pred, self.a, self.b)
        y = y_pred + self.iteration(h, f2_test, x_pred, y_pred, self.c, self.d)
        return x, y


def test_solution(t):
    return 4*np.exp(-t)-np.exp(2*t), np.exp(-t)-np.exp(2*t)

def get_plot(x, y, label, color = 'darkblue'):
    plt.plot(x, y, ls="-", label = label, color = color)
    plt.legend()
    plt.savefig("2.png", dpi = 500)
    plt.show()

if __name__ == '__main__':
    a = -2
    b = 4
    c = -1
    d = 3
    x0 = 3
    y0 = 0

    eps = 0.001
    solution = Solution(a, b, c, d, x0, y0)

    t, x, y = solution.acc(solution.euler_solver, eps)
    get_plot(t, x - test_solution(t)[0], "Погрешность Р-К по x")
    get_plot(t, y - test_solution(t)[1], "Погрешность Р-К по y")


    t, x, y = solution.acc(solution.runge_kutta_solver, eps)
    get_plot(t, x - test_solution(t)[0], "Погрешность Эйлер по x")
    get_plot(t, y - test_solution(t)[1], "Погрешность Эйлер по y")

    t, x, y = solution.acc(solution.adam_solver, eps)
    get_plot(t, x - test_solution(t)[0], "Погрешность Адама по x")
    get_plot(t, y - test_solution(t)[1], "Погрешность Адама по y")
```

3. Аналитическое решение

   Приведения к неявному виду:

$\begin{cases} \frac{dx}{dt} = x(a-by) \ (1) \\ \frac{dy}{dt} = -y(c-dx) \ (2) \end{cases}$
$t \in [t_0, T], t_0 = 0$

Преобразуем систему $(2) : (1)$

$$\frac{dy}{dx} = \frac{-y(c-d\cdot x)}{x(a-by)}$$

$$-\frac{(a-by)}{y}dy = \frac{(c-d\cdot x)}{x}dx$$

Интегрируя получаем

$$by - a\cdot ln(y) + C_1 = c\cdot ln(x) - d\cdot x + C_2$$
$$by - a\cdot ln(y) - c\cdot ln(x) + d\cdot x + C_1 - C_2 = 0$$

$C = \mathrm{C}_1 + \mathrm{C}_2$ - это константа, зависящая от начальных условий, вообще говоря $C_1 = by(0) - a\cdot ln(y(0))$, $C_2 = c\cdot ln(x(0)) - d\cdot x(0)$

Зададим начальные условия:

$a = 1.4, b = 0.4, c = 1.4, d = 0.4, x(0) = 1, y(0) = 1$

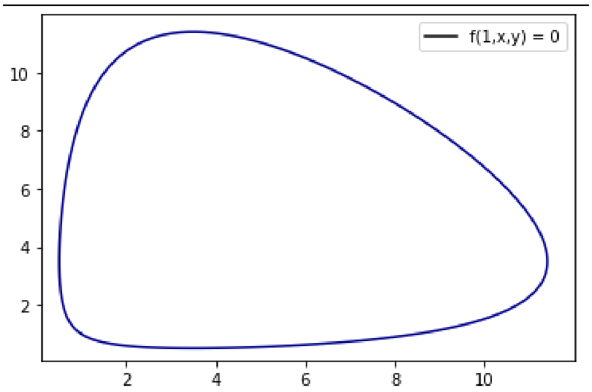
Тогда $C_1 = by(0) - a\cdot ln(y(0)) = b = 0.4$

$C_2 = c\cdot ln(x(0)) - d\cdot x(0) = -d = -0.4$

Можем построить график неявной функции

▼ Код

```
import matplotlib.pyplot as plt
import numpy as np

delta = 0.0025
x = np.arange(0.1, 12, delta)
y = np.arange(0.1, 12, delta)
p, q = np.meshgrid(x, y)
# define some function f(n,x,y)

f = lambda n, x, y: 0.4*y - 1.4*np.log(y) - 1.4*np.log(x) + 0.4*x - 0.8
z=f(0,p,q)

# plot contour line of f(1,x,y)==0
plt.contour(p, q, z , [0], colors="darkblue")

#make legend
proxy, = plt.plot([], color="k")
plt.legend(handles=[proxy], labels=["f(1,x,y) = 0"])
plt.show()
```



4. Численное решение задачи Коши при различных начальных данных и различных значениях коэффициентов a, b, c, d

   Решение ищем по неявному одношаговому методу Адамса-Моултона. Детально о его работе написано в п.2. Ниже приведен код программы.

▼ код программы

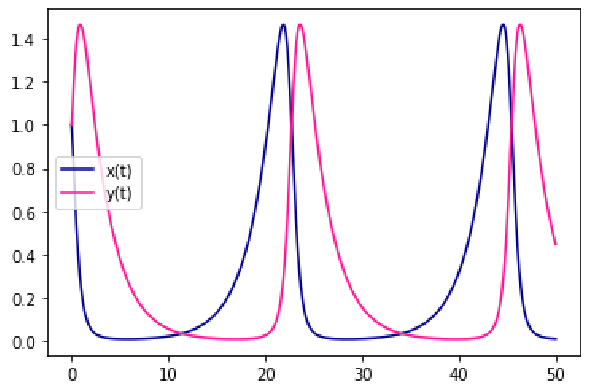
```
import numpy as np
import matplotlib.pyplot as plt

def f1(x, y, a, b):
    return a*x - b*y*x

def f2(x, y, c, d):
    return -c*y + d*x*y


class Solution(object):

    def __init__(self, a, b, c, d, x0, y0):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

        self.x0 = x0
        self.y0 = y0

        self.T0 = 0
        self.T = 25



    @staticmethod
    def find_e(y1, y2, x1, x2, p):
        r1 = np.array(abs(y1 - y2)/(2**p -1))
        e1 = max(r1)
        r2 = np.array(abs(x1 - x2)/(2**p -1))
        e2 = max(r2)
        return max(e1, e2)

    def adam_solver(self, h):
        t = np.arange(self.T0, self.T, h)
        y = np.zeros(len(t))
        x = np.zeros(len(t))
        y[0] = self.y0
        x[0] = self.x0
        for i in range(len(y) - 1):
            x_next, y_next = self.predicator(h, x[i], y[i])
            x[i + 1], y[i + 1] = self.corrector(h, x[i], y[i], x_next, y_next)
        return t, x, y, 2
```

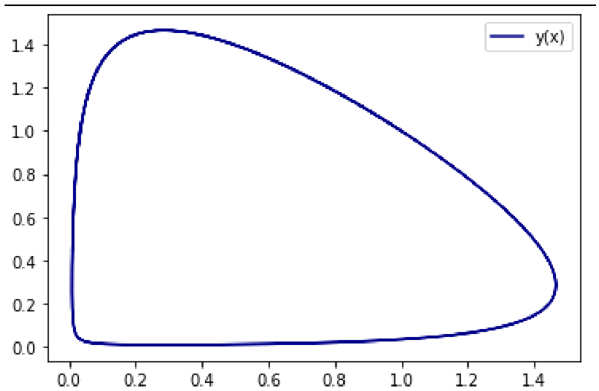
```
    def acc(self, solver, eps):
        k = 0
        h = 0.2
        t1, x_1, y_1, p = solver(h/2)
        y1 = y_1[::2]
        x1 = x_1[::2]
        t1, x2, y2 = solver(h)[0], solver(h)[1], solver(h)[2]

        e = self.find_e(y1, y2, x1, x2, p)
        h = h/2

        while(e >= eps):
            k += 1
            h /= 2
            y2 = y_1
            x2 = x_1
            t1, x_1, y_1 = solver(h)[0], solver(h)[1], solver(h)[2]
            y1 = y_1[::2]
            x1 = x_1[::2]
            e = self.find_e(y1, y2, x1, x2, p)
        print("k", k, "h = ", h)
        return t1, x_1, y_1

    def predicator(self, h, x_pred, y_pred):
        x = x_pred + h*f1(x_pred, y_pred, self.a, self.b)
        y = y_pred + h*f2(x_pred, y_pred, self.c, self.d)
        return x, y

    def predicator_1(self, h, x_pred, y_pred):
        half_y = x_pred + h/2 * f1(x_pred, y_pred, self.a, self.b)
        half_x = y_pred + h/2 * f2(x_pred, y_pred, self.c, self.d)
        x = x_pred + h*f1(half_x, half_y, self.a, self.b )
        y = y_pred + h*f2(half_x, half_y, self.c, self.d)
        return x, y

    def corrector(self, h, x_pred, y_pred, x_next, y_next):
        x = x_pred + h/2*(f1(x_pred, y_pred, self.a, self.b) + f1(x_next, y_next, self.a, self.b))
        y = y_pred + h/2*(f2(x_pred, y_pred, self.c, self.d) + f2(x_next, y_next, self.c, self.d))
        return x, y

def get_plot(x, y, label, color = 'darkblue'):
    plt.plot(x, y, ls="-", label = label, color = color)
    plt.legend()
    plt.savefig("test1.png", dpi = 500)
    plt.show()

def get_plot_2(x, y, z, label1, label2, color1 = 'darkblue', color2 = 'deeppink'):
    plt.plot(x, y, ls="-", label = label1, color = color1)
    plt.plot(x, z, ls="-", label = label2, color = color2)
    plt.legend()
    plt.savefig("test1.png", dpi = 500)
    plt.show()

if __name__ == '__main__':
    a = 0.4
    b = 1.4
    c = 0.4
    d = 1.4
    x0 = 1
    y0 = 1

    eps = 0.001
    solution = Solution(a, b, c, d, x0, y0)


    t, x, y = solution.acc(solution.adam_solver, eps)
    get_plot_2(t, x, y, "x(t)", "y(t)")
    get_plot(x, y, "y(x)")
```

5. Постройте графики зависимости решения $x(t)$ и $y(t)$ от времени, а также фазовый портрет (в переменных x, y). Сравнительная таблица, выводы.

Рассмотрим частный случай, когда коэффициенты a=c, b =d

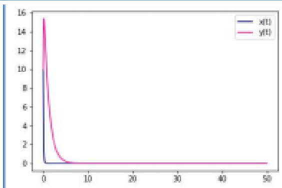
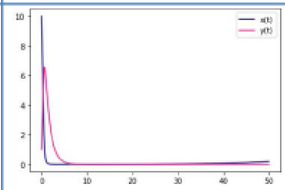
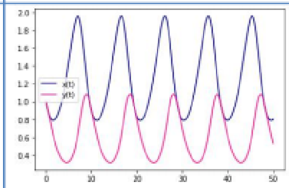
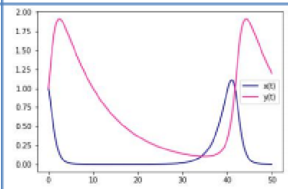
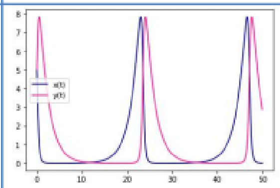
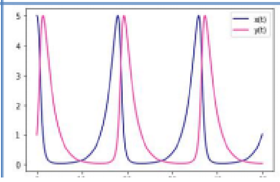
При $a = 0.4, b = 1.4, c = 0.4, d = 1.4, x(0) = 1, y(0) = 1$





Видим, что численность циклична.

Составим сравнительную таблицу с другими исходными данными.

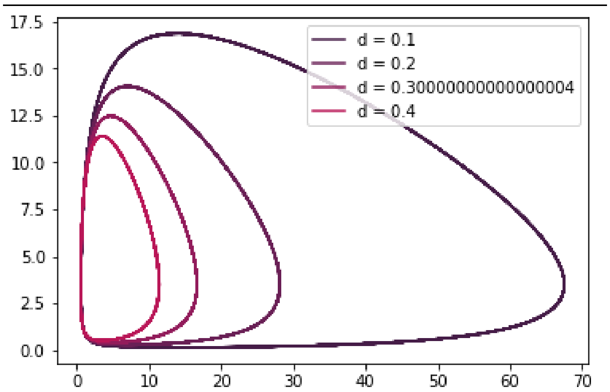
№	a	b	c	d	x_0	y_0	График	Вывод
1	0.1	0.8	0.9	0.7	10	10		**Скорость прироста жертв - a** в несколько раз меньше, чем остальные коэффициенты. Тогда даже при уменьшение в 10 раз начального количества хищников, численность обоих видов уходит в 0.
2	0.1	0.8	0.9	0.7	10	1		
3	0.5	0.8	0.9	0.7	1	1		Увеличим в 5 раз по сравнению с №1 коэффициент **а**, тогда появляется цикличность.
4	0.5	0.8	0.1	0.7	1	1		Уменьшим коэффициент **c - убыли хищников.** Цикличность сохранилась, но период увеличился.
5	0.5	0.5	0.5	0.5	5	5		Пусть коэффициенты равны. Посмотрим, как влияет на чиленность уменьшение начального кол-ва хищников. Видим, что даже уменьшая в 5 раз начальное значени, процесс остаётся цикличным.
6	0.5	0.5	0.5	0.5	5	1		

Построим фазовые портреты

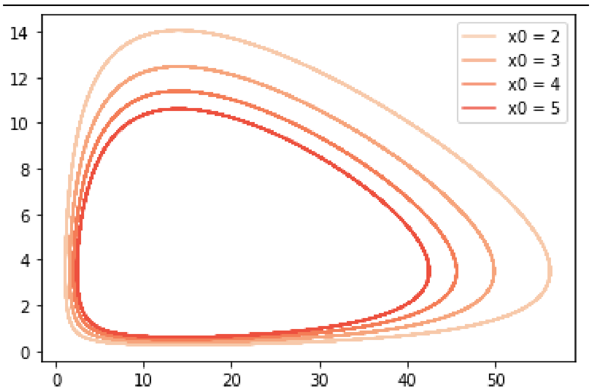
Оставим неизменными коэффициенты

$$a = 1.4, b = 0.4, c = 1.4, x_0 = 1, y_0 = 1$$

При этом будем менять d от 0.1 до 0.5. То есть по графикам видно, что при увеличение коэффициента d период будет возрастать.

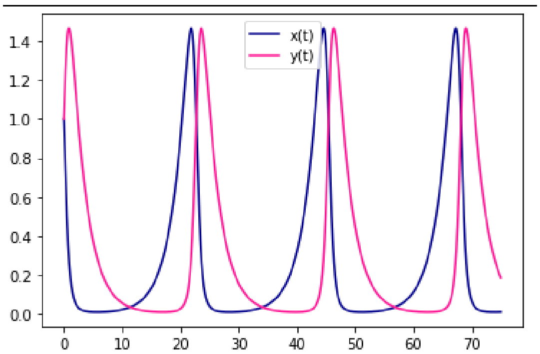


Можем также исследовать зависимость фазового портрета от начальных данных. Фиксируем все переменные, меняя x_0



6. Приблизительно определите длительность цикла, получившегося в п.3 (усреднением по достаточно большому кол-ву циклов). Используя информацию о точности решения задачи Коши, оцените точность рассчитанного периода циклов.

   $a = 0.4, b = 1.4, c = 0.4, d = 1.4, x(0) = 1, y(0) = 1$



   По графику получаем, что период $T \approx 22$. Решение искали с точностью $eps = 0.001$.

7. Найдите (аналитически) стационарные решения системы. Попытайтесь подобрать значения коэффициентов, при которых решение выходит на стационарный режим.

   Стационарные точки получаем, когда

$$\frac{dx}{dt} = 0 \text{ и } \frac{dy}{dt} = 0$$
$$\begin{cases} 0 = x(a - by) \ (1) \\ 0 = -y(c - dx) \ (2) \end{cases}$$

Раскроем скобки

$$\begin{cases} 0 = xa - bxy \ (1) \\ 0 = -yc + dxy \ (2) \end{cases}$$

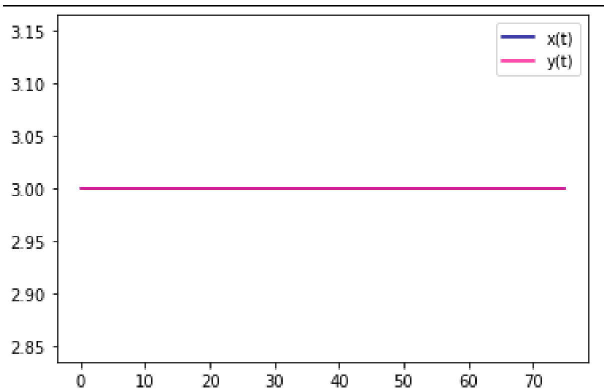
$(1) : x, (2) : y-$так как x, y не обращаются в ноль. Отдельно рассматривать решение $x = 0, y = 0$нет смысла. Тогда поулчаем, что

$$\begin{cases} y = \frac{a}{b} \\ x = \frac{c}{d} \end{cases}$$

Вот, например,случай

$a = 0.6, b = 0.2, c = 0.6, d = 0.2, x_0 = 3, y_0 = 3$. То есть это случай, при котором

$$\begin{cases} y(0) = \frac{a}{b} \\ x(0) = \frac{c}{d} \end{cases}$$



Что совпадает с математичким представлением. Численность не меняется.

Решение выходит на стационарный уровень, когда

$$\frac{a}{b} = \frac{b}{c}$$

То есть это случай, который был подобран в пункте ранее.

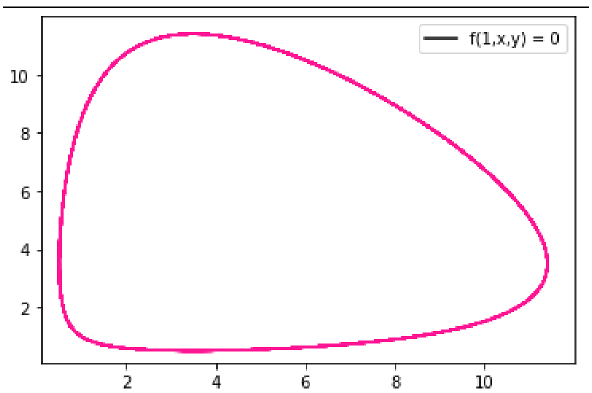
8. Сравнение фазовых портретов

   Сравним найденное приближенное решение с аналитическим решением.

   Розовым цветом обозначен график приближенного цвета, а синим, которого почти не видно обозначен график точного решения.

   ▼ Код

```
if __name__ == '__main__':
    a = 1.4
    b = 0.4
    c = 1.4
    d = 0.4
    x0 = 1
    y0 = 1

    eps = 0.001
    solution = Solution(a, b, c, d, x0, y0)
    t, x1, y1 = solution.acc(solution.adam_solver, eps)

    plt.plot(x1, y1, ls="-", label = "la", color = "deeppink")

    delta = 0.0025
    x = np.arange(0.1, 12, delta)
    y = np.arange(0.1, 12, delta)
    p, q = np.meshgrid(x, y)

    f = lambda n, x, y: 0.4*y - 1.4*np.log(y) - 1.4*np.log(x) + 0.4*x - 0.78
    z=f(0,p,q)

    plt.contour(p, q, z , [0], colors="darkblue")

    proxy, = plt.plot([], color="k")
    plt.legend(handles=[proxy], labels=["f(1,x,y) = 0"])
    plt.show()
```



9. Вывод

Аналитическое решение задачи может быть представлено только в неявном виде. Для нахождения приближённого решения был выбран метод прогноза(модифицированным методом Эйлера) и коррекции(методом Адамса-Моултона). Работоспособность алгоритма проверена на тестовом примере. Также графики аналитического и приближённого решений были выведены в п.8. В ходе решения задачи были подобраны коэффициенты, при которых функции решение выходит на стационарный режим. При данных коэффициентах функции изменения численности населений хищников и жертв периодичны.

10. Литература

The behaviour and attractiveness of the Lotka-Volterra equations - Timon Idema