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

ФАКУЛЬТЕТ «Робототехники и комплексной автоматизации»
КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

по дисциплине «Методы математического моделирования сложных процессов и систем»

Студент:	Юрченко Александр Алексеевич
Группа:	РК6-11М
Тип задания:	лабораторная работа
Тема:	Модель, описывающая реакцию пациента на лекарственный препарат

Студент _____ (подпись, дата) Юрченко А.А. (Фамилия, И.О.)

Преподаватель _____ (подпись, дата) Соколов А.П. (Фамилия, И.О.)

Москва, 2025

Содержание

Модель, описывающая реакцию пациента на лекарственный препарат

Задание на лабораторную работу

Требуется(базовая часть).

1. Программно реализовать требуемые в варианте задания численные методы решения СОДУ на языке C++, собрать соответствующие библиотеки динамической компоновки (каждый метод в своей библиотеке). Объяснить каким образом разработанная программа позволит подключать реализации других методов решения СОДУ без перекомпиляции исходных кодов.

2. Численно решить поставленную задачу Коши с помощью реализованного ПО, используя разные численные методы.

3. Полученные результаты представить графически на одной координатной плоскости и включить в отчёте.

4. Результат решения задачи одним методом должен сохраняться в виде одного текстового файла с разделителями – .csv формат (при сдаче должно быть представлено).

Требуется (продвинутая часть).

5. Если возможно, аналитически решить поставленную задачу Коши (эталонную) и далее аналитическое решение считать эталонным. В случае невозможности получить аналитическое решения объяснить причину и далее в качестве эталонного решения использовать численное решение эталонной СОДУ, определённой согласно варианту.

6. Провести сравнение численных решений с эталонным (результат представить графически).

7. Обеспечить возможность определять неточно значение двух скалярных входных параметров.

8. Представить в отчёте описание алгоритма решения поставленной задачи при двух интервально заданном параметре. Описать каким образом возможно доработать предложенный алгоритм для случая произвольного числа скалярных параметров, заданных в виде их интервальных оценок.

Дано:

$$\begin{cases} \dfrac{dy_1}{dt} = -k_{abs} \cdot y_1; \\ \dfrac{dy_2}{dt} = k_{abs} \cdot y_1 - (k_1 + k_{el}) \cdot y_2 + k_2 \cdot y_3; \\ \dfrac{dy_3}{dt} = k_1 \cdot y_2 - k_2 \cdot y_3; \\ \dfrac{dx_1}{dt} = k_{11} \cdot y_2 - k_{12} \cdot x_1 + k_{13}; \\ \dfrac{dx_2}{dt} = k_{21} \cdot y_2 - k_{22} \cdot x_2 + k_{23}; \\ \dfrac{dx_3}{dt} = k_{31} \cdot y_2 - k_{32} \cdot x_3 + k_{33}. \end{cases} \tag{1}$$

В зависимости от различных комбинаций коэффициентов k модель (1) описывает реакцию пациента на лекарственный препарат для случаев без прогрессирования заболевания M и с прогрессированием M + (Табл. 1).

Таблица 1. Значения коэффициентов k для случаев с и без прогрессирования заболевания

Коэффициент	**Значение,** M^-	**Значение,** M^+
k_{abs}	1.029	2.481
k_{el}	0.670	0.649
k_1	0.118	0.949
k_2	1.981	2.757
k_{11}	0.201	2.900
k_{12}	0.295	0.466
k_{13}	8.998	1.376
k_{21}	0.138	0.531
k_{22}	0.347	0.869
k_{23}	0.913	2.814
k_{31}	0.687	1.490
k_{32}	0.580	1.666
k_{33}	2.343	7.060

Требуется

1. Реализовать следующие численные методы решения СОДУ: а) Рунге-Кутты 4-го порядка, б) Адамса-Башфорта-Моултона 4-го порядка, в) Милна-Симпсона.

2. Полученные графики зависимостей xi(t) при разных комбинациях k (для M, M+) выводить на одной координатной плоскости.

3. Для случае интервально задаваемых параметров: поставить обратные задачи идентификации комбинаций коэффициентов k· моделей M и M+. В качестве эталон-

ных решений использовать получаемые в результате решения прямых задач с коэффициентам, заданными условием задания.

4. Предложить метод решения обратной задачи и описать его в отчете.

Цель выполнения лабораторной работы

Целью данной работы является изучение принципов создания библиотек динамической компоновки на примере решения СОДУ различными численными методами, реализуемыми независимо в виде функций, экспортируемых из данных библиотек.

Выполненные задачи

1. На языке программирования C++ реализованы функция, возвращающая дискретную траекторию системы ОДУ, где дискретная траектория строится с помощью метода Рунге-Кутты 4-го порядка.

2. На языке программирования C++ реализована функция, возвращающая дискретную траекторию системы ОДУ, где дискретная траектория строится с помощью метода Адамса-Башфорта-Моултона 4-го порядка.

3. На языке программирования C++ реализована функция, возвращающая дискретную траекторию системы ОДУ, где дискретная траектория строится с помощью метода Милна-Симпсона.

4. На языке программирования Python реализована программа для вывода траектории системы ОДУ на координатной плоскости.

5. Для каждого из реализованных методов численно найдена траектория заданной динамической системы (**??**). Произведено сравнение результатов каждого метода.

1 Реализация численных методов решеения СОДУ

Метод Рунге-Кутты 4-го порядка

Формулировка метода Рунге-Кутта 4-го порядка для системы ОДУ 1-го порядка имеет вид:

$$\begin{gathered}
\mathbf{w_0} = \boldsymbol{\alpha}, \\
\mathbf{k_1} = h \cdot \mathbf{f}(t_i, \mathbf{w_i}), \\
\mathbf{k_2} = h \cdot \mathbf{f}(t_i + \tfrac{h}{2}, \mathbf{w_i} + \tfrac{1}{2}\mathbf{k_1}), \\
\mathbf{k_3} = h \cdot \mathbf{f}(t_i + \tfrac{h}{2}, \mathbf{w_i} + \tfrac{1}{2}\mathbf{k_2}), \\
\mathbf{k_4} = h \cdot \mathbf{f}(t_i + h, \mathbf{w_i} + \mathbf{k_3}), \\
\mathbf{w_{i+1}} = \mathbf{w_i} + \tfrac{1}{6}(\mathbf{k_1} + 2\mathbf{k_2} + 2\mathbf{k_3} + \mathbf{k_4}), i = 0, 1, ..., m-1,
\end{gathered} \tag{2}$$

где $\boldsymbol{\alpha}$ – вектор начальных условий $y_1(a) = \alpha_1,\ y_2(a) = \alpha_2,\ ...,\ y_n(a) = \alpha_n$ размера n, а $\mathbf{w_i}$ – вектор решений ОДУ размера n на $i-1$ шаге.

Реализована функция, которая получает на вход указатель на систему дифференциальных уравнений, вектор начальных условий, время начала, время конца, шаг и указатель на дескриптор .csv файла для записи дискретных решений.

Программная реализация данного метода в качестве функции в библиотеке динамической компоновки представлена в листинге 1:

Листинг 1: Определение функции `RungeKutta` численного решения задачи Коши для системы ОДУ методом Рунге-Кутты 4-го порядка.

```
void RungeKutta(std::vector<double> f(double, std::vector<double>&), std::vector<double>& y,
double t0, double t1, double h, std::ofstream* csv)
{
    double t = t0;
    for (int i=0; i<y.size()-1; i++)
        *csv << y[i] << ",  ";
    *csv << y[y.size()-1] << std::endl;

    while (t < t1)
    {
        std::vector<double> k1 = f(t, y);
        std::vector<double> yk1;
        for (int i=0; i<y.size(); i++)
            yk1.push_back(y[i]+h*k1[i]/2);
        std::vector<double> k2 = f(t + h / 2, yk1);
        std::vector<double> yk2;
        for (int i=0; i<y.size(); i++)
            yk2.push_back(y[i]+h*k2[i]/2);
        std::vector<double> k3 = f(t + h / 2, yk2);
        std::vector<double> yk3;
        for (int i=0; i<y.size(); i++)
            yk3.push_back(y[i]+h*k2[i]/2);
        std::vector<double> k4 = f(t + h, yk3);

        for (int i=0; i<y.size(); i++)
        {
            double delta =  h / 6 * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]);
            y[i] += delta;
        }
        t += h;
        for (int i=0; i<y.size()-1; i++)
            *csv << y[i] << ",  ";
        *csv << y[y.size()-1] << std::endl;
   }
}
```

Метод Адамса-Башфорта-Моултона 4-го порядка

Метод Адамса–Башфорта–Моултона является методом, построенным по схеме предиктор-корректор за счет комбинации многошаговых методов Адамса–Башфорта и Адамса–Моултона одного порядка точности, что повышает устойчивость численной схемы. Например, для методов 4-го порядка мы получаем Метод Адамса–Башфорта–Моултона 4-го порядка следующего вида:

$$\begin{gathered}\mathbf{w_0} = \boldsymbol{\alpha_0}, \mathbf{w_1} = \boldsymbol{\alpha_1}, \mathbf{w_2} = \boldsymbol{\alpha_2}, \mathbf{w_3} = \boldsymbol{\alpha_3}, \\ \tilde{\mathbf{w}}_{\mathbf{i+1}} = \mathbf{w_i} + \frac{h}{24}\Big[55\,\mathbf{f}(t_i, \mathbf{w_i}) - 59\,\mathbf{f}(t_{i-1}, \mathbf{w_{i-1}}) + 37\,\mathbf{f}(t_{i-2}, \mathbf{w_{i-2}}) - 9\,\mathbf{f}(t_{i-3}, \mathbf{w_{i-3}})\Big], \\ \mathbf{w_{i+1}} = \mathbf{w_i} + \frac{h}{24}\Big[9\,\mathbf{f}(t_{i+1}, \tilde{\mathbf{w}}_{\mathbf{i+1}}) + 19\,\mathbf{f}(t_i, \mathbf{w_i}) - 5\,\mathbf{f}(t_{i-1}, \mathbf{w_{i-1}}) + \mathbf{f}(t_{i-2}, \mathbf{w_{i-2}})\Big].\end{gathered} \tag{3}$$

Реализованная функция принимает указатель на систему дифференциальных уравнений, вектор начальных условий, начальный и конечный моменты времени, шаг интегрирования и указатель на открытый `.csv`-файл, в который записываются значения решения на каждом шаге. Первые три шага вычисляются по схеме Рунге–Кутты 4-го порядка; далее вступает в действие схема предиктор-корректор (3)

Программная реализация данного метода в качестве функции в библиотеке динамической компоновки представлена в листинге 2:

Листинг 2: Определение функции `Adams_Bashforth_Moulton` численного решения задачи Коши для системы ОДУ методом Адамса-Башфорта-Моултона 4-го порядка.

```
void Adams_Bashforth_Moulton(std::vector<double> f(double, std::vector<double>&),
                             std::vector<double>& y,
                             double t0, double t1, double h,
                             std::ofstream* csv)
{
    // запись начального вектора
    for (int i = 0; i < y.size() - 1; ++i) *csv << y[i] << ", ";
    *csv << y.back() << '\n';

    double t = t0;
    std::vector<double> y_1, y_2, y_3;   // предыдущие значения y
    y_1.resize(y.size());
    y_2.resize(y.size());
    y_3.resize(y.size());

    // Первые 4 шага - Рунге-Кутта 4-го порядка
    for (int step = 0; step < 3; ++step)
    {
        std::vector<double> k1 = f(t, y);
        std::vector<double> yk1(y.size());
        for (size_t i = 0; i < y.size(); ++i) yk1[i] = y[i] + h * k1[i] / 2.0;

        std::vector<double> k2 = f(t + h / 2.0, yk1);
        std::vector<double> yk2(y.size());
        for (size_t i = 0; i < y.size(); ++i) yk2[i] = y[i] + h * k2[i] / 2.0;

        std::vector<double> k3 = f(t + h / 2.0, yk2);
        std::vector<double> yk3(y.size());
        for (size_t i = 0; i < y.size(); ++i) yk3[i] = y[i] + h * k3[i] / 2.0;

        std::vector<double> k4 = f(t + h, yk3);

        for (size_t i = 0; i < y.size(); ++i)
            y[i] += h / 6.0 * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]);

        // сохраняем историю
        if (step == 0)      y_3 = y;
        else if (step == 1) y_2 = y;

        t += h;
        for (size_t i = 0; i < y.size() - 1; ++i) *csv << y[i] << ", ";
        *csv << y.back() << '\n';
    }

    // Основной цикл: предиктор + корректор AM-4 (одна итерация)
    while (t < t1)
    {
        std::vector<double> k1 = f(t, y);
        std::vector<double> k2 = f(t - h, y_3);
        std::vector<double> k3 = f(t - 2 * h, y_2);
```

```
        std::vector<double> k4 = f(t - 3 * h, y_1);

        std::vector<double> y_pred(y.size());
        for (size_t i = 0; i < y.size(); ++i)
            y_pred[i] = y[i] + h / 24.0 * (55 * k1[i] - 59 * k2[i] + 37 * k3[i] - 9 * k4[i]);

        // корректор Адамса-Мултона 4-го порядка
        k1 = f(t + h, y_pred);
        k2 = f(t, y);
        k3 = f(t - h, y_3);
        k4 = f(t - 2 * h, y_2);

        y_1 = y_2;
        y_2 = y_3;
        y_3 = y;

        for (size_t i = 0; i < y.size(); ++i)
            y[i] += h / 24.0 * (9 * k1[i] + 19 * k2[i] - 5 * k3[i] + k4[i]);

        t += h;
        for (size_t i = 0; i < y.size() - 1; ++i) *csv << y[i] << ", ";
        *csv << y.back() << '\n';
    }
}
```

Метод Милна-Симпсона

Метод Милна-Симпсона является методом, построенным по схеме предиктор-корректор за счет комбинации многошагового явного метода Милна и неявного многшагового метода Симпсона. Данный метод имеет следующий вид (4)

$$\begin{gathered}\mathbf{w_0} = \boldsymbol{\alpha_0}, \mathbf{w_1} = \boldsymbol{\alpha_1}, \mathbf{w_2} = \boldsymbol{\alpha_2}, \mathbf{w_3} = \boldsymbol{\alpha_3}, \\ \tilde{\mathbf{w}}_{\mathbf{i+1}} = \mathbf{w}_{i-1} + \frac{4h}{3}\Big[2\mathbf{f}(t_i, \mathbf{w}_i) - \mathbf{f}(t_{i-1}, \mathbf{w}_{i-1}) + 2\mathbf{f}(t_{i-2}, \mathbf{w}_{i-2})\Big], \\ \mathbf{w}_{i+1} = \mathbf{w}_{i-1} + \frac{h}{3}\Big[\mathbf{f}(t_{i+1}, \tilde{\mathbf{w}}_{\mathbf{i+1}} + 4\mathbf{f}(t_i, \mathbf{w}_i) + \mathbf{f}(t_{i-1}, \mathbf{w}_{i-1})\Big].\end{gathered} \tag{4}$$

Реализованная функция принимает указатель на систему дифференциальных уравнений, вектор начальных условий, начальный и конечный моменты времени, шаг интегрирования и указатель на открытый `.csv`-файл для записи дискретных решений. Программная реализация представлена в листинге 3.

Листинг 3: определение функции `Milne_Simpson` численного решения задачи Коши для системы ОДУ методом Милна–Симпсона.

```
void Milne_Simpson(std::vector<double> (*f)(double, const std::vector<double>&), std::vector<double>& y,
{
    for (int i=0; i<y.size()-1; i++)
        *csv << y[i] << ",  ";
    *csv << y[y.size()-1] << std::endl;

    double t = t0;
    std::vector<double> yim3;
    yim3.resize(y.size());
    std::vector<double> yim2;
    yim2.resize(y.size());
```

```
std::vector<double> yim1;
yim1.resize(y.size());
std::vector<double> yi;
yi.resize(y.size());

yim3=y;

// Рунге-Кутта для первых 3х шагов
for (int i=0; i<3; i++)
{
    std::vector<double> k1 = f(t, y);
    std::vector<double> yk1;
    for (int i=0; i<y.size(); i++)
        yk1.push_back(y[i]+h*k1[i]/2);
    std::vector<double> k2 = f(t + h / 2, yk1);
    std::vector<double> yk2;
    for (int i=0; i<y.size(); i++)
        yk2.push_back(y[i]+h*k2[i]/2);
    std::vector<double> k3 = f(t + h / 2, yk2);
    std::vector<double> yk3;
    for (int i=0; i<y.size(); i++)
        yk3.push_back(y[i]+h*k2[i]/2);
    std::vector<double> k4 = f(t + h, yk3);

    for (int i=0; i<y.size(); i++)
    {
        double delta =  h / 6 * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]);
        y[i] += delta;
    }

    switch (i)
    {
    case 0:
        {
            yim2=y;
            break;
        }
    case 1:
        {
            yim1=y;
            break;
        }
    }
    t += h;
    for (int i=0; i<y.size()-1; i++)
        *csv << y[i] << ",  ";
    *csv << y[y.size()-1] << std::endl;
}
yi = y;

// Милн-Симпсон
while (t < t1)
{
    //предиктор - Милн
    std::vector<double> ki = f(t, yi);
    std::vector<double> kiM1 = f(t-h, yim1);
    std::vector<double> kiM2 = f(t-2*h, yim2);
    std::vector<double> y_;
    y_.resize(y.size());

    for (int i=0; i<y.size(); i++)
    {
        y_[i] = yim3[i] + 4*h/3 * (2*ki[i] - kiM1[i] + 2*kiM2[i]);
    }

    std::vector<double> kiP1 = f(t+h, y_);
```

```
        for (int i=0; i<y.size(); i++)
        {
            y[i] = yim1[i] + h/3 * (kiP1[i] + 4*ki[i] + kiM1[i]);
        }
        t+=h;

        yim3 = yim2;
        yim2 = yim1;
        yim1 = yi;
        yi = y;

        for (int i=0; i<y.size()-1; i++)
            *csv << y[i] << ",  ";
        *csv << y[y.size()-1] << std::endl;

    }
}
```

2 Реализация сборки библиотек динамической компоновки

Библиотеки динамической компоновки (DLL, Dynamic-Link Libraries) — это файлы, содержащий исполняемый код и данные, которые могут использоваться сразу несколькими программами одновременно. В Windows они имеют расширение .dll, в Unix-подобных системах — .so (shared object). Представленные выше функции были реализованы на основе DLL операционной системы Windows. DLL — это библиотеки позднего связывания. Позднее связывание означает, что адреса функций и сам код библиотеки не встраиваются в исполняемый файл на этапе компиляции или сборки, а находятся и подключаются уже после запуска программы, в момент, когда операционная система загружает DLL в память.

В качестве инструмента разработки была выбрана среда разработки CLion. Она поддерживает управление проектами, возможность сборки и отладки как исполняемых файлов, так и библиотек динамической компоновки (DLL).

CLion не использует собственные системы сборки, а полагается на CMake: в корне проекта достаточно описать CMakeLists.txt, и IDE сама вызовет CMake, сгенерирует проектные файлы, подхватит флаги компиляции, пути к заголовкам и библиотекам, после чего обеспечит полноценную навигацию, рефакторинг и отладку.

Для Windows-версии DLL в настройках конфигурации проекта явно задан генератор «Visual Studio 17 2022» и соответствующий тулчейн MSVC 17. Это значит, что CLion делегирует всю работу по компиляции и линковке фирменному компилятору Microsoft: исходники собираются в *.obj-файлы, затем линковщик LINK.exe формирует из них динамическую библиотеку *.dll и сопутствующий import-lib *.lib, который впоследствии можно подключать к внешним проектам.

Инструкции сборки Cmake для проекта представлены в листинге 4.

Листинг 3: Инструкции Cmake по сборке проекта

```
cmake_minimum_required(VERSION 3.31)
project(mms1)
```

```
set(CMAKE_CXX_STANDARD 20)

add_executable(main main.cpp)

add_library(RungeKutta SHARED RungeKutta.cpp)

add_library(Adams_Bashforth_Moulton SHARED Adams_Bashforth_Moulton.cpp)

add_library(Milne_Simpson SHARED Milne_Simpson.cpp)
```

Функция `add_executable` указывает на сборку компонента `main.cpp` проекта в .exe файл. А функции `add_library(SHARED)` указывают компилятору на сборку динамических библиотек (.dll) из исходных файлов `RungeKutta.cpp`, `Adams_Bashforth_Moulton.cpp` и `Milne_Simpson.cpp`.

Для подключения .dll библиотек к проекту нужно подключить директиву `<windows.h>`. Код программы по объявлению и вызову функций для решения задачи Коши представлены в листинге 5.

Листинг 3: Подключение внешних функций из .dll

```
#include <windows.h>

HMODULE dll1 = LoadLibrary("RungeKutta.dll");
typedef int FT1(std::vector<double>(double, std::vector<double>&), std::vector<double>&,
double, double, double, std::ofstream*);
FT1* RungeKutta = (FT1*)GetProcAddress(dll1, "RungeKutta");

HMODULE dll2 = LoadLibrary("Adams_Bashforth_Moulton.dll");
typedef int FT2(std::vector<double>(double, std::vector<double>&), std::vector<double>&,
double, double, double, std::ofstream*);
FT2* Adams_Bashforth_Moulton = (FT2*)GetProcAddress(dll2, "Adams_Bashforth_Moulton");

HMODULE dll3 = LoadLibrary("Milne_Simpson.dll");
typedef int FT3(std::vector<double>(double, std::vector<double>&), std::vector<double>&,
double, double, double, std::ofstream*);
FT3* Milne_Simpson = (FT3*)GetProcAddress(dll3, "Milne_Simpson");

    RungeKutta(System, y, t0, t1, h, &csv1);
    Adams_Bashforth_Moulton(System, y, t0, t1, h, &csv3);
    Milne_Simpson(System, y, t0, t1, h, &csv2);
```

Для корректного экспорта функций из DLL и их импорта в EXE проект используют макросы: `extern "C" __declspec(dllexport) void Milne_Simpson()`. Когда DLL проект компилируется, функции помечаются как экспортируемые (dllexport). В других проектах, подключающих DLL, этот макрос меняет поведение на импорт (dllimport).

Итоговая структура взаимодействия выглядит следующим образом: проект DLL содержит реализованные функции и экспортирует их с помощью специальных макросов. Компилятор формирует динамическую библиотеку (DLL) и соответствующий файл LIB, который затем используется в основном EXE-проекте. Приложение подключает заголовочные файлы DLL и импортирует LIB, а при запуске загружает саму библиотеку, получая доступ к её функциям во время выполнения.

Такой подход позволяет создавать многократно используемые и легко обновляемые библиотеки: при изменении или замене DLL нет необходимости перекомпилировать основной проект. Это существенно упрощает поддержку и развитие программного обеспечения, поскольку изменения внутри библиотеки не затрагивают код исполняемого файла, обеспечивая при этом модульность и гибкость архитектуры.

Использование библиотек динамической компоновки (DLL) делает возможным подключение реализованных численных методов к основной программе непосредственно во время выполнения. Благодаря этому можно добавлять новые алгоритмы решения систем обыкновенных дифференциальных уравнений (СОДУ) без пересборки основного приложения — достаточно создать новую DLL с нужной функцией и указать путь к ней. При запуске программа динамически подгружает выбранную библиотеку и обращается к её функциям, что устраняет необходимость перекомпиляции EXE-файла. В результате система приобретает модульную и легко расширяемую архитектуру.

3 Получение аналитического решения СОДУ

Выделим часть исходной системы:

$$\begin{cases} \dfrac{dy_1}{dt} = -k_{abs} \cdot y_1; \\ \dfrac{dy_2}{dt} = k_{abs} \cdot y_1 - (k_1 + k_{el}) \cdot y_2 + k_2 \cdot y_3; \\ \dfrac{dy_3}{dt} = k_1 \cdot y_2 - k_2 \cdot y_3; \end{cases} \tag{5}$$

Данная подсистема представляет собой систему линейных однородных дифференциальных уравнений, разре- шенных относительно производных, которую можно записать в виде:

$$\dot{\mathbf{y}}(t) = \mathbf{A}\mathbf{y}(t) \tag{6}$$

где $\mathbf{A} = a_{i,j}$ - матрица из коэффициентов системы, $\dot{\mathbf{y}}(t)$ - вектор производных, $\mathbf{y}(t)$ - вектор функций.

Систему (5) можно решить с помощью метода Эйлера. Решение системы (5) в общем виде имеет вид:

$$\mathbf{y}(t) = C_1 e^{\lambda_1 t}\mathbf{e_1} + C_2 e^{\lambda_2 t}\mathbf{e_2} + C_3 e^{\lambda_3 t}\mathbf{e_3} \tag{7}$$

где λ_i - собственные числа матрицы $\mathbf{A}$, $\mathbf{e_i}$ - собственные векторы матрицы $\mathbf{A}$, C_i - некоторые константы.

Для нахождения собственных чисел λ_i матрицы нужно решить следующее матричное уравнение (8):

$$|\mathbf{A} - \lambda\mathbf{E}| = 0 \tag{8}$$

Получаем:

$$\begin{vmatrix} -K_{abs} - \lambda & 0 & 0 \\ K_{abs} & -(K_1 + K_{el}) - \lambda & K_2 \\ 0 & K_1 & -K_2 - \lambda \end{vmatrix} = 0 \tag{9}$$

После решения данного уравнения получаем:

$$\begin{cases} \lambda_1 = -K_{abs}, \\ \lambda_2 = \frac{-(k_1+k_{el}+k_2)+\sqrt{(k_1+k_{el}+k_2)^2-4\,k_2\,k_{el}}}{2}, \\ \lambda_3 = \frac{-(k_1+k_{el}+k_2)-\sqrt{(k_1+k_{el}+k_2)^2-4\,k_2\,k_{el}}}{2}, \end{cases} \tag{10}$$

Для нахождения собственных векторов нужно решить слудующее матричное уравнение, подставив туда λ_i:

$$|\mathbf{A} - \lambda \mathbf{E}|\mathbf{e_i} = 0 \tag{11}$$

После подстановки λ получаем систему линейных алгебраических уравнений, которые решаются методом Гаусса в программной реализации. Константы C_i вычисляются путем подстановки начальных условий в получившиеся уравнения.

Оставшиеся 3 уравнения для $\dot{x_1}, \dot{x_2}, \dot{x_3}$ являются обыкновенными дифференциальными уравнениями с разделяющимися переменными. Переменная y_2 становится константой, так как вычисляется в первой подсистеме.

После интегрирования соответствующих уравнений получаем следующие выражения для $\dot{x_1}, \dot{x_2}, \dot{x_3}$:

$$\begin{cases} x_1 = C_1 e^{-K_{12}t} + \frac{K_{13}-K_{11}*y_2}{K_{12}}, \\ x_2 = C_2 e^{-K_{22}t} + \frac{K_{23}-K_{21}*y_2}{K_{22}}, \\ x_3 = C_3 e^{-K_{32}t} + \frac{K_{33}-K_{31}*y_2}{K_{32}} \end{cases} \tag{12}$$

где константы C_i вычисляются путем подставновки начальных условий в полученные уравнения и решения соответствующей СЛАУ.

Программная реализация рассчета дискретных значений аналитического решения СОДУ представлена в листинге 4.

Листинг 4: Аналитическое решение СОДУ

```
std::vector<double> analytical_solution(double t, std::vector<std::vector<double>> initial)
{
    // Initial - начальные условия;   задаются как initial = {t, c}, где yi(t) = c
    // Решение системы
    std::vector<double> y;
    y.resize(6);

    // Решение первой подсистемы из y1 y2 y3 методом Эйлера
```

```
    // Собственные числа лямбда |A-labmdaE| = 0
    double lambda_1 = Kabs[M];
    double lambda_2 = ((-1)*(K1[M]+Kel[M]+K2[M]) + (sqrt((K1[M]+Kel[M]+K2[M])*(K1[M]
    +Kel[M]+K2[M])-4*K2[M]*Kel[M])))/2;
    double lambda_3 = ((-1)*(K1[M]+Kel[M]+K2[M]) - (sqrt((K1[M]+Kel[M]+K2[M])*(K1[M]
    +Kel[M]+K2[M])-4*K2[M]*Kel[M])))/2;

    //Собственные векторы для лямбд |A-labmdaE| = 0
    std::vector<double> e1;
    std::vector<double> e2;
    std::vector<double> e3;

    std::vector<std::vector<double>> A1 = {
        {-Kabs[M]-lambda_1,         0,                              0},
        {Kabs[M],                   -(K1[M]+Kel[M])-lambda_1,    K2[M]},
        {0,                         K1[M],                       -K2[M]-lambda_1}
    };
    std::vector<std::vector<double>> A2 = {
        {-Kabs[M]-lambda_2,  0,                    0},
        {Kabs[M],    -(K1[M]+Kel[M])-lambda_2,    K2[M]},
        {0,          K1[M],              -K2[M]-lambda_2}
    };
    std::vector<std::vector<double>> A3 = {
        {-Kabs[M]-lambda_3,  0,                    0},
        {Kabs[M],    -(K1[M]+Kel[M])-lambda_3,    K2[M]},
        {0,          K1[M],              -K2[M]-lambda_3}
    };

    std::vector<double> b = {0,0,0};

    e1 = gauss(A1,b);
    e2 = gauss(A3,b);
    e3 = gauss(A3,b);

    // вычисление C из начальных условий RC = I для y
    std::vector<double> C;

    std::vector<std::vector<double>> R = {
        {exp(lambda_1*initial[0][0])*e1[0],  exp(lambda_2*initial[0][0])*e2[0],
        exp(lambda_3*initial[0][0])*e3[0]},
        {exp(lambda_1*initial[1][0])*e1[1],exp(lambda_2*initial[1][0])*e2[1],
        exp(lambda_3*initial[1][0])*e3[1]},
        {exp(lambda_1*initial[2][0])*e1[2],exp(lambda_2*initial[2][0])*e2[2],
        exp(lambda_3*initial[2][0])*e3[2]}
    };

    std::vector<double> I = {initial[0][1], initial[1][1],initial[2][1]};

    C = gauss(R,I);

    for (int i=0; i<3;i++)
        y[i] = C[0]*exp(lambda_1*t)*e1[i] + C[1]*exp(lambda_2*t)*e2[i] + C[2]*exp(lambda_3*t)*e3[i];

    // вычисление C из начальных условий RC = I для x
    double C4 = (initial[3][1] - (K13[M]-K11[M]*y[1])/(K12[M]) * exp(K12[M]*initial[3][0]));
    double C5 = (initial[4][1] - (K23[M]-K21[M]*y[1])/(K22[M]) * exp(K22[M]*initial[4][0]));
    double C6 = (initial[5][1] - (K33[M]-K31[M]*y[1])/(K32[M]) * exp(K32[M]*initial[5][0]));

    y[3] = C4/exp(K12[M]*t) + (K13[M]-K11[M]*y[1])/(K12[M]);
    y[4] = C5/exp(K22[M]*t) + (K23[M]-K21[M]*y[1])/(K22[M]);
    y[5] = C6/exp(K32[M]*t) + (K33[M]-K31[M]*y[1])/(K32[M]);

    return y;
}
```

4 Решение задачи Коши

В результате применения методов (2), (3), (4) и аналитического решения были получены решения задачи Коши для шага h = 0.1 и конечном шаге h_{max} для случаев без прогрессирования заболевания M^- и с прогрессированием M^+. Эти данные были записаны в файлах `RungeKutta.csv`, `Adams_Bashforth_Moulton.csv`, `Milne_Simpson.csv` и `Analytical_solution.csv` для метода Рунге-Кутты, Адамса-Башфорта-Моултона, Милна-Симпсона и эталонного решения соответственно. На основе полученных результатов были построены графики, приведенные на рисунках (5), 2, 3, 4.

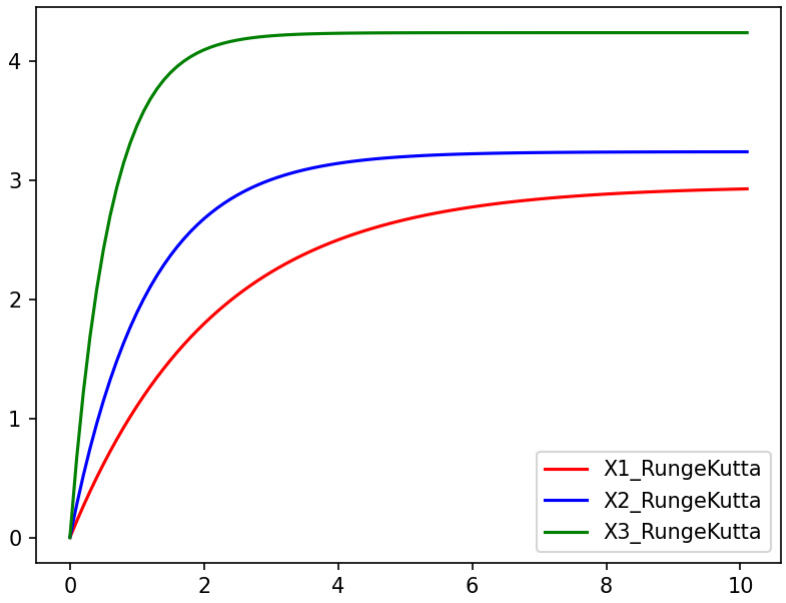

Рис. 1. Решения СОДУ, полученные методом Рунге-Кутты 4-го порядка (M^+)

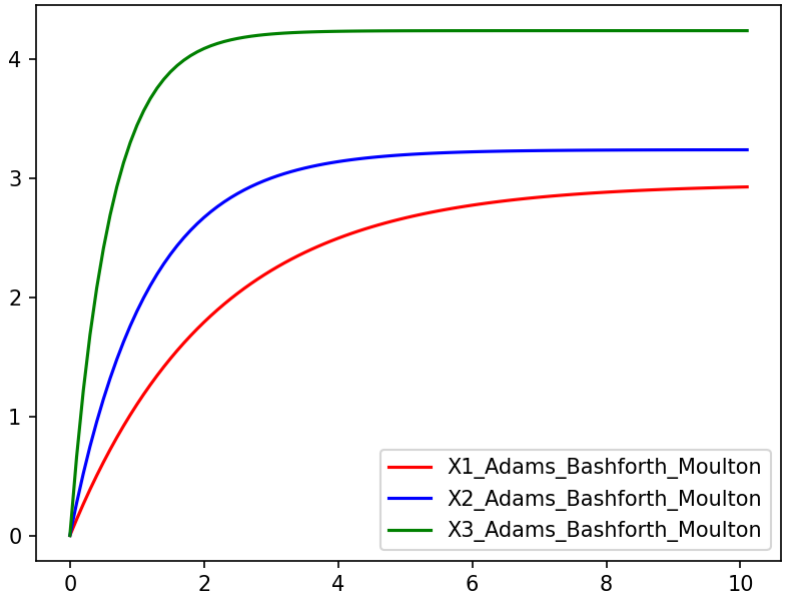

Рис. 2. Решения СОДУ, полученные методом Адамса-Башфорта-Моултона 4-го порядка (M^+)

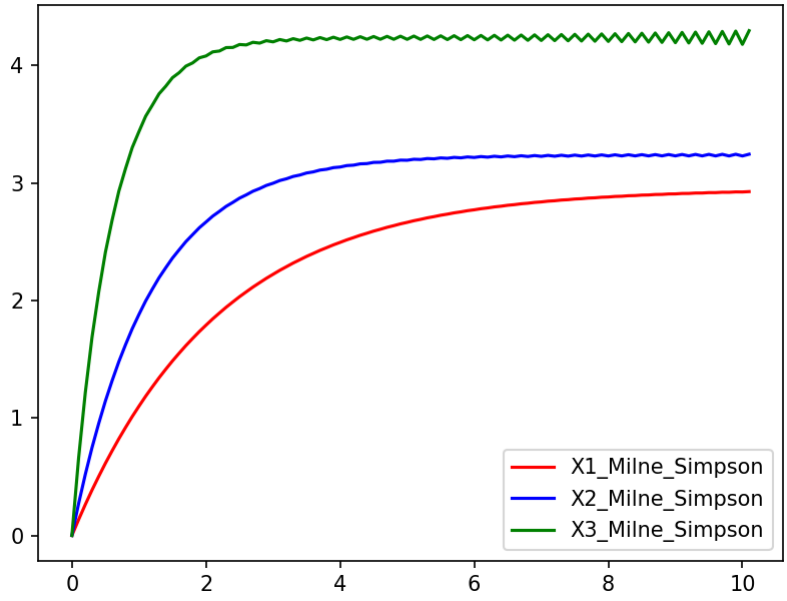

Рис. 3. Решения СОДУ, полученные методом Милна-Симпсона (M^+)

Осцилляции решения, полученного методом Милна–Симпсона при большом времени интегрирования связанно с численной неустойчивостью метода.

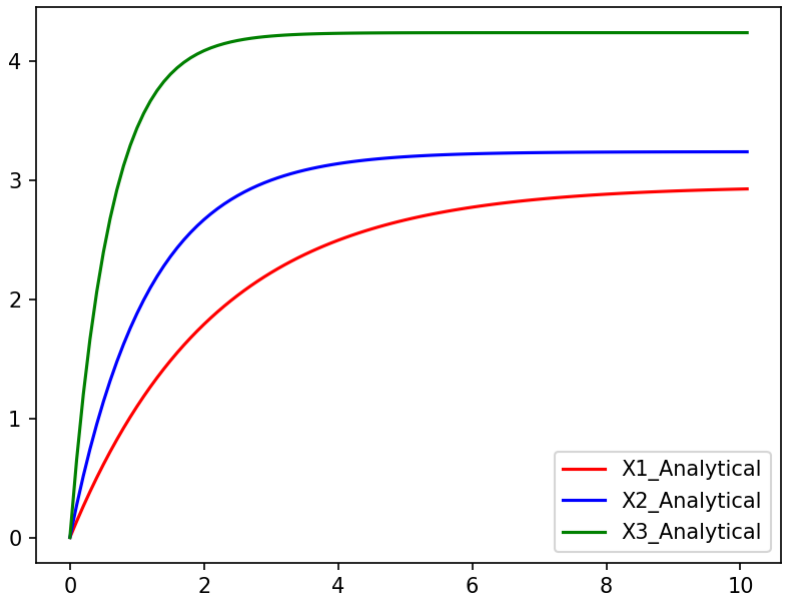

Рис. 4. Решения СОДУ, полученные аналитическим путем (M^+)

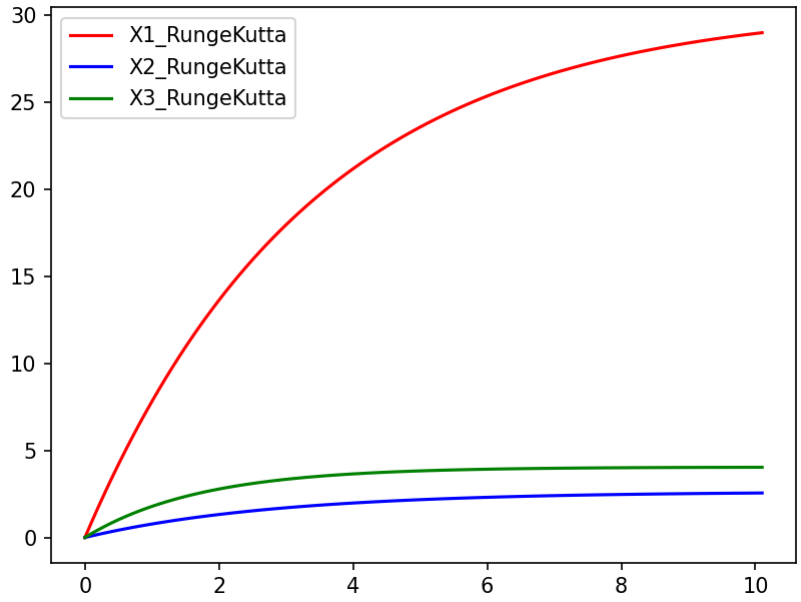

Рис. 5. Решения СОДУ, полученные методом Рунге-Кутты 4-го порядка (M^-)

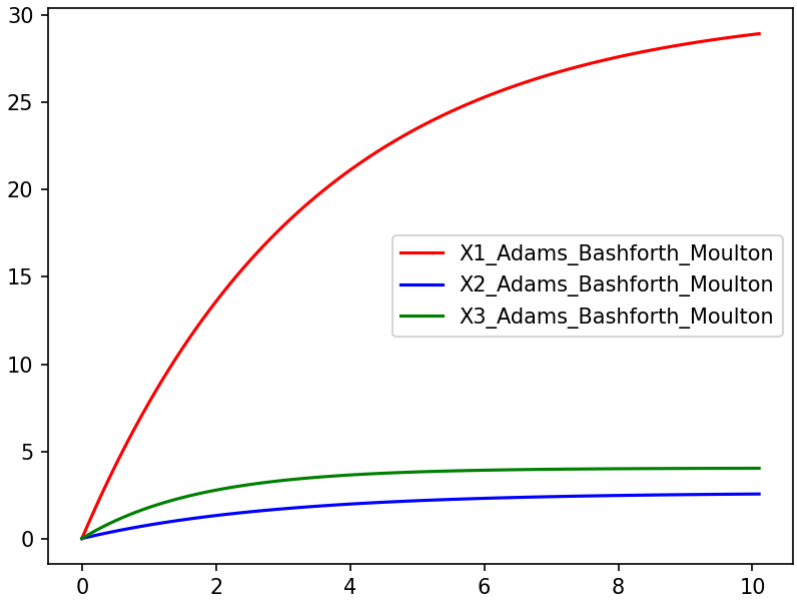

Рис. 6. Решения СОДУ, полученные методом Адамса-Башфорта-Моултона 4-го порядка (M^-)

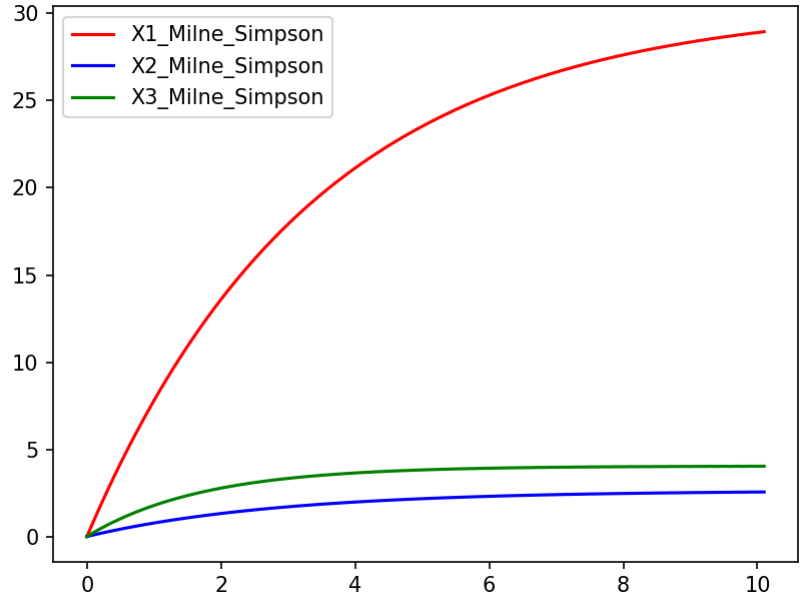

Рис. 7. Решения СОДУ, полученные методом Милна-Симпсона (M^-)

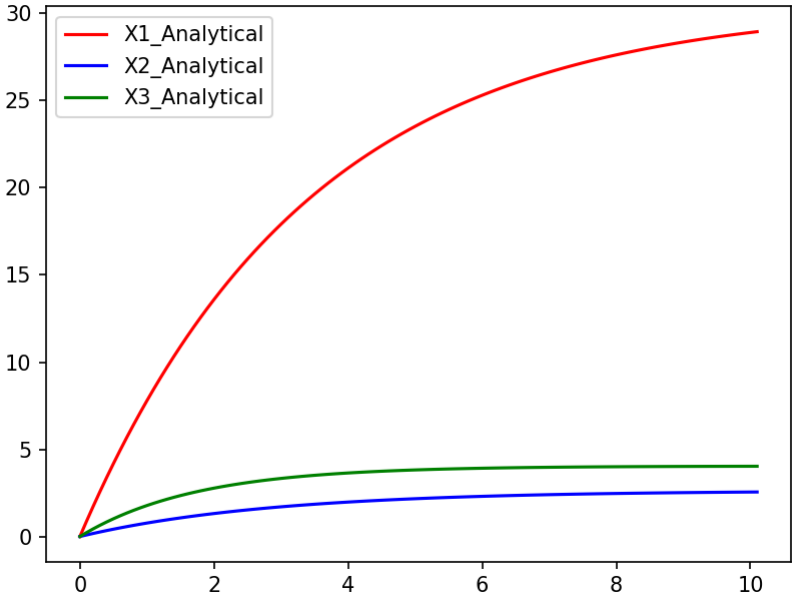

Рис. 8. Решения СОДУ, полученные аналитическим путем (M^{-})

5 Сравнение результатов

Сравнение результатов решения, полученных разными методами для случаев без прогрессирования заболевания M^- и с прогрессированием M^+ приведено на рисунках (9) и (10) соответственно.

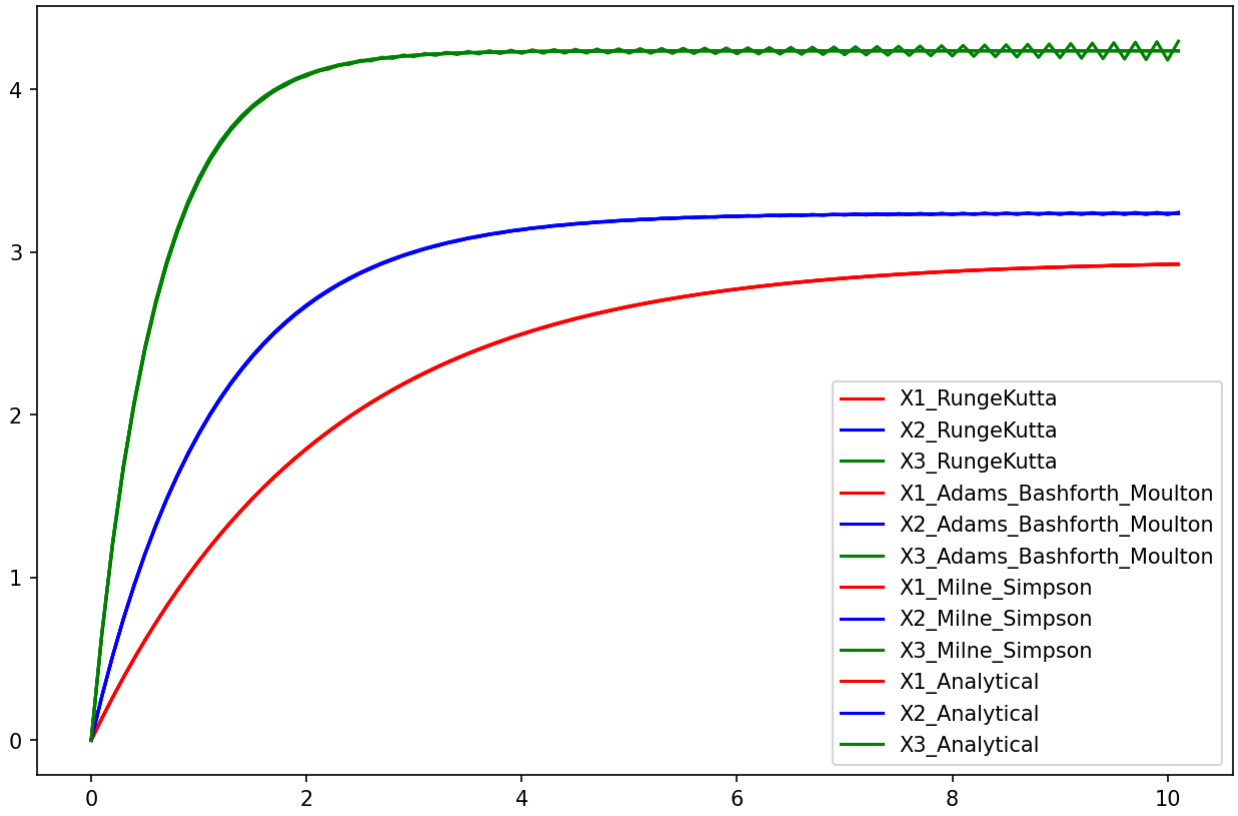

Рис. 9. Решения СОДУ для слуая без прогрессирования заболевания (M^-)

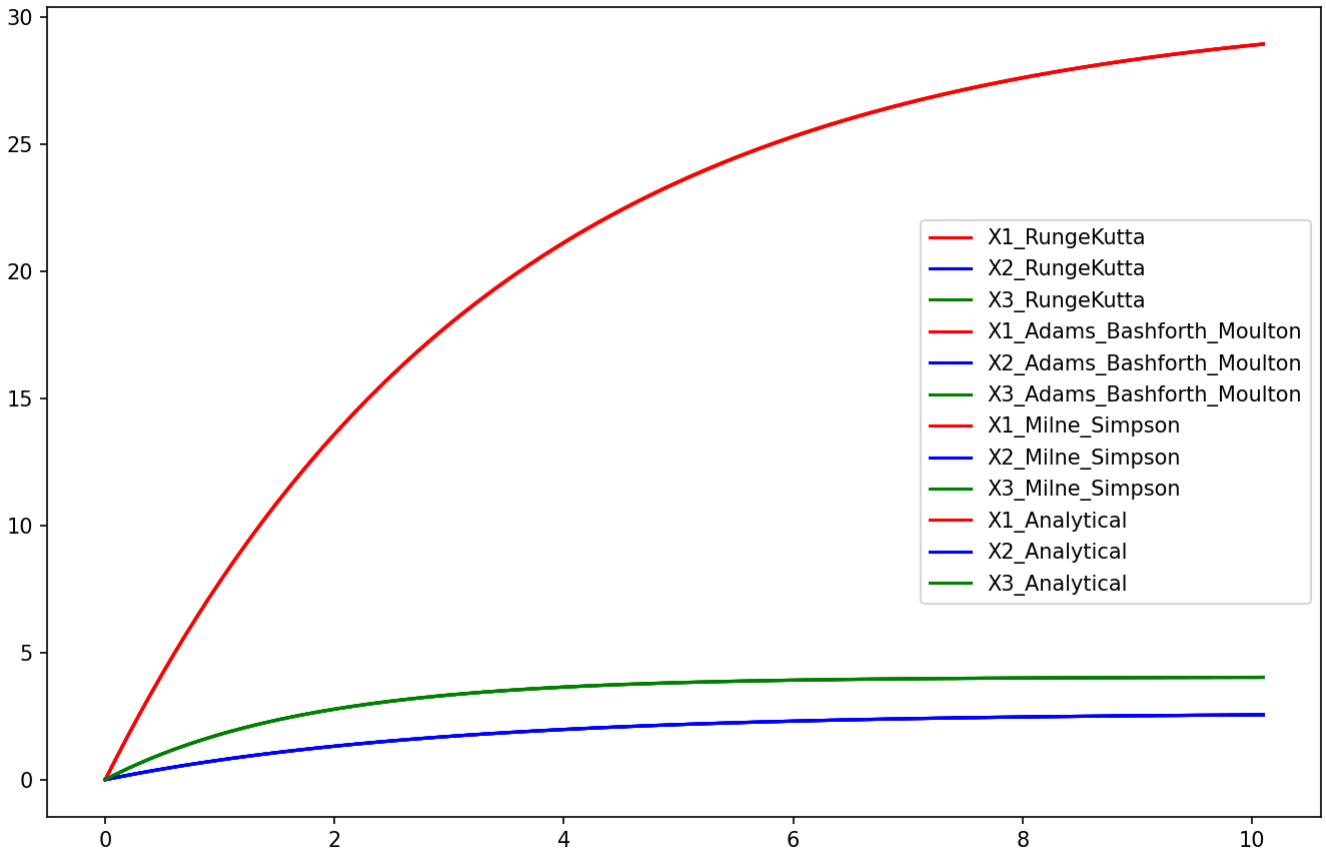

Рис. 10. Решения СОДУ, с прогрессированием заболевания (M^-)

6 Постановка обратной задачи и метод решения

В общем случае при решении задач восстановления параметров математических моделей на основе экспериментальных данных возникает некорректность малые погрешности в исходных данных могут приводить к большим отклонениям искомых параметров.

Согласно определению Ж. Адамара, задача считается корректной, если решение существует, единственно и устойчиво по отношению к малым возмущениям исходных данных.

Если хотя бы одно из этих условий нарушено, задача является некорректной. Такие задачи часто встречаются при идентификации коэффициентов в динамических системах, описываемых системами обыкновенных дифференциальных уравнений, когда параметры модели восстанавливаются на основе наблюдаемых траекторий состояний.

В классической форме некорректная задача записывается в виде операторного уравнения:

$$Au = f^S \tag{13}$$

где A - линейный или нелинейный оператор, f^S - правая часть, заданная с погрешностью.

Вариационный метод предлагает искать придлиженное решение в виде минимума функционала невязки:

$$J_0(v) = \|Av - f^S\| \tag{14}$$

Однако при наличии ошибок в f^S такое решение оказывается неустойчивым. Для стабилизации решения А. Н. Тихонов предложил добавлять сглаживающий член, отражающий априорную информацию о решении, что приводит к функционалу:

$$J_a(v) = \|Av - f^S\|^2 + \alpha\|v\|^2 \tag{15}$$

где $\alpha > 0$ - параметр регуляризации, контролирующий компромисс между точностью и устойчивостью.

Пусть K вектор неизвестных коэффициентов модели реакции пациента на лекарственный препарат. Тогда оператор $F(K) = x(t; K)$ определяет отображение от параметров модели к её решению. На практике значения $x_i(t)$ известны лишь в дискретные моменты времени t_j. Требуется определить такие коэффициенты K, при которых решения системы ОДУ наилучшим образом согласуется с наблюдаемыми данными $x_i^*(t_j)$. Таким, образом, обратная задача сводится к минимизации функционала невязки:

$$J_0(K) = \sum_{i=1}\sum_{j=1}\Big(x_i(t_j, K) - x_i^*(t_j)\Big)^2 \tag{16}$$

Однако задача идентификации коэффициентов является некорректной, поскольку малые ошибки в исходных данных могут вызывать значительные колебания найденных параметров. Для стабилизации решения применяется метод регуляризации Тихонова. С учётом априорной информации о допустимых диапазонах параметров и их типичных значениях K^* вводится регуляризованный функционал:

$$\min_{\mathbf{K}\in[\mathbf{K}_{\min},\mathbf{K}_{\max}]} J_0(K) = \sum_{i=1}\sum_{j=1}\Big(x_i(t_j, K) - x_i^*(t_j)\Big)^2 + \lambda\sum_{m=1}\left(\frac{K_m - K_m^*}{\Delta K_m}\right)^2 \tag{17}$$

где λ - параметр регуляризации, ΔK_m - априорно известная ширина диапозона изменения параметра K_m

Минимизация функционала, описанного выше, можно провести с помощью итерационного метода Левенберга–Марквардта, который сочетает в себе градиентный спуск и метод Гаусса–Ньютона. На каждой итерации вычисляется направление коррекции коэффициентов p_k, который вычисляется из системы:

$$(J^T(\Theta_k)J(\Theta_k + \lambda_k I)p_k = -J^T(\Theta_K)f(\Theta_K) \tag{18}$$

где $f(\Theta)$ - определяется как вектор невязки между моделируемыми и эталонными данными, $f(\Theta_K)$ - якобиан $f(\Theta)$ по Θ, $\lambda_k \geq 0$ - параметр, подбираемый для обеспечения монотонного уменьшения функции невязки, I - единичная матрица

Коэффициенты обновляются по правилу:

$$\Theta_{k+1} = \Theta_k + p_k \tag{19}$$

Итерации продолжаются до сходимости функции невязки, при этом регуляризация λ предотвращает чрезмерное отклонение коэффициентов от заданных диапазонов.

Заключение

1. В результате проделанной работы были реализованы три метода численного решения задачи Коши для системы ОДУ 1-го порядка. Каждый из этих методов имеет как своих недостатки, так и определенные достоинства.

2. Были приобретены навыки работы с динамическими библиотеками

1. Першин А.Ю. Лекции по курсу «Вычислительная математика». Москва, 2018-2021. С. 140.

2. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html

3. Cruz William, Martínez José, Raydan Marcos. Spectral Residual Method without Gradient Information for Solving Large-Scale Nonlinear Systems of Equations // Math. Comput. 2006. 07. Т. 75. С. 1429–1448.

Выходные данные

Юрченко А.А.. Отчет о выполнении лабораторной работы по дисциплине «Методы математического моделирования сложных процессов и систем». [Электронный ресурс] — Москва: 2025. — 24 с. URL: https://sa2systems.ru:88 (система контроля версий кафедры РК6)

Постановка: *д-р тех. наук, профессор кафедры САПР А.П. Соколов*

Решение и вёрстка: *студент группы РК6-11М, Юрченко А.А.*

2025, осенний семестр