МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу «ВВЕДЕНИЕ В ЧИСЛЕННЫЕ МЕТОДЫ»

ЗАДАНИЕ № 2

Решение задачи коши для дифференциального уравнения первого порядка или системы дифференциальных уравнений первого порядка

Подвариант 1: 1 - 3, 2 - 12

Подвариант 2: 9

ОТЧЕТ

о выполненном задании

студента 205 учебной группы факультета ВМК МГУ

Попова Алексея Павловича

Москва, 2018 г.

Подвариант 1

Постановка задачи

Возьмем в рассмотрение обыкновенное дифференциальное уравнение первого порядка, разрешенное относительно первой производной, и соотвественно имеющее вид (1), с дополнительным начальным условием заданным в точке $x = x_0$ и имеющим вид (2):

$$\frac{dy}{dx} = f(x,\ y),\ \ x_0 < x,\ (1)$$

$$y(x_0) = y_0, (2)$$

В постановке задачи учтем, что правая часть уравнения (1) является функцией, которая гарантирует существование и единственность решения задачи Коши (1), (2). А именно удовлетворяет условиям (3), (4), (5):

$$f(x, y) \in C[D], (3)$$

$$f_y(x, y) \in C[D], (4)$$

$$(x_0, y_0) \in D, (5)$$

Также необходимо рассмотреть случай, когда приходится решать не одно уравнение, а систему обыкновенных дифференциальных уравнений первого порядка, разрешенных относительно производных соответствующих неизвестных функций. Рассмотрим задачу Коши (6), (7), соответствующую данному типу задачи (рассматривай частный случай системы двух дифференциальных уравнений), она имеет вид (6), (7), где начальные условия заданы в точке $x = x_0$ и представляют собой (7):

$$\begin{cases} \frac{dy_1}{dx} = f_1(x,\ y_1,\ y_2), \\ \frac{dy_2}{dx} = f_2(x,\ y_1,\ y_2),\ \ x > x_0, (6) \end{cases}$$

$$y_1(x_0) = y_1^{(0)},\ \ y_2(x_0) = y_2^{(0)}\ , (7)$$

Обратим внимание, что в данной задаче гарантируется выполнения условий существования и единственности задачи Коши (6), (7) для системы обыкновенных дифференциальных уравнений разрешенных относительно производной соответствующих неизвестных функции, и эти условия имеют вид, аналогичный (3), (4), (5).

Стоит обратить внимание, что задачи данного класса очень широко распространены, и их численное решение очень важная задача. Многие прикладные задачи, сводятся к задач данного типа, например: задачи механики (уравнения движения материальной точки), небесной механике, химической кинетике, гидродинамике, аэродинамике, и во многих других сферах.

Цели и задачи практической работы

1. Решить задачу Коши (1), (2) (и (6), (7)), используя наиболее известные и широкоприменимые на практике методы Рунге-Кутта второго и четвертого порядка точности, аппроксимировав дифференциальную задачу соответствующей разностной схемой (на равномерной сетке), полученное конечно-разностное уравнение (в случае обыкновенного дифференциального уравнения) или уравнения (в случае системы обыкновенных дифференциальных уравнений), представляющее/ие собой некоторую рекуррентную формулу, которую необходимо пересчитать численно.

2. Найти численное решение задачи и построить его график.

3. Сравнить численное решение с точным решением дифференциального уравнения (системы дифференциальных уравнений), точное решени получить аналитически.

Алгоритм решения

Метод Рунге-Кутта второго порядка точности для одного уравнения

Данный метод реализуется следующими формулми:

$$K_1 = f(x_{n-1}, y_{n-1}),$$
$$K_2 = f(x_{n-1} + \frac{h}{2 \cdot \alpha}, y_{n-1} + \frac{h}{2 \cdot \alpha} \cdot K_1),$$
$$y_n = y_{n-1} + h \cdot (1 - \alpha) \cdot K_1 + h \cdot \alpha \cdot K_2,$$
$$x_n = x_{n-1} + h,$$

где: α - параметр алгоритма, h - величина шага сетки по x.

Метод Рунге-Кутта четвертого порядка точности для одного уравнения

Данный метод реализуется следующими формулми:

$$K_1 = f(x_{n-1}, y_{n-1}),$$
$$K_2 = f(x_{n-1} + \frac{h}{2}, y_{n-1} + \frac{h}{2} \cdot K_1),$$
$$K_3 = f(x_{n-1} + \frac{h}{2}, y_{n-1} + \frac{h}{2} \cdot K_2),$$
$$K_4 = f(x_{n-1} + h, y_{n-1} + h \cdot K_3),$$
$$y_n = y_{n-1} + \frac{h}{6} \cdot (K_1 + 2 \cdot K_2 + 2 \cdot K_3 + K_4),$$
$$x_n = x_{n-1} + h,$$

где: h - величина шага сетки по x.

Метод Рунге-Кутта второго порядка точности для системы уравнений

Данный метод реализуется следующими формулми:

$$K_1^1 = f(x_{n-1}, y_{n-1}^1, y_{n-1}^2),$$
$$K_1^2 = g(x_{n-1}, y_{n-1}^1, y_{n-1}^2),$$
$$K_2^1 = f(x_{n-1} + \frac{h}{2 \cdot \alpha}, y_{n-1}^1 + \frac{h}{2 \cdot \alpha} \cdot K_1^1, y_{n-1}^2 + \frac{h}{2 \cdot \alpha} \cdot K_1^2),$$
$$K_2^2 = g(x_{n-1} + \frac{h}{2 \cdot \alpha}, y_{n-1}^1 + \frac{h}{2 \cdot \alpha} \cdot K_1^1, y_{n-1}^2 + \frac{h}{2 \cdot \alpha} \cdot K_1^2),$$
$$y_n^1 = y_{n-1}^1 + h \cdot (1 - \alpha) \cdot K_1^1 + h \cdot \alpha \cdot K_2^1,$$
$$y_n^2 = y_{n-1}^2 + h \cdot (1 - \alpha) \cdot K_1^2 + h \cdot \alpha \cdot K_2^2,$$
$$x_n = x_{n-1} + h,$$

где: α - параметр алгоритма, h - величина шага сетки по x.

Метод Рунге-Кутта четвертого порядка точности для системы уравнений

Данный метод реализуется следующими формулми:

$$
\begin{aligned}
&K_1^1 = f(x_{n-1}, y_{n-1}^1, y_{n-1}^2),\\
&K_1^2 = g(x_{n-1}, y_{n-1}^1, y_{n-1}^2),\\
&K_2^1 = f(x_{n-1} + \frac{h}{2}, y_{n-1}^1 + \frac{h}{2} \cdot K_1^1, y_{n-1}^2 + \frac{h}{2} \cdot K_1^2),\\
&K_2^2 = g(x_{n-1} + \frac{h}{2}, y_{n-1}^1 + \frac{h}{2} \cdot K_1^1, y_{n-1}^2 + \frac{h}{2} \cdot K_1^2),\\
&K_3^1 = f(x_{n-1} + \frac{h}{2}, y_{n-1}^1 + \frac{h}{2} \cdot K_2^1, y_{n-1}^2 + \frac{h}{2} \cdot K_2^2),\\
&K_3^2 = g(x_{n-1} + \frac{h}{2}, y_{n-1}^1 + \frac{h}{2} \cdot K_2^1, y_{n-1}^2 + \frac{h}{2} \cdot K_2^2),\\
&K_4^1 = f(x_{n-1} + h, y_{n-1}^1 + h \cdot K_3^1, y_{n-1}^2 + h \cdot K_3^2),\\
&K_4^2 = g(x_{n-1} + h, y_{n-1}^1 + h \cdot K_3^1, y_{n-1}^2 + h \cdot K_3^2),\\
&y_n^1 = y_{n-1}^1 + \frac{h}{6} \cdot (K_1^1 + 2 \cdot K_2^1 + 2 \cdot K_3^1 + K_4^1),\\
&y_n^2 = y_{n-1}^2 + \frac{h}{6} \cdot (K_1^2 + 2 \cdot K_2^2 + 2 \cdot K_3^2 + K_4^2),\\
&x_n = x_{n-1} + h,
\end{aligned}
$$

где: h - величина шага сетки по x.

Описание программы

В программе реализован интерфейс работы с пользователем, посредствам которого можно осуществлять решение поставленных задач.

В аргументах командной строки задаются следующие параметры:

1. *mode* - устанавливается значение 1 (если пользователь решает уравнение), и значение 2 (если пользователь решает систему уравнений).

2. *input* - файл с параметрами решаемой задачи.

3. *output* - файл в который будет загружено решение поставленной задачи.

В случае, если пользователь ошибся при вводе параметров, ему будет предложена инструкция по использования данного программного обеспечения, которая будет выведена в консоль (Приложение 1).

Для тестирования программы реализованы функции вывода информации по работе программы с конкретной функцией, также в программе записан пакет функций, которые необходимо решать, и функционально описаны их точные решения.

Текст программы

Программа написана на языке программирования С, с использованием некоторых библиотечных функций, которые позволяют повысить точность производимых вычислений. При расчетах использовался тип *long double,* который также увеличивает точность.

Текст программы полностью представлен в Приложении 1.

Тестирование программы

При тестировании использовались специальные интерфейсы, реализованные в программе, которые позволяют сократить время тестирования и убедится в точности написанного алгоритма, и метода лежащего в основе алгоритма.

С проведенным тестированием можно ознакомиться в Приложении 2.

Выводы

В ходе практической работы был изучен метод Рунге-Кутта второго и четвертого порядка для решения задачи Коши для дифференциального уравнения первого порядка разрешенного относительно производной и системы дифференциальных уравнений первого порядка разрешенных относительно производной. Данные методы были реализованы в виде программы на языке программирования С. Они просты в реализации и обладают относительно высокой точностью. Также были найдены численные решения задачи Коши и построены графики решений некоторых задач, также произведен анализ полученных данных, и на их основе была произведена некоторая работа над ошибками, в ходе написания программы. В том числе во время тестирования был произведен анализ ошибок соответствующих методов на некоторых классах ОДУ и СОДУ (Приложение 2).

Если говорить о вычислительной сложности алгоритма, то легко заметить, что метод Рунге-Кутта 2-ого порядка точности для обыкновенного дифференциального уравнения разрешенного относительно первой производной требует вычисления функции $f(x_c, y_c)$ дважды, а метод Рунге-Кутта 4-ого порядка точности требует 4 раза вычислять эту функции (для систем из двух уравнений соответственно в 2 раза больше вычислений). Но необходимо заметить, что временные затраты на вычисление окупаются точностью, которую дает метод Рунге-Кутта 4-ого порядка точности. Данные о точности соответствующих методов представлены в Приложении 2.

Подвариант 2

Цель работы

Изучить схему работы метода прогонки, и его применимости к решению дифференциальных уравнений с краевыми условиями. Определить погрешности данного метода, на практике разобраться в тонкостях реализации данного метода на ЭВМ.

Постановка задачи

Рассматривается линейное дифференциальное уравнение второго порядка, которое имеет вид (1), с начальными условиями вида (2), которые заданы в начальных точках:

$$y'' + p(x)y' + q(x)y = -f(x), \qquad 0 < x < 1, (1)$$

$$\begin{cases} \sigma_1 y(0) + \gamma_1 y'(0) = \delta_1 \\ \sigma_2 y(1) + \gamma_2 y'(1) = \delta_2, (2) \end{cases}$$

Необходимо решить задачу численного решения уравнения (1), с начальными условиями (2).

Цели и задачи практической работы

1. Решить краевую задачу (1), (2) методом конечных разностей, посредствам аппроксимации ее разностной схемой второго порядка точности на равномерной сетке, полученную в результате применения данного метода систему, решить методом прогонки.

2. Найти разностное решение задачи и построить его график.

3. Сравнить численное решение с точным решением дифференциального уравнения, точное решени получить аналитически.

Алгоритмы решения

Перед нами стоит задача решения линейного дифференциального уравнения второго порядка с заданными краевыми условиями (3), (4):

$$y'' + p(x)y' + q(x)y = f(x), \qquad a < x < b, \ (3)$$

$$\begin{cases} \alpha_0 y(a) + \beta_1 y'(a) = A, \ \left|\alpha_0\right| + \left|\alpha_1\right| \neq 0; \\ \alpha_0 y(b) + \beta_1 y'(b) = B, \ \left|\beta_0\right| + \left|\beta_1\right| \neq 0, (4) \end{cases}$$

Из постановки задачи очевидно, что условия даны в краевых точках множества, на котором мы будет аппроксимировать решение дифференциального уравнения. Для этого построим сетку на этом множестве. Зададим некоторое число n - число узлов сетки. Тогда, имея число узлов сетки, получим значение шага сетки $h = \dfrac{(b-a)}{n}$, на основе этих данных разобьем отрезок $[a, b]$ на части с шагом h. Для этого зададим параметр обхода узлов сетки: $x_i = a + i \cdot h, 0 \leq i \leq n$. Для более удобного обозначения дальнейших действий, примем во внимание следующие равенства: $p_i = p(x_i)$, $q_i = q(x_i), f_i = f(x_i),$ где $x_i \in [a, b]$, а именно являются внутренними точками отрезка $[a, b]$. Известно, что можно заменить производные функции y ее разностными аналогами на равномерной сетке, а именно:

$$y'' = \frac{y_{i+1} - 2 \cdot y_i + y_{i-1}}{h^2},$$
$$y' = \frac{y_{i+1} - y_{i-1}}{2 \cdot h},$$

тогда, принимая во внимание новые обозначения, и производя несложные преобразования с (3) получаем:

$$A_i \cdot y_{i-1} + C_i \cdot y_i + B_i \cdot y_{i-1} = D_i.$$

В данных обозначениях:

$$A_i = \frac{1}{h^2} - \frac{p_i}{2 \cdot h},$$
$$C_i = \frac{2}{h^2} - q_i,$$
$$B_i = \frac{1}{h^2} + \frac{p_i}{2 \cdot h},$$
$$D_i = f_i,$$

заметим, что мы получили систему из $(n+1)$ уравнений с $(n+1)$ неизвестными.

Но есть проблема в интерпретации начальных условий (4).Это легко сделать с точностью $O(h)$, но это нас не устраивает. Поэтому опишем следующий вариант, который позволяет приблизить краевые условия с точностью $O(h^2)$.

Пусть, используя наши предыдущие обозначения: $h = \dfrac{(b-a)}{2}$ -шаг сетки. Тогда введем сетку: $x_i = a - \dfrac{h}{2} + i \cdot h, i = \overline{1...n}$, так, чтобы: $x_0 = a - \dfrac{h}{2}, x_{n+1} = b + \dfrac{h}{2}$. Такую сетку назовем сдвинутой (Рис.1).

Рис.1 Сдвинутая сетка

На ней будем аппроксимировать граничные условия (4):

$$\alpha_1 \cdot \frac{y_0 + y_1}{2} - \alpha_2 \cdot \frac{y_1 - y_0}{h} = \alpha,$$
$$\beta_1 \cdot \frac{y_{n+1} + y_n}{2} + \beta_2 \cdot \frac{y_{n+1} - y_n}{h} = \beta.$$

На этом этапе имеем все необходимые коэффициенты для осуществления метода прогонки, и можем утверждать, что граничные условия аппроксимированы нами с точностью $O(h^2)$.

Прямой ход метода прогонки представляет из себя поиск коэффициентов $\alpha_{i+1}, \beta_{i+1}$, для следующего соотношения: $y_i = y_{i+1} \cdot \alpha_{i+1} + \beta_{i+1}$, которые, при учете предыдущих обозначений можно записать в виде:

$$\alpha_{i+1} = \frac{-B_i}{A_i \cdot \alpha_i + C_i},$$
$$\beta_{i+1} = \frac{D_i - A_i \cdot \beta_i}{A_i \cdot \alpha_i + C_i}.$$

Учитывая, что граничные условия аппроксимированы на сетке, мы можем вычислить все эти коэффициенты. И тогда, при обратном ходе метода прогонки, мы просто получаем решения заданного уравнения в узлах стеки, из уравнения: $y_i = y_{i+1} \cdot \alpha_{i+1} + \beta_{i+1}$.

Описание программы

В программе реализован интерфейс работы с пользователем, посредствам которого можно осуществлять решение поставленной задачи.

В аргументах командной строки задаются следующие параметры:

1. *count_grid* - количество узлов сетки.
2. *mode* - параметр программы, который устанавливается в 1, если пользователь хочет получить решение поставленной задачи в количеством шагов *count_grid*, или в 2, если пользователь желает получить значение количества шагов, при котором ошибка минимальна, для данной задачи.

В случае, если пользователь ошибся при вводе параметров, ему будет предложена инструкция по использования данного программного обеспечения, которая будет выведена в консоль (Приложение 3).

Для тестирования программы реализованы функции вывода информации по работе программы с конкретной функцией, также в программе записан пакет функций, которые необходимо решать, и функционально описаны их точные решения (точность решений не идеально, они получены при помощи ресурса *https://www.wolframalpha.com/*).

Текст программы

Программа написана на языке программирования С, с использованием некоторых библиотечных функций, которые позволяют повысить точность производимых вычислений. При расчетах использовался тип *long double,* который также увеличивает точность.

Текст программы полностью представлен в Приложении 3.

Тестирование программы

При тестировании использовались специальные интерфейсы, реализованные в программе, которые позволяют сократить время тестирования и убедится в точности написанного алгоритма, и метода лежащего в основе алгоритма.

С проведенным тестированием можно ознакомиться в Приложении 4.

Выводы

В ходе практической работы была реализована программа на языке программирования С. В ней реализованы методы, которые позволяют решать линейное дифференциальное уравнение второго порядка с краевыми условиями. Метод прогонки реализующий решение данной задачи. По результатам работы данной программы, были построены графики решений дифференциальных уравнений, которые сравнены с решениями предоставленными сервисом *wolframalfa.* Также стоит отметить, что в результате работы над данным алгоритмом, были разобраны моменты аппроксимации граничных условий, что является очень важной задачей в классе предложенных условий.

Приложение 1

```
#include <stdio.h>

#include <stdlib.h>

#include <inttypes.h>

#include <math.h>

#include <tgmath.h>

#include <fcntl.h>

#include <unistd.h>


typedef struct Point {

    long double x;

    long double y;

    long double y_1;

} Point;


long double set_math_func( Point point , int type ); //input parametr and number of use function

long double set_math_func_sys( Point point , int type , int num ); //input parametr and number of use sys function

long double set_math_real_solution( Point point , int type ); //its test of real solution DD

long double set_math_real_solution_sys( Point point , int type , int num ); ////its test of real solution LODU

void	start_method_line( char *in_file , char *out_file ); //agregate all function to free main() line

void	start_method_sys( char *in_file , char *out_file ); //agregate all function to free main() sys

void	test_method_line( char *in_file , char *out_file ); //testing of my method line

void	test_method_sys( char *in_file , char *out_file ); //testing of my method sys

void	runge_kutt_second_accur( Point *point , int type , long double parm , long double grid ); //iteration of METHOD

void	runge_kutt_forth_accur( Point *point , int type , long double grid ); //iteration of METHOD
```

```
void        runge_kutt_second_accur_sys( Point *point , int type , long double parm , long double grid ); //iteration of METHOD

void        runge_kutt_forth_accur_sys( Point *point , int type , long double grid ); //iteration of METHOD


int

main( int argc , char **argv )

{

   if ( argc < 4 ) {

      printf( "!!!INPUT FORMAT!!!\n\n" );

      printf( "arg1 = TYPE OF YOU TASK\n\n");

      printf( "1 - is line DD\n\n" );

      printf( "2 - is sys DD\n\n" );

      printf( "arg2 = INPUT FILE WITH SET FORMAT\n\n");

      printf( "FOR LINE (arg1 = 1)\n/*\n// type\n// method_type\n// parm\n// grid\n// set point x\n// set point y\n// length\n*/\n\n" );

      printf( "FOR SYSTEM (arg1 = 2)\n/*\n// type\n// method_type\n// parm\n// grid\n// set point x\n// set point y\n// set point y_1\n// length\n*/\n\n" );

      printf( "arg3 = OUTPUT FILE WITH SET FORMAT\n\n");

      printf( "!!!INPUT FORMAT!!!\n\n" );

      return 0;

   }

   int select = (int) strtol( argv[1] , NULL , 10 );

   if ( select == 1 ) {

      test_method_line( argv[2] , argv[3] );

   }

   if ( select == 2 ) {

      start_method_sys( argv[2] , argv[3] );

   }

   return 0;

}
```

```
long double
set_math_func( Point point , int type )
{
   long double in_x = point.x; //represent types
   long double in_y = point.y; //represent types
   switch( type ) {
      case 1:
      {
         return ( -1 ) * in_y - in_x * in_x; //its only var function
      }
      case 2:
      {
         return 3 - in_y - in_x; //test 1
      }
      case 3:
      {
         return sinl( in_x ) - in_y; //test2
      }
      default:
      {
         break;
      }
   }
   return 0;
}

long double
set_math_real_solution( Point point , int type )
```

```
{
   long double in_x = point.x; //represent types
   switch( type ) {
      case 1:
      {
         return (-1) * in_x * in_x + 2 * in_x - 2 + 12 * expl( (-1) * in_x ); //its only var function
      }
      case 2:
      {
         return 4 - in_x - 4 * expl( (-1) * in_x ); //test 1
      }
      case 3:
      {
         return (1.0 / 2) * sinl( in_x ) - (1.0 / 2) * cosl( in_x ) + (21.0 / 2) * expl( (-1) * in_x ); //
test2
      }
      default:
      {
         break;
      }
   }
   return 0;
}

/*
// type
// method_type
// parm
// grid
```

```
// set point x
// set point y
// length
*/

void
start_method_line( char *in_file , char *out_file )
{
    int type , method_type , count_grid;
    long double parm , grid , SIZE;
    Point point;
    FILE *file = fopen( in_file , "r" ); //read data
    fscanf( file , "%d%d" , &type , &method_type ); //read data
    fscanf( file , "%Lf%d" , &parm , &count_grid ); //read data
    fscanf( file , "%Lf%Lf%Lf" , &(point.x) , &(point.y) , &SIZE ); //read data
    fclose( file ); //read data
    FILE *out = fopen( out_file , "w" ); //write data
    grid = SIZE / count_grid; //step of method
    long double ERROR = -1.0;
    if ( method_type == 1 ) {
        for ( int i = 0 ; i < count_grid ; i++ ) {
            long double y = set_math_real_solution( point , type ); //accuracy
            ERROR = fabsl( point.y - y ) > ERROR ? fabsl( point.y - y ) : ERROR; //accuracy
            fprintf( out , "X = %.10Lf , Y = %.10Lf\n" , point.x , point.y ); //print solution
            runge_kutt_second_accur( &point , type , parm , grid ); //METHOD dolve
            point.x += grid; //next iteration
        }
        fprintf( out , "SECOND_ACCURE METHOD MAX ERROR :: %.20Lf\n" , ERROR ); //
accuracy
```

```
    }
    if ( method_type == 2 ) {
        for ( int i = 0 ; i < count_grid ; i++ ) {
            long double y = set_math_real_solution( point , type ); //accuracy
            ERROR = fabsl( point.y - y ) > ERROR ? fabsl( point.y - y ) : ERROR; //accuracy
            fprintf( out , "X = %.10Lf , Y = %.10Lf\n" , point.x , point.y ); //print solution
            runge_kutt_forth_accur( &point , type , grid ); //METHOD dolve
            point.x += grid; //next iteration
        }
        fprintf( out , "FORTH_ACCURE METHOD MAX ERROR :: %.20Lf\n" , ERROR ); //
accuracy
    }
    fclose( out ); //write data
}

/*
// type
// set x
// set y
*/

void
test_method_line( char *in_file , char *out_file )
{
    FILE *file = fopen( in_file , "r" );
    int type;
    Point point;
    fscanf( file , "%d%Lf%Lf" , &type , &point.x , &point.y );
    fclose( file );
```

```
FILE *out = fopen( out_file , "w" );
int count_grid = 10000;
long double length = 10.0;
long double grid = length / count_grid;
Point copy;
copy.x = point.x;
copy.y = point.y;
fprintf( out , "TYPE = %d __ COUNT_GRID = %d __ LENGTH = %.5Lf __ GRID = %.
5Lf\n\n" ,
type , count_grid , length , grid );
for ( long double parm = 0.1 ; parm < 1.01 ; parm += 0.1 ) {
    fprintf( out , "ALFA = %.2Lf\n" , parm );
    long double ERROR = -1.0;
    for ( int i = 0 ; i < count_grid ; i++ ) {
        long double y = set_math_real_solution( point , type ); //accuracy
        ERROR = fabsl( point.y - y ) > ERROR ? fabsl( point.y - y ) : ERROR; //accuracy
        runge_kutt_second_accur( &point , type , parm , grid ); //solution
        point.x += grid; //next iteration
    }
    point.x = copy.x;
    point.y = copy.y;
    fprintf( out , "SECOND_ACCURE METHOD MAX ERROR :: %.30Lf\n" , ERROR ); //
accuracy
    fprintf( out ,
">_______________________________________________________________<\n" );
}
fprintf( out , "\n>+++++++++++++++++++++++++++++++++++++++++++++++++++++++
+<\n\n" );
fprintf( out , "TYPE = %d __ LENGTH = %.5Lf\n\n" , type , length );
for ( count_grid = 10 ; count_grid < 100000000 ; count_grid *= 10 ) {
    grid = length / count_grid;
```

```
        fprintf( out , "COUNT_GRID = %d __ GRID = %.10Lf\n" , count_grid , grid );
        long double ERROR = -1.0;
        for ( int i = 0 ; i < count_grid ; i++ ) {
            long double y = set_math_real_solution( point , type ); //accuracy
            ERROR = fabsl( point.y - y ) > ERROR ? fabsl( point.y - y ) : ERROR; //accuracy
            runge_kutt_forth_accur( &point , type , grid ); //solution
            point.x += grid; //next iteration
        }
        point.x = copy.x;
        point.y = copy.y;
        fprintf( out , "FORTH_ACCURE METHOD MAX ERROR :: %.30Lf\n" , ERROR ); //
accuracy
        fprintf( out ,
">_________________________________________________________________<\n" );
    }
    count_grid = 5;
    length = 2.0;
    long double parm = 0.5;
    grid = length / count_grid;
    fprintf( out , "\nSECOND_ACCURE METHOD\n" );
    fprintf( out , "TYPE = %d __ COUNT_GRID = %d\nLENGTH = %.5Lf __ GRID = %.5Lf __
PARM = %.5Lf\n\n" ,
    type , count_grid , length , grid , parm );
    for ( int i = 0 ; i < count_grid ; i++ ) {
        fprintf( out , "%.10Lf;%.10Lf;%.10Lf\n" , point.x , point.y , set_math_real_solution( point ,
type ) ); //print solve
        runge_kutt_second_accur( &point , type , parm , grid ); //solution
        point.x += grid; //next iteration
    }
    point.x = copy.x;
    point.y = copy.y;
```

```
    count_grid = 5;
    fprintf( out , "\nEND\n\n" );
    fprintf( out , "FORTH_ACCURE METHOD\n" );
    fprintf( out , "TYPE = %d __ COUNT_GRID = %d __ LENGTH = %.5Lf __ GRID = %.
5Lf\n\n" ,
    type , count_grid , length , grid );
    for ( int i = 0 ; i < count_grid ; i++ ) {
        fprintf( out , "%.10Lf;%.10Lf;%.10Lf\n" , point.x , point.y , set_math_real_solution( point ,
type ) ); //print solve
        runge_kutt_forth_accur( &point , type , grid ); //solution
        point.x += grid; //next iteration
    }
    fclose( out );
}

void
runge_kutt_second_accur( Point *point , int type , long double parm , long double grid )
{
    long double cur = 0 , sum = 0; //current memory
    Point cur_p = {}; //save point for use in functions
    cur = set_math_func( *point , type ); //real formula only
    sum += (1 - parm) * cur; //real formula only
    cur_p.x = point->x + grid / ( 2 * parm ); //real formula only
    cur_p.y = point->y + grid / ( 2 * parm ) * cur; //real formula only
    cur = set_math_func( cur_p , type ); //real formula only
    sum += parm * cur; //real formula only
    point->y += grid * sum; //real formula only
}

void
```

```
runge_kutt_forth_accur( Point *point , int type , long double grid )
{
    long double K1 , K2 , K3 , K4 , sum; //current memory
    Point cur_p = {}; //save point for use in other functions
    K1 = set_math_func( *point , type ); //real formula only
    cur_p.x = point->x + ( grid / 2 ); //real formula only
    cur_p.y = point->y + ( grid / 2 ) * K1; //real formula only
    K2 = set_math_func( cur_p , type ); //real formula only
    cur_p.y = point->y + ( grid / 2 ) * K2; //real formula only
    K3 = set_math_func( cur_p , type ); //real formula only
    cur_p.x = point->x + grid; //real formula only
    cur_p.y = point->y + grid * K3; //real formula only
    K4 = set_math_func( cur_p , type ); //real formula only
    sum = K1 + 2 * K2 + 2 * K3 + K4; //real formula only
    sum /= 6; //real formula only
    point->y += grid * sum; //real formula only
}

//______________________________________________________________________________________________________

long double
set_math_func_sys( Point point , int type , int num )
{
    long double in_x = point.x; //represent types
    long double in_y = point.y; //represent types
    long double in_y_1 = point.y_1; //represent types
    switch( type ) {
        case 1:
```

```
{
   if ( num == 1 ) {
      return (-2.0) * in_x * in_y * in_y + in_y_1 * in_y_1 - in_x - 1; //its only var function
   } else {
      return 1.0 / in_y_1 / in_y_1 - in_y - in_x / in_y; //its only var function
   }
}
case 2:
{
   if ( num == 1 ) {
      return ( in_y - in_y_1 ) / in_x; //test 1
   } else {
      return ( in_y + in_y_1 ) / in_x; //test 1
   }
}
case 3:
{
   if ( num == 1 ) {
      return in_y_1 - cosl( in_x ); //test2
   } else {
      return in_y + sinl( in_x ); //test2
   }
}
case 4:
{
   if ( num == 1 ) {
      return in_x + in_y; //test3
   } else {
      return in_x - in_y_1; //test3
```

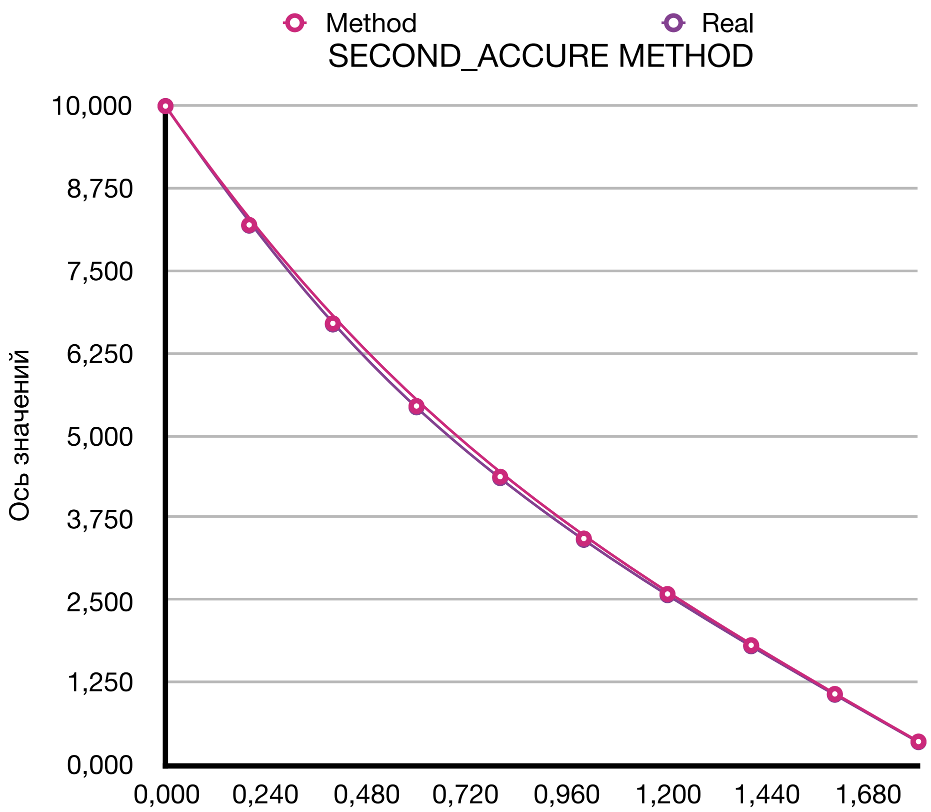
```
        }
      }
      default:
      {
        break;
      }
    }
    return 0;
}

long double
set_math_real_solution_sys( Point point , int type , int num )
{
    long double in_x = point.x; //represent types
    switch( type ) {
      case 1:
      {
        if ( num == 1 ) {
          return 0; //its only var function
        } else {
          return 0; //its only var function
        }
      }
      case 2:
      {
        if ( num == 1 ) {
          return in_x * ( cosl( logl( in_x ) ) - sinl( logl( in_x ) ) ); //test1
        } else {
          return in_x * ( cosl( logl( in_x ) ) + sinl( logl( in_x ) ) ); //test1
```

```
            }
        }
        case 3:
        {
            if ( num == 1 ) {
                return (-1) * sinl( in_x ); //test2
            } else {
                return 0; //test2
            }
        }
        case 4:
        {
            if ( num == 1 ) {
                return expl( in_x ) - 1 - in_x; //test2
            } else {
                return expl( (-1) * in_x ) - 1 + in_x; //test2
            }
        }
        default:
        {
            break;
        }
    }
    return 0;
}

/*
// type
// method_type
```

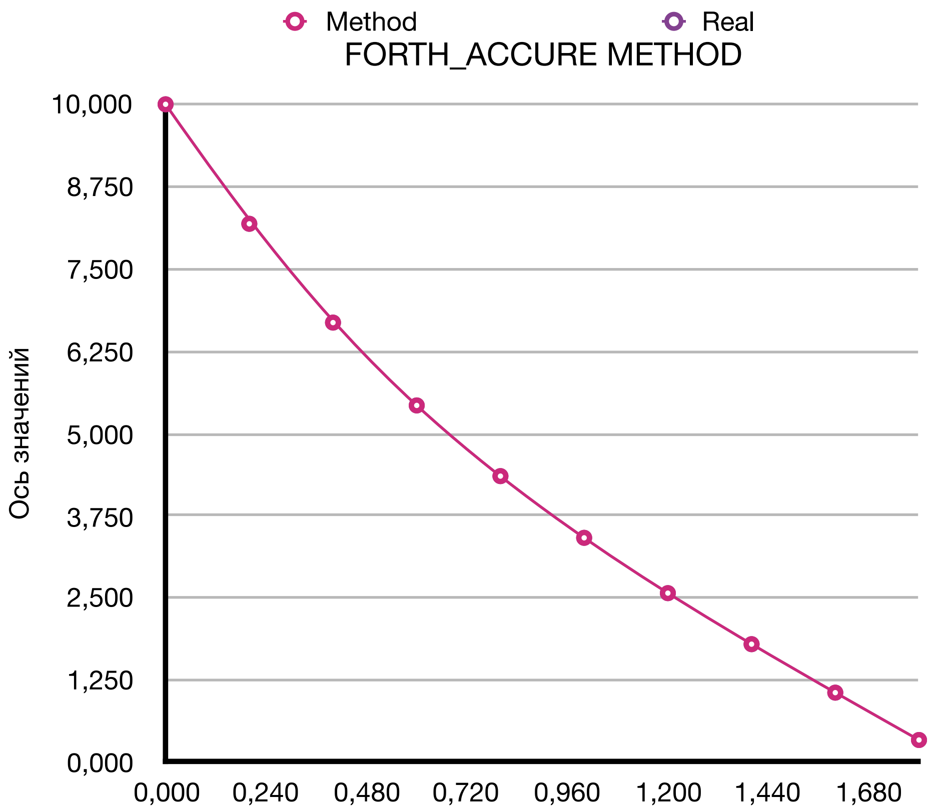
```
// parm
// grid
// set point x
// set point y
// set point y_1
// length
*/

void
start_method_sys( char *in_file , char *out_file )
{
    int type , method_type , count_grid; //cur memory
    long double parm , grid , SIZE; //cur memory
    Point point; //cur memory
    FILE *file = fopen( in_file , "r" ); //read data
    fscanf( file , "%d%d" , &type , &method_type ); //read data
    fscanf( file , "%Lf%d" , &parm , &count_grid ); //read data
    fscanf( file , "%Lf%Lf%Lf%Lf" , &(point.x) , &(point.y) , &(point.y_1) , &SIZE ); //read
data
    fclose( file ); //read data
    FILE *out = fopen( out_file , "w" ); //write data
    grid = SIZE / count_grid; //step of method
    if ( method_type == 1 ) {
        long double ERROR1 = -1.0 , ERROR2 = -1.0;
        for ( int i = 0 ; i < count_grid ; i++ ) {
            long double cur = fabsl( set_math_real_solution_sys( point , type , 1 ) - point.y ); //
accuracy
            ERROR1 = cur > ERROR1 ? cur : ERROR1; //accuracy
            cur = fabsl( set_math_real_solution_sys( point , type , 2 ) - point.y_1 ); //accuracy
            ERROR2 = cur > ERROR2 ? cur : ERROR2; //accuracy
```

```
        fprintf( out , "X = %.10Lf , Y1 = %.10Lf , Y2 = %.10Lf\n" , point.x , point.y ,
point.y_1 ); //print solution
        runge_kutt_second_accur_sys( &point , type , parm , grid ); //METHOD solve second
        point.x += grid; //next iteration
      }
      fprintf( out , "RUNGE-KUTT SECOND_ACCURE\n" );
      fprintf( out , "ERROR1 = %.10Lf __ ERROR2 = %.10Lf\n" , ERROR1 , ERROR2 );
   }
   if ( method_type == 2 ) {
      long double ERROR1 = -1.0 , ERROR2 = -1.0;
      for ( int i = 0 ; i < count_grid ; i++ ) { //accuracy
        long double cur = fabsl( set_math_real_solution_sys( point , type , 1 ) - point.y ); //
accuracy
        ERROR1 = cur > ERROR1 ? cur : ERROR1; //accuracy
        cur = fabsl( set_math_real_solution_sys( point , type , 2 ) - point.y_1 ); //accuracy
        ERROR2 = cur > ERROR2 ? cur : ERROR2; //accuracy
        fprintf( out , "X = %.10Lf , Y1 = %.10Lf , Y2 = %.10Lf\n" , point.x , point.y ,
point.y_1 ); //print solution
        runge_kutt_forth_accur_sys( &point , type , grid ); //METHOD solve forth function
        point.x += grid; //next iteration
      }
      fprintf( out , "RUNGE-KUTT FORTH_ACCURE\n" );
      fprintf( out , "ERROR1 = %.10Lf __ ERROR2 = %.10Lf\n" , ERROR1 , ERROR2 );
   }
   fclose( out ); //write data
}


/*
// type
// set x
```

```
// set y
// set y_1
*/

void
test_method_sys( char *in_file , char *out_file )
{
    FILE *file = fopen( in_file , "r" );
    int type; //cur memory
    Point point; //cur memory
    fscanf( file , "%d%Lf%Lf%Lf" , &type , &point.x , &point.y , &point.y_1 );
    fclose( file );
    FILE *out = fopen( out_file , "w" );
```




```
    int count_grid = 10000; //cur memory
    long double length = 1.0; //cur memory
    long double grid = length / count_grid; //cur memory
```

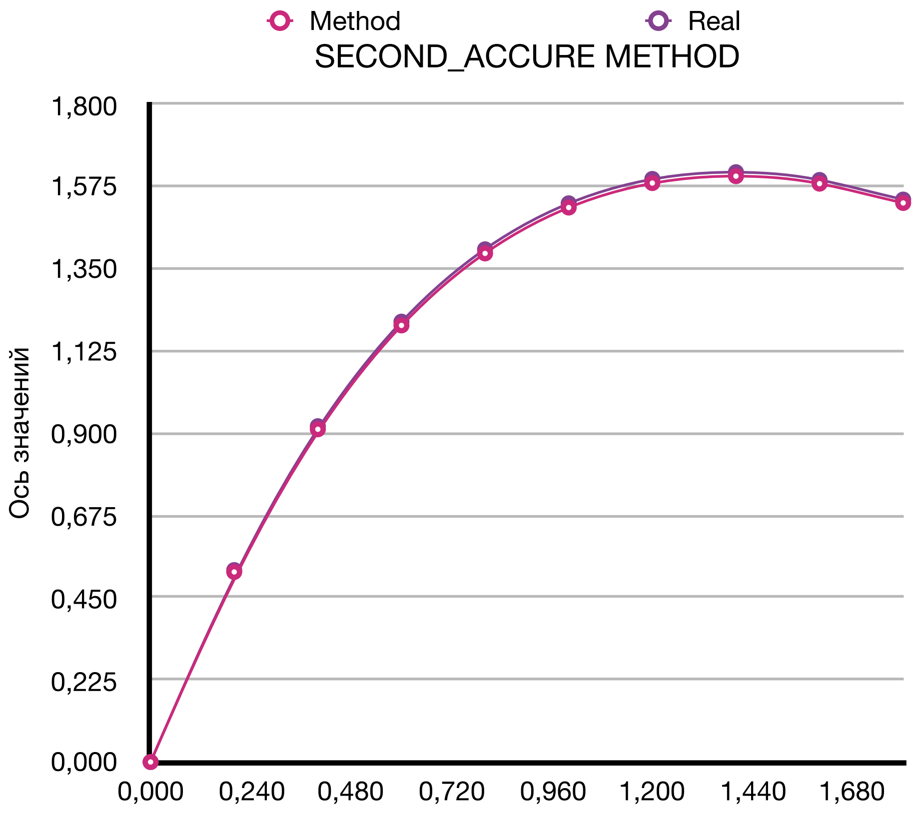



```
Point copy;
copy.x = point.x; //save pre sig
copy.y = point.y; //save pre sig
copy.y_1 = point.y_1; //save pre sig
fprintf( out , "TYPE = %d __ COUNT_GRID = %d\nLENGTH = %.5Lf __ GRID = %.
5Lf\n\n" ,
type , count_grid , length , grid );
for ( long double parm = 0.1 ; parm < 1.01 ; parm += 0.1 ) {
    fprintf( out , "ALFA = %.2Lf\n" , parm );
    long double ERROR1 = -1.0 , ERROR2 = -1.0;
    for ( int i = 0 ; i < count_grid ; i++ ) {
        long double cur = fabsl( set_math_real_solution_sys( point , type , 1 ) - point.y ); //
accuracy
        ERROR1 = cur > ERROR1 ? cur : ERROR1; //accuracy
        cur = fabsl( set_math_real_solution_sys( point , type , 2 ) - point.y_1 ); //accuracy
        ERROR2 = cur > ERROR2 ? cur : ERROR2; //accuracy
        runge_kutt_second_accur_sys( &point , type , parm , grid ); //METHOD solve second
```

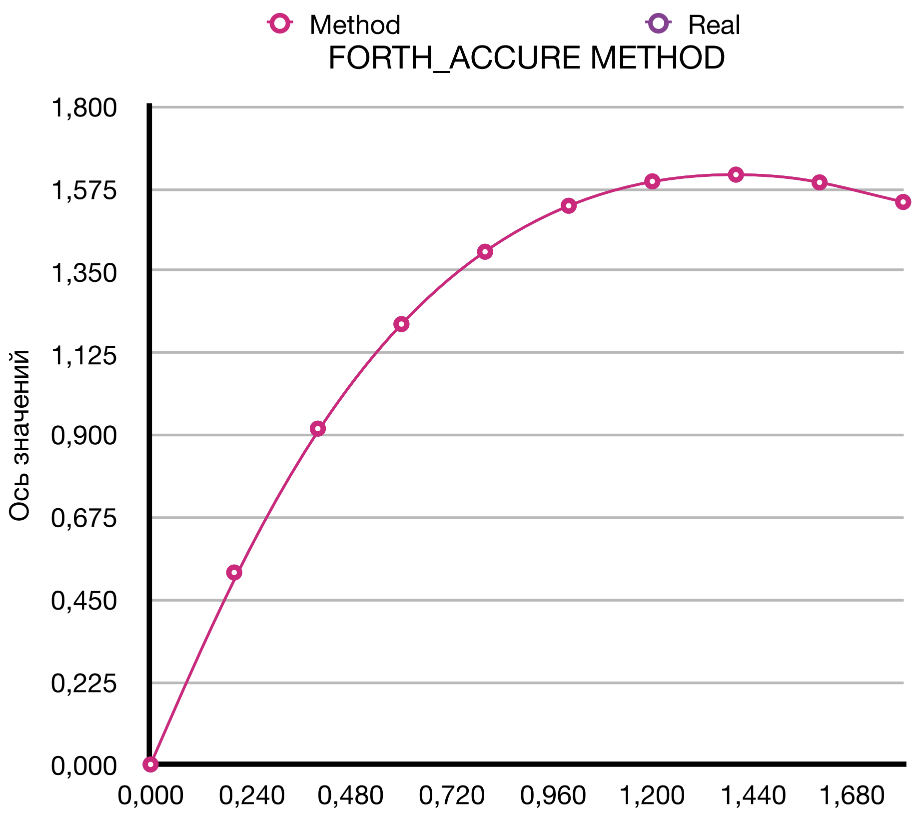
```
            point.x += grid; //next iteration
        }
        point.x = copy.x; //save pre sig
        point.y = copy.y; //save pre sig
        point.y_1 = copy.y_1; //save pre sig
        fprintf( out , "SECOND_ACCURE METHOD MAX ERROR\nY1 = %.30Lf;\nY2 = %.
30Lf;\n" , ERROR1 , ERROR2 );
        fprintf( out , ">________________________________________<\n" );
    }
    fprintf( out , "\n>++++++++++++++++++++++++++++++++++++++++<\n\n" );
    fprintf( out , "TYPE = %d __ LENGTH = %.5Lf\n\n" , type , length );
    for ( count_grid = 10 ; count_grid < 100000000 ; count_grid *= 10 ) {
        grid = length / count_grid;
        fprintf( out , "COUNT_GRID = %d __ GRID = %.7Lf\n" , count_grid , grid );
        long double ERROR1 = -1.0 , ERROR2 = -1.0;
        for ( int i = 0 ; i < count_grid ; i++ ) {
            long double cur = fabsl( set_math_real_solution_sys( point , type , 1 ) - point.y ); //
accuracy
            ERROR1 = cur > ERROR1 ? cur : ERROR1; //accuracy
            cur = fabsl( set_math_real_solution_sys( point , type , 2 ) - point.y_1 ); //accuracy
            ERROR2 = cur > ERROR2 ? cur : ERROR2; //accuracy
            runge_kutt_forth_accur_sys( &point , type , grid ); //METHOD solve forth function
            point.x += grid; //next iteration
        }
        point.x = copy.x; //save pre sig
        point.y = copy.y; //save pre sig
        point.y_1 = copy.y_1; //save pre sig
        fprintf( out , "FORTH_ACCURE METHOD MAX ERROR\nY1 = %.30Lf;\nY2 = %.30Lf;
\n" , ERROR1 , ERROR2 );
        fprintf( out , ">________________________________________<\n" );
```

```
    }
    count_grid = 5;
    length = 2.0;
    long double parm = 0.5;
    grid = length / count_grid;
    fprintf( out , "\nSECOND_ACCURE METHOD\n" );
    fprintf( out , "TYPE = %d __ COUNT_GRID = %d\nLENGTH = %.5Lf __ GRID = %.5Lf __ 
PARM = %.5Lf\n\n" ,
    type , count_grid , length , grid , parm );
    fprintf( out , "(x , u_m , v_m , u_r , v_r)\n" );
    for ( int i = 0 ; i < count_grid ; i++ ) {
        fprintf( out , "%.10Lf;%.10Lf;%.10Lf;%.10Lf;%.10Lf\n" , point.x , point.y , point.y_1 ,
        set_math_real_solution_sys( point , type , 1 ) , set_math_real_solution_sys( point , type , 2 )
); //print solve
        runge_kutt_second_accur_sys( &point , type , parm , grid ); //solution
        point.x += grid; //next iteration
    }
    point.x = copy.x; //save pre sig
    point.y = copy.y; //save pre sig
    point.y_1 = copy.y_1; //save pre sig
    count_grid = 5;
    fprintf( out , "\nEND\n\n" );
    fprintf( out , "FORTH_ACCURE METHOD\n" );
    fprintf( out , "TYPE = %d __ COUNT_GRID = %d __ LENGTH = %.5Lf __ GRID = %.
5Lf\n\n" ,
    type , count_grid , length , grid );
    fprintf( out , "(x , u_m , v_m , u_r , v_r)\n" );
    for ( int i = 0 ; i < count_grid ; i++ ) {
        fprintf( out , "%.10Lf;%.10Lf;%.10Lf;%.10Lf;%.10Lf\n" , point.x , point.y , point.y_1 ,
        set_math_real_solution_sys( point , type , 1 ) , set_math_real_solution_sys( point , type , 2 )
); //print solve
```

```
        runge_kutt_forth_accur_sys( &point , type , grid ); //solution
        point.x += grid; //next iteration
    }
    fclose( out );
}

void
runge_kutt_second_accur_sys( Point *point , int type , long double parm , long double grid )
{
    long double K1_1 , K2_1; //current memory
    long double K1_2 , K2_2; //current memory
    Point cur_p = {}; //save point for use in other functions
    K1_1 = grid * set_math_func_sys( *point , type , 1 );
    K1_2 = grid * set_math_func_sys( *point , type , 2 );
    cur_p.x = point->x + ( grid / ( 2 * parm ) ); //real formula only
    cur_p.y = point->y + ( K1_1 / ( 2 * parm ) ); //real formula only
    cur_p.y_1 = point->y_1 + ( K1_2 / ( 2 * parm ) ); //real formula only
    K2_1 = grid * set_math_func_sys( cur_p , type , 1 ); //real formula only
    K2_2 = grid * set_math_func_sys( cur_p , type , 2 ); //real formula only
    point->y += ( K1_1 * ( 1 - parm) + K2_1 * parm ); //real formula only
    point->y_1 += ( K1_2 * ( 1 - parm) + K2_2 * parm ); //real formula only
}

void
runge_kutt_forth_accur_sys( Point *point , int type , long double grid )
{
    long double K1_1 , K2_1 , K3_1 , K4_1; //current memory
    long double K1_2 , K2_2 , K3_2 , K4_2; //current memory
    Point cur_p = {}; //save point for use in other functions
```



K1_1 = grid * set_math_func_sys(*point , type , 1); //real formula only



K1_2 = grid * set_math_func_sys(*point , type , 2); //real formula only

cur_p.x = point->x + (grid / 2); //real formula only

```
    cur_p.y = point->y + ( K1_1 / 2 ); //real formula only
    cur_p.y_1 = point->y_1 + ( K1_2 / 2 ); //real formula only
    K2_1 = grid * set_math_func_sys( cur_p , type , 1 ); //real formula only
    K2_2 = grid * set_math_func_sys( cur_p , type , 2 ); //real formula only
    cur_p.y = point->y + ( K2_1 / 2 ); //real formula only
    cur_p.y_1 = point->y_1 + ( K2_2 / 2 ); //real formula only
    K3_1 = grid * set_math_func_sys( cur_p , type , 1 ); //real formula only
    K3_2 = grid * set_math_func_sys( cur_p , type , 2 ); //real formula only
    cur_p.x = point->x + grid; //real formula only
    cur_p.y = point->y + K3_1; //real formula only
    cur_p.y_1 = point->y_1 + K3_2; //real formula only
    K4_1 = grid * set_math_func_sys( cur_p , type , 1 ); //real formula only
    K4_2 = grid * set_math_func_sys( cur_p , type , 2 ); //real formula only
    point->y += ( K1_1 + 2 * K2_1 + 2 * K3_1 + K4_1 ) / 6.0; //real formula only
    point->y_1 += ( K1_2 + 2 * K2_2 + 2 * K3_2 + K4_2 ) / 6.0; //real formula only
}
```

Приложение 2

Тест 1

Уравнение:

$y' = -y - x^2,$

Начальные условия:

$(x_0, y_0) = (0,10),$

Точное решение:

$y = -x^2 + 2 \cdot x - 2 + 12 \cdot e^{-x},$

Вывод программы:

TYPE = 1 __ COUNT_GRID = 10000 __ LENGTH = 10.00000 __ GRID = 0.00100

ALFA = 0.10

SECOND_ACCURE METHOD MAX ERROR :: 0.0000025002274557858750192451 68

>____<

ALFA = 0.20

SECOND_ACCURE METHOD MAX ERROR :: 0.00000124965897823486882245 4473

>____<

ALFA = 0.30

SECOND_ACCURE METHOD MAX ERROR :: 0.000000832802819164535357288059

>____<

ALFA = 0.40

SECOND_ACCURE METHOD MAX ERROR :: 0.000000624374739660593647272435

>____<

ALFA = 0.50

SECOND_ACCURE METHOD MAX ERROR :: 0.000000499317891880513009539300

>____<

ALFA = 0.60

SECOND_ACCURE METHOD MAX ERROR :: 0.000000489935667276949343706960

>____<

ALFA = 0.70

SECOND_ACCURE METHOD MAX ERROR :: 0.000000522904979023339211430255

>____<

ALFA = 0.80

SECOND_ACCURE METHOD MAX ERROR :: 0.000000548144796862388983260672

>____<

ALFA = 0.90

SECOND_ACCURE METHOD MAX ERROR :: 0.000000568072583634251365714007

>____<

ALFA = 1.00

SECOND_ACCURE METHOD MAX ERROR :: 0.000000584198529416301778893761

>____<

>+++<

TYPE = 1 __ LENGTH = 10.00000

COUNT_GRID = 10 __ GRID = 1.0000000000

FORTH_ACCURE METHOD MAX ERROR :: 0.064613372609358807416995995609

>____<

COUNT_GRID = 100 __ GRID = 0.1000000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000002701857395404592965260093

>____<

COUNT_GRID = 1000 __ GRID = 0.0100000000

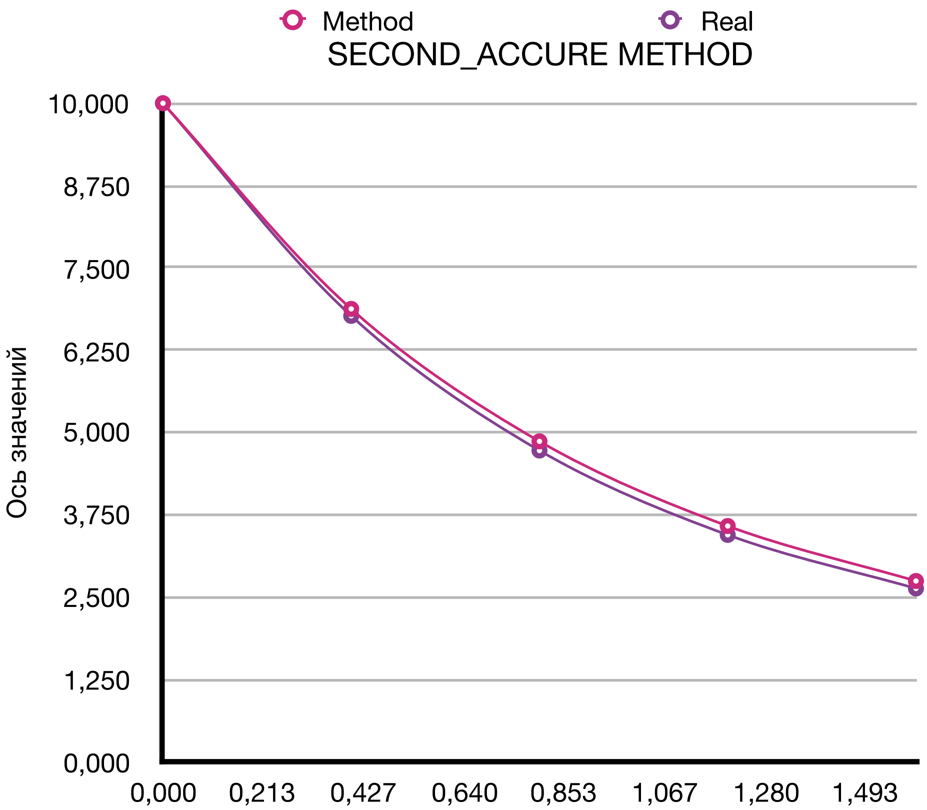
FORTH_ACCURE METHOD MAX ERROR :: 0.000000000247185847311592876707

>____<

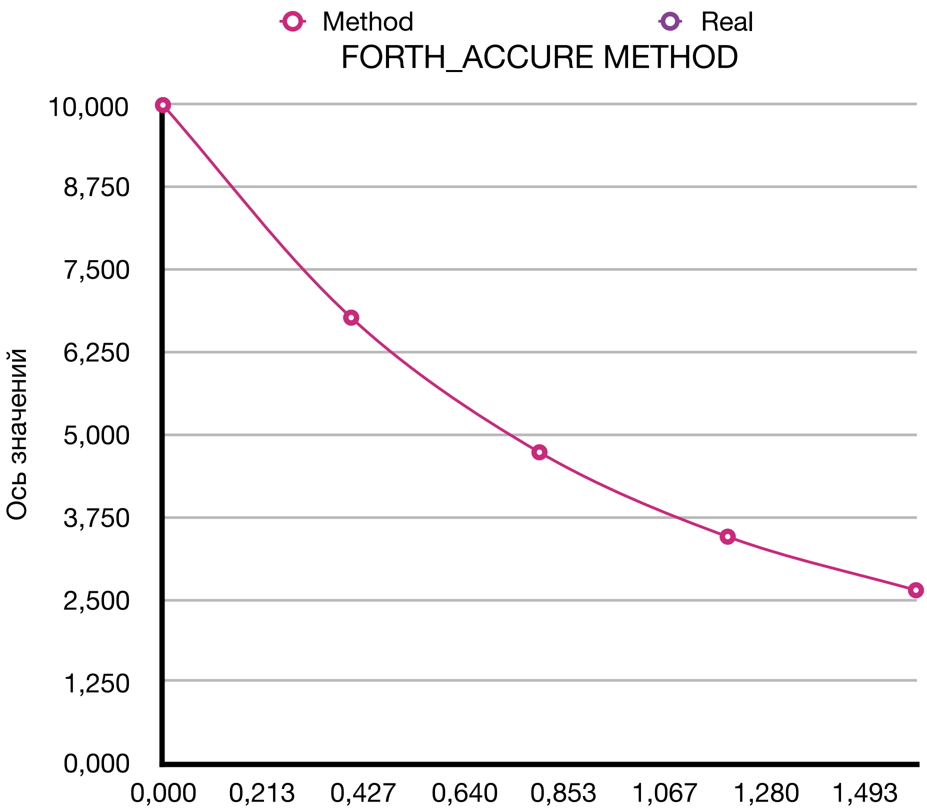
COUNT_GRID = 10000 __ GRID = 0.0010000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000024518147928587197

>____<



COUNT_GRID = 100000 __ GRID = 0.0001000000



FORTH_ACCURE METHOD MAX ERROR :: 0.0000000000000352426421379448 13

>____<

COUNT_GRID = 1000000 __ GRID = 0.0000100000

FORTH_ACCURE METHOD MAX ERROR :: 0.00000000000581285020118116336

>____<

COUNT_GRID = 10000000 __ GRID = 0.0000010000

FORTH_ACCURE METHOD MAX ERROR :: 0.00000000000212261458409912506 8

>____<

SECOND_ACCURE METHOD

TYPE = 1 __ COUNT_GRID = 10

LENGTH = 2.00000 __ GRID = 0.20000 __ PARM = 0.50000

(x , y , r)

0.0000000000;10.0000000000;10.0000000000

0.2000000000;8.1960000000;8.1847690369

0.4000000000;6.7015200000;6.6838405524

0.6000000000;5.4464464000;5.4257396331

0.8000000000;4.3732860480;4.3519475694

1.0000000000;3.4348945594;3.4145532941

1.2000000000;2.5926135387;2.5743305429

1.4000000000;1.8147431017;1.7991635673

1.6000000000;1.0752893434;1.0627582159

1.8000000000;0.3529372616;0.3435866587

END

FORTH_ACCURE METHOD

TYPE = 1 __ COUNT_GRID = 10 __ LENGTH = 2.00000 __ GRID = 0.20000

(x , y , r)

0.0000000000;10.0000000000;10.0000000000

0.2000000000;8.1847933333;8.1847690369

0.4000000000;6.6838791284;6.6838405524

0.6000000000;5.4257853051;5.4257396331

0.8000000000;4.3519952888;4.3519475694

1.0000000000;3.4145996094;3.4145532941

1.2000000000;2.5743731869;2.5743305429

1.4000000000;1.7992011406;1.7991635673

1.6000000000;1.0627899471;1.0627582159

1.8000000000;0.3436122227;0.3435866587

По графику и по таблице данных видим, что метод Рунге-Кутта 4-го порядка оказался в данном случае точнее, чем метод Рунге-Кутта 2-го порядка.

Тест 2

Уравнение:

$y' = -y - x,$

Начальные условия:

$(x_0, y_0) = (0,0),$

Точное решение:

$y = 4 - x - 4 \cdot e^{-x},$

Вывод программы:

TYPE = 2 __ COUNT_GRID = 10000 __ LENGTH = 10.00000 __ GRID = 0.00100

ALFA = 0.10

SECOND_ACCURE METHOD MAX ERROR :: 0.000000245436994550526053904260

>____<

ALFA = 0.20

SECOND_ACCURE METHOD MAX ERROR :: 0.000000245436994550526053904260

>____<

ALFA = 0.30

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 0.40

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 0.50

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 0.60

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 0.70

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 0.80

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 0.90

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

ALFA = 1.00

SECOND_ACCURE METHOD MAX ERROR :: 0.00000024543699455052605390426

>____<

>+++<

TYPE = 2 __ LENGTH = 10.00000

COUNT_GRID = 10 __ GRID = 1.0000000000

FORTH_ACCURE METHOD MAX ERROR :: 0.028482235314230713559349728037

>____<

COUNT_GRID = 100 __ GRID = 0.1000000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000001332964224447404089080038

>____<

COUNT_GRID = 1000 __ GRID = 0.0100000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000123652759472653306361

>____<

COUNT_GRID = 10000 __ GRID = 0.0010000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000012277071720356858

>____<

COUNT_GRID = 100000 __ GRID = 0.0001000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000002049575786866598

>____<

COUNT_GRID = 1000000 __ GRID = 0.0000100000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000034864472420181869

>____<

COUNT_GRID = 10000000 __ GRID = 0.0000010000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000132168147953803938

>____<

SECOND_ACCURE METHOD

TYPE = 2 __ COUNT_GRID = 10

LENGTH = 2.00000 __ GRID = 0.20000 __ PARM = 0.50000

(x , y , r)

0.0000000000;0.0000000000;0.0000000000

0.2000000000;0.5200000000;0.5250769877

0.4000000000;0.9104000000;0.9187198159

0.6000000000;1.1945280000;1.2047534556

0.8000000000;1.3915129600;1.4026841435

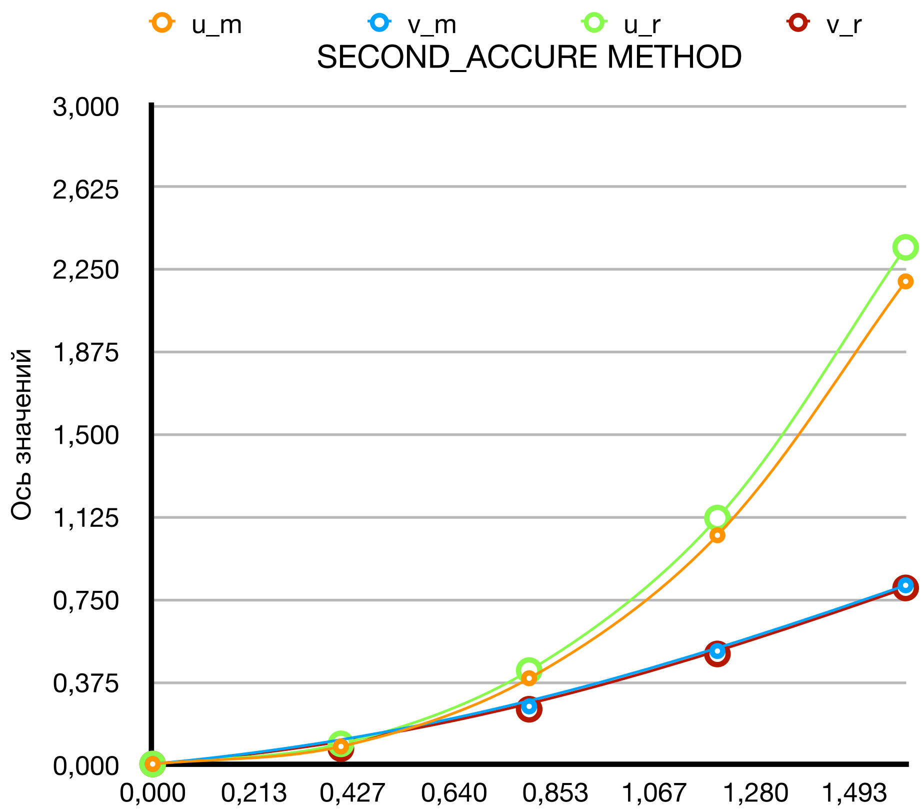
1.0000000000;1.5170406272;1.5284822353

1.2000000000;1.5839733143;1.5952231524

1.4000000000;1.6028581177;1.6136121442

1.6000000000;1.5823436565;1.5924139280

1.8000000000;1.5295217984;1.5388044471



END

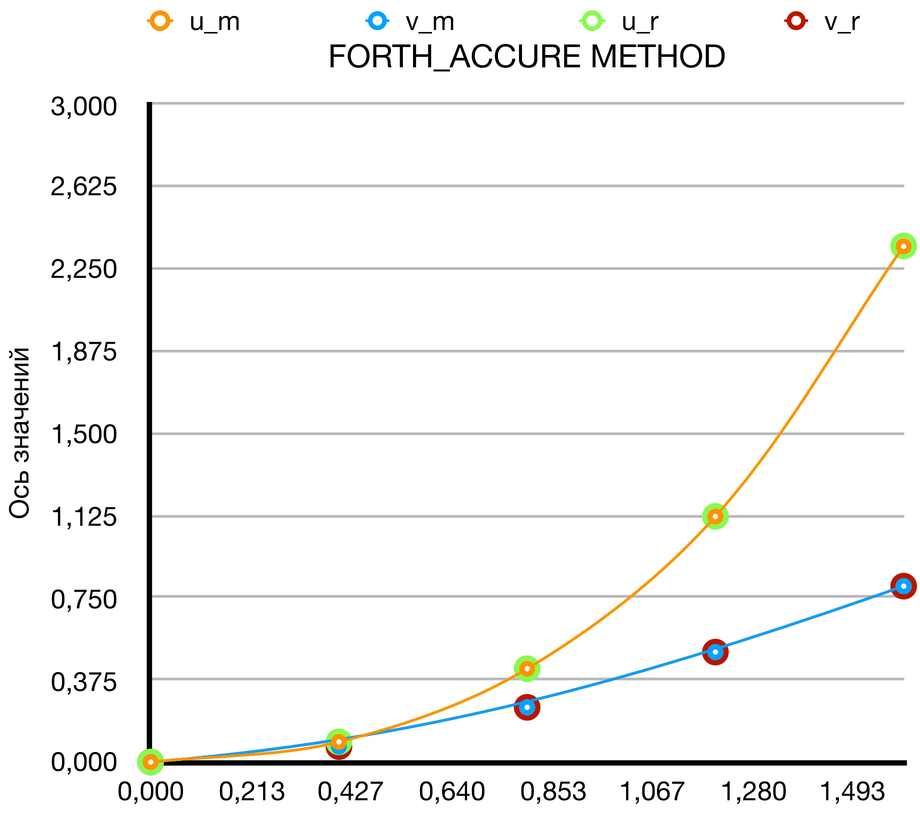
FORTH_ACCURE METHOD

TYPE = 2 __ COUNT_GRID = 10 __ LENGTH = 2.00000 __ GRID = 0.20000

(x , y , r)

0.0000000000;0.0000000000;0.0000000000

0.2000000000;0.5250666667;0.5250769877



0.4000000000;0.9187029156;0.9187198159

0.6000000000;1.2047327004;1.2047534556

0.8000000000;1.4026614862;1.4026841435

1.0000000000;1.5284590475;1.5284822353

1.2000000000;1.5952003708;1.5952231524

1.4000000000;1.6135903836;1.6136121442

1.6000000000;1.5923935667;1.5924139280

1.8000000000;1.5387856929;1.5388044471

По графику и по таблице данных видим, что метод Рунге-Кутта 4-го порядка оказался в данном случае точнее, чем метод Рунге-Кутта 2-го порядка.

Тест 3

Уравнение:

$y' = sin(x) - y,$

Начальные условия:

$(x_0, y_0) = (0{,}10),$

Точное решение:

$$y = -\frac{1}{2} \cdot cos(x) + \frac{1}{2} \cdot sin(x) + \frac{21}{2} \cdot e^{-x},$$

Вывод программы:

TYPE = 3 __ COUNT_GRID = 10000 __ LENGTH = 10.00000 __ GRID = 0.00100

ALFA = 0.10

SECOND_ACCURE METHOD MAX ERROR :: 0.0000008627274728009314171128880

>____<

ALFA = 0.20

SECOND_ACCURE METHOD MAX ERROR :: 0.0000005454093857234706965719 79

>____<

ALFA = 0.30

SECOND_ACCURE METHOD MAX ERROR :: 0.0000005914600787358323397135 96

>____<

ALFA = 0.40

SECOND_ACCURE METHOD MAX ERROR :: 0.0000006179078562403364505151 16

>____<

ALFA = 0.50

SECOND_ACCURE METHOD MAX ERROR :: 0.0000006350844593055668396708 31

>____<

ALFA = 0.60

SECOND_ACCURE METHOD MAX ERROR :: 0.0000006471393578758206921452 97

>____<

ALFA = 0.70

SECOND_ACCURE METHOD MAX ERROR :: 0.0000006560661341562315507225 38

>____<

ALFA = 0.80

SECOND_ACCURE METHOD MAX ERROR :: 0.0000006629424010406829992358 98

>____<

ALFA = 0.90

SECOND_ACCURE METHOD MAX ERROR :: 0.000000668401864414251006829559

>____<

ALFA = 1.00

SECOND_ACCURE METHOD MAX ERROR :: 0.000000672841290423348975524576

>____<

>++<

TYPE = 3 __ LENGTH = 10.00000

COUNT_GRID = 10 __ GRID = 1.0000000000

FORTH_ACCURE METHOD MAX ERROR :: 0.076687333449711229949041779719

>____<

COUNT_GRID = 100 __ GRID = 0.1000000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000003464641069917968191349544

>____<

COUNT_GRID = 1000 __ GRID = 0.0100000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000320182893653181421811

>____<

COUNT_GRID = 10000 __ GRID = 0.0010000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000031789675059012978

>____<

COUNT_GRID = 100000 __ GRID = 0.0001000000

FORTH_ACCURE METHOD MAX ERROR :: 0.000000000000001006397114082513

>____<

COUNT_GRID = 1000000 __ GRID = 0.0000100000

FORTH_ACCURE METHOD MAX ERROR :: 0.00000000000014016159110077919

>____<

COUNT_GRID = 10000000 __ GRID = 0.0000010000

FORTH_ACCURE METHOD MAX ERROR :: 0.00000000000066701310273697967

>____<

SECOND_ACCURE METHOD

TYPE = 3 __ COUNT_GRID = 5

LENGTH = 2.00000 __ GRID = 0.40000 __ PARM = 0.50000

(x , y , r)

0.0000000000;10.0000000000;10.0000000000

0.4000000000;6.8778836685;6.7725391575

0.8000000000;4.8671623138;4.7282788140

1.2000000000;3.5821609215;3.4473798908

1.6000000000;2.7476288375;2.6343000016

END

FORTH_ACCURE METHOD

TYPE = 3 __ COUNT_GRID = 5 __ LENGTH = 2.00000 __ GRID = 0.40000

(x , y , r)

0.0000000000;10.0000000000;10.0000000000

0.4000000000;6.7734035832;6.7725391575

0.8000000000;4.7294202410;4.7282788140

1.2000000000;3.4484958869;3.4473798908

1.6000000000;2.6352531540;2.6343000016

По графику и по таблице данных видим, что метод Рунге-Кутта 4-го порядка оказался в данном случае точнее, чем метод Рунге-Кутта 2-го порядка.

Тест 4

Система:

$u' = x + u,$

$v' = x - v,$

Начальные условия:

$(x_0, u_0, v_0) = (0{,}0{,}0),$

Точное решение:

$u = e^x - x - 1,$

$v = e^{-x} + x - 1,$

Вывод программы:

TYPE = 4 __ COUNT_GRID = 10000

LENGTH = 1.00000 __ GRID = 0.00010

ALFA = 0.10

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;

>____<

ALFA = 0.20

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;

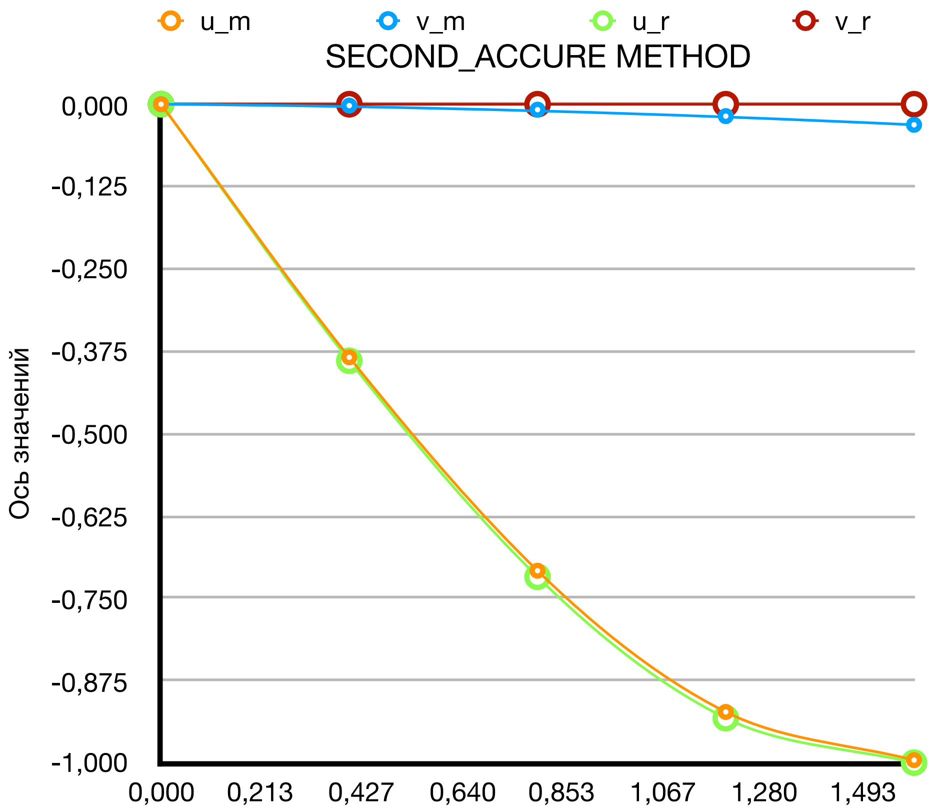
>____<

ALFA = 0.30

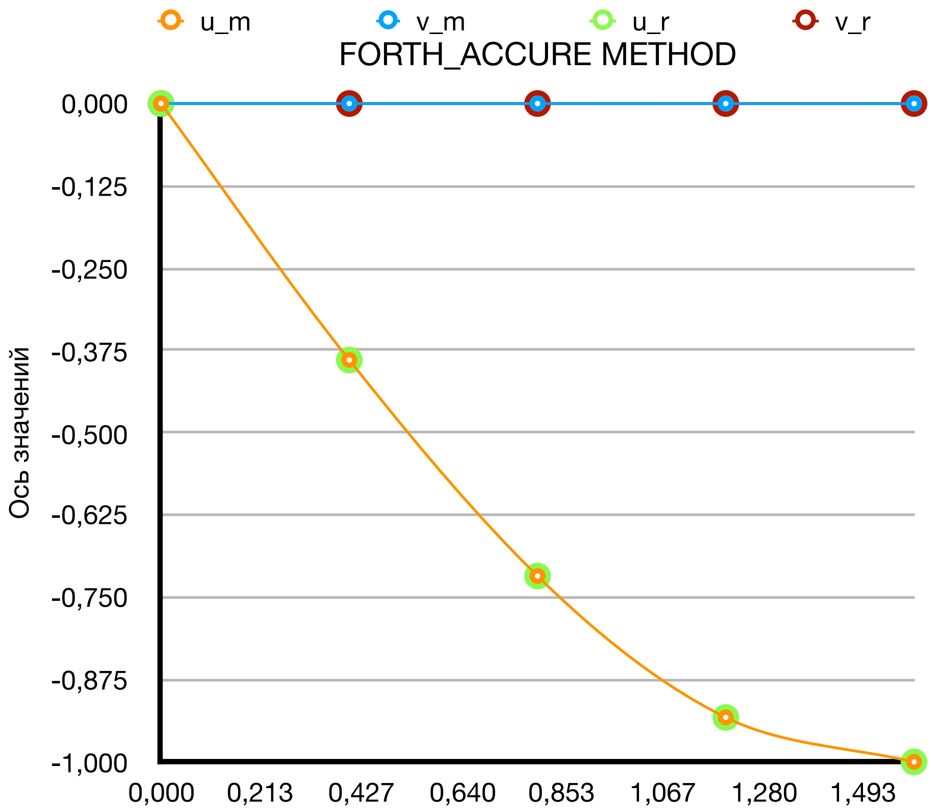
SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;



>____<



ALFA = 0.40

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;

>____<

ALFA = 0.50

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;

>____<

ALFA = 0.60

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408032490421253;

>____<

ALFA = 0.70

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;

>____<

ALFA = 0.80

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408005385366941;

>____<

ALFA = 0.90

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408032490421253;

>____<

ALFA = 1.00

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000004529223902448743094284;

Y2 = 0.000000000613178408032490421253;

>____<

>++<

TYPE = 4 __ LENGTH = 1.00000

COUNT_GRID = 10 __ GRID = 0.1000000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000001697376878871692008898808;

Y2 = 0.000000331459476518662079710942;

>____<

COUNT_GRID = 100 __ GRID = 0.0100000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000220184525722821183846;

Y2 = 0.000000000030911634146466027140;

>____<

COUNT_GRID = 1000 __ GRID = 0.0010000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000022601603748284571;

Y2 = 0.000000000000003064172179878533;

>____<

COUNT_GRID = 10000 __ GRID = 0.0001000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000000755146813136l5;

Y2 = 0.000000000000000212503625807l6;

>____<

COUNT_GRID = 100000 __ GRID = 0.0000100

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.0000000000000001842547382030491;

Y2 = 0.0000000000000000543429233904047;

>____<

COUNT_GRID = 1000000 __ GRID = 0.0000010

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.0000000000000001648702875611807 2;

Y2 = 0.0000000000000000494016719893020 1;

>____<

COUNT_GRID = 10000000 __ GRID = 0.0000001

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.0000000000000002412929881942527 1;

Y2 = 0.0000000000000000639221206348572 5;

>____<

SECOND_ACCURE METHOD

TYPE = 4 __ COUNT_GRID = 5

LENGTH = 2.00000 __ GRID = 0.40000 __ PARM = 0.50000

(x , u_m , v_m , u_r , v_r)

0.0000000000;0.0000000000;0.0000000000;0.0000000000;0.0000000000

0.4000000000;0.0800000000;0.0800000000;0.0918246976;0.0703200460

0.8000000000;0.3904000000;0.2624000000;0.4255409285;0.2493289641

1.2000000000;1.0417920000;0.5144320000;1.1201169227;0.5011942119

1.6000000000;2.1978521600;0.8138137600;2.3530324244;0.8018965180

END

FORTH_ACCURE METHOD

TYPE = 4 __ COUNT_GRID = 5 __ LENGTH = 2.00000 __ GRID = 0.40000

(x , u_m , v_m , u_r , v_r)

0.0000000000;0.0000000000;0.0000000000;0.0000000000;0.0000000000

0.4000000000;0.0917333333;0.0704000000;0.0918246976;0.0703200460

0.8000000000;0.4252683378;0.2494361600;0.4255409285;0.2493289641

1.2000000000;1.1195069551;0.5013020017;1.1201169227;0.5011942119

1.6000000000;2.3518191751;0.8019928619;2.3530324244;0.8018965180

На примере этой задачи очень хорошо видно, что метод Рунге-Кутта 4-ого порядка, заметно точнее чем 2-ого порядка, даже на малом количестве итераций метода.

Тест 5

Система:

$u' = v - cos(x),$

$v' = u + sin(x),$

Начальные условия:

$(x_0, u_0, v_0) = (0,0,0),$

Точное решение:

$u = -sin(x),$

$v = 0,$

Вывод программы:

TYPE = 3 __ COUNT_GRID = 10000

LENGTH = 1.00000 __ GRID = 0.00010

ALFA = 0.10

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000088358840272142850913 45;

Y2 = 0.0000000008376429508868937306 69;

>____<

ALFA = 0.20

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000035782813723791301319 11;

Y2 = 0.0000000008359112500751329563 99;

>____<

ALFA = 0.30

SECOND_ACCURE METHOD MAX ERROR

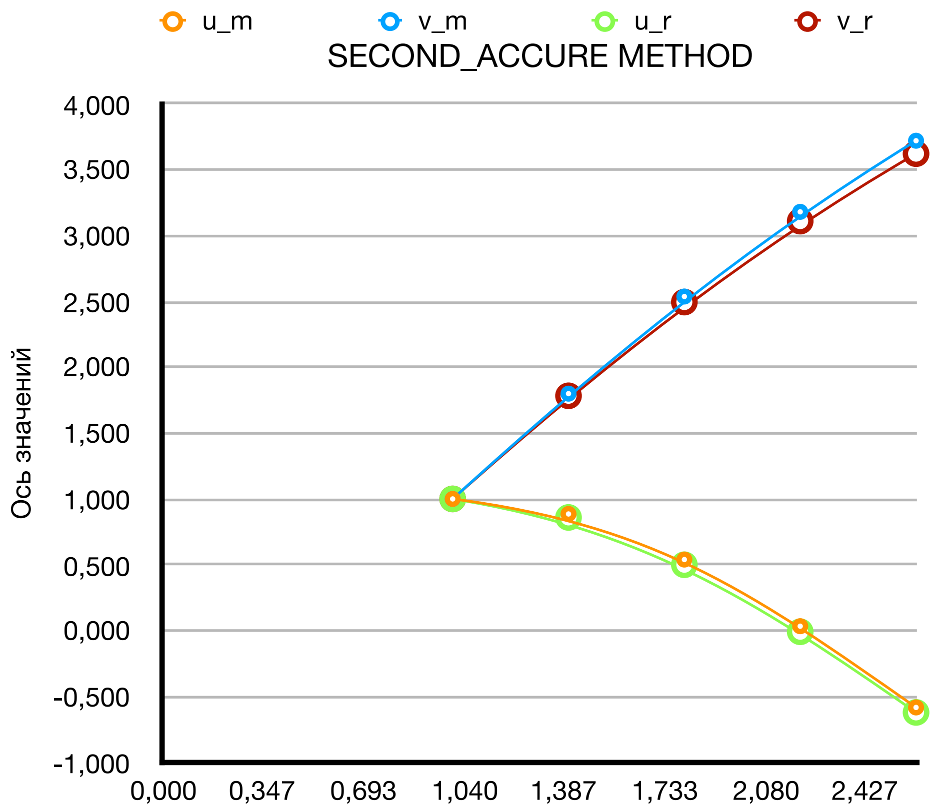
Y1 = 0.0000000018255150498343829246 21;

Y2 = 0.0000000008356060082334001402 01;

>____<

ALFA = 0.40

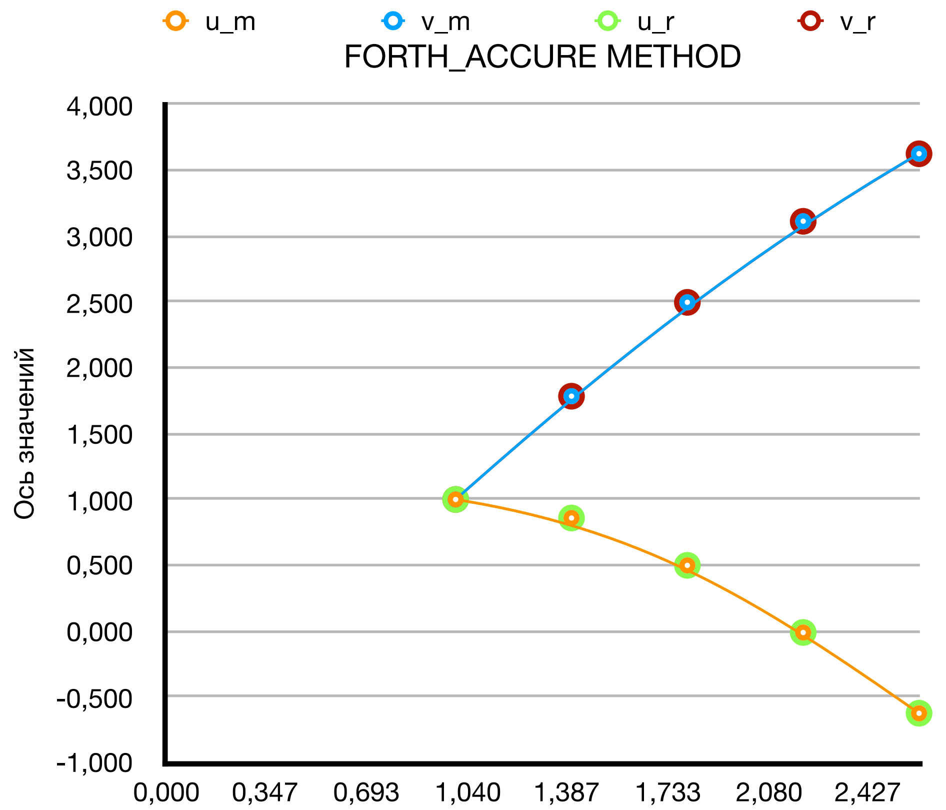
SECOND_ACCURE METHOD MAX ERROR



Y1 = 0.0000000009494753435085670489 89;

Y2 = 0.0000000008355043842045874548 56;

>____<



ALFA = 0.50

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000000456104503937428823468;

Y2 = 0.000000000835459729114283420344;

>____<

ALFA = 0.60

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000000176046329857654226814;

Y2 = 0.000000000835436759619246585086;

>____<

ALFA = 0.70

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000000177788542351717260281;

Y2 = 0.000000000835423682382907347887;

>____<

ALFA = 0.80

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000000365606027603101318979;

Y2 = 0.000000000835415696983133317415;

>____<

ALFA = 0.90

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000000511687212728911128279;

Y2 = 0.000000000835410564932955769416;

>____<

ALFA = 1.00

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.000000000628552740400672299570;

Y2 = 0.000000000835407138743585415715;

>____<

>++<

TYPE = 3 __ LENGTH = 1.00000

COUNT_GRID = 10 __ GRID = 0.1000000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000203387362549315407755923;

Y2 = 0.000000204624832502619174725001;

>____<

COUNT_GRID = 100 __ GRID = 0.0100000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000018045133041901667559;

Y2 = 0.000000000027041681448206177603;

>____<

COUNT_GRID = 1000 __ GRID = 0.0010000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000001784759406237013;

Y2 = 0.000000000000002776917121246940;

>____<

COUNT_GRID = 10000 __ GRID = 0.0001000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000000049005938196345;

Y2 = 0.000000000000000017867940181766;

>____<

COUNT_GRID = 100000 __ GRID = 0.0000100

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000000953067919723383;

Y2 = 0.000000000000000234394988916398;

>____<

COUNT_GRID = 1000000 __ GRID = 0.0000010

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000007762833344887587;

Y2 = 0.000000000000001526584030573355;

>____<

COUNT_GRID = 10000000 __ GRID = 0.0000001

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000021900721253881317;

Y2 = 0.000000000000010250508893654950;

>____<

SECOND_ACCURE METHOD

TYPE = 3 __ COUNT_GRID = 5

LENGTH = 2.00000 __ GRID = 0.40000 __ PARM = 0.50000

(x , u_m , v_m , u_r , v_r)

0.0000000000;0.0000000000;0.0000000000;-0.0000000000;0.0000000000

0.4000000000;-0.3842121988;-0.0021163315;-0.3894183423;0.0000000000

0.8000000000;-0.7081957806;-0.0083005105;-0.7173560909;0.0000000000

1.2000000000;-0.9225960527;-0.0181003649;-0.9320390860;0.0000000000

1.6000000000;-0.9957124025;-0.0312528978;-0.9995736030;0.0000000000

END

FORTH_ACCURE METHOD

TYPE = 3 __ COUNT_GRID = 5 __ LENGTH = 2.00000 __ GRID = 0.40000

(x , u_m , v_m , u_r , v_r)

0.0000000000;0.0000000000;0.0000000000;-0.0000000000;0.0000000000

0.4000000000;-0.3893864778;-0.0000042700;-0.3894183423;0.0000000000

0.8000000000;-0.7172951540;-0.0000300413;-0.7173560909;0.0000000000

1.2000000000;-0.9319653839;-0.0000741234;-0.9320390860;0.0000000000

1.6000000000;-0.9995154678;-0.0001341718;-0.9995736030;0.0000000000

Эта задача также очень хорошо показывает, что метод Рунге-Кутта 4-ого порядка, заметно точнее чем 2-ого порядка, даже на малом количестве итераций метода.

Тест 6

Система:

$$u' = \frac{(u - v)}{x},$$

$$v' = \frac{(u + v)}{x},$$

Начальные условия:

$(x_0, u_0, v_0) = (1,1,1),$

Точное решение:

$u = x \cdot (cos(ln(x)) - sin(ln(x))),$

$v = x \cdot (cos(ln(x)) + sin(ln(x))),$

Вывод программы:

TYPE = 2 __ COUNT_GRID = 10000

LENGTH = 1.00000 __ GRID = 0.00010

ALFA = 0.10

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000253168523895006165880 25;

Y2 = 0.00000002427303106211295413707 8;

>____<

ALFA = 0.20

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000109011438070036412867 35;

Y2 = 0.0000000123018669972996264050 38;

>____<

ALFA = 0.30

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.00000000609466196539970410384 1;

Y2 = 0.00000000831044453196948251783 0;

>____<

ALFA = 0.40

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.00000000369181459898169569977 5;

Y2 = 0.0000000063145392859877136348 96;

>______________________________________<

ALFA = 0.50

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000023055266500993706196 84;

Y2 = 0.0000000051169340411962721537 62;

>______________________________________<

ALFA = 0.60

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000014596079668809276708 16;

Y2 = 0.0000000043185046673648724846 47;

>______________________________________<

ALFA = 0.70

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000009216846243468787558 41;

Y2 = 0.0000000037481852907875201630 36;

>______________________________________<

ALFA = 0.80

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000005732149391647969882 69;

Y2 = 0.0000000033204388325252381930 50;

>______________________________________<

ALFA = 0.90

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000003466196593529362324 22;

Y2 = 0.0000000029877430412529604186 74;

>______________________________________<

ALFA = 1.00

SECOND_ACCURE METHOD MAX ERROR

Y1 = 0.0000000006354593438094319896O3;

Y2 = 0.0000000002721583837635155322943;

>____<

>+++++++++++++++++++++++++++++++++++++++<

TYPE = 2 __ LENGTH = 1.00000

COUNT_GRID = 10 __ GRID = 0.1000000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000001855775217779389186373296;

Y2 = 0.000002919486374277106485730648;

>____<

COUNT_GRID = 100 __ GRID = 0.0100000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000243040720876571900155;

Y2 = 0.000000000300205095157823897978;

>____<

COUNT_GRID = 1000 __ GRID = 0.0010000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000024856608741782860;

Y2 = 0.000000000000030000307793542902;

>____<

COUNT_GRID = 10000 __ GRID = 0.0001000

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.0000000000000000802933023888452;

Y2 = 0.0000000000000000962771529167128;

>____<

COUNT_GRID = 100000 __ GRID = 0.0000100

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000004636780326014445;

Y2 = 0.000000000000005609878880874497;

>____<

COUNT_GRID = 1000000 __ GRID = 0.0000010

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000043016019348959822;

Y2 = 0.000000000000051802312439619413;

>____<

COUNT_GRID = 10000000 __ GRID = 0.0000001

FORTH_ACCURE METHOD MAX ERROR

Y1 = 0.000000000000917362648518438295;

Y2 = 0.000000000001104525975889614209;

>____<

SECOND_ACCURE METHOD

TYPE = 2 __ COUNT_GRID = 5

LENGTH = 2.00000 __ GRID = 0.40000 __ PARM = 0.50000

(x , u_m , v_m , u_r , v_r)

1.0000000000;1.0000000000;1.0000000000;1.0000000000;1.0000000000

1.4000000000;0.8857142857;1.8000000000;0.8592724725;1.7837182563

1.8000000000;0.5392290249;2.5383219955;0.4997691804;2.4960430217

2.2000000000;0.0328133947;3.1818365056;-0.0095179724;3.1112552785

2.6000000000;-0.5846964003;3.7222760244;-0.6224864518;3.6238806020

END

FORTH_ACCURE METHOD

TYPE = 2 __ COUNT_GRID = 5 __ LENGTH = 2.00000 __ GRID = 0.40000

(x , u_m , v_m , u_r , v_r)

1.0000000000;1.0000000000;1.0000000000;1.0000000000;1.0000000000

1.4000000000;0.8592592593;1.7841269841;0.8592724725;1.7837182563

1.8000000000;0.4995765796;2.4966686263;0.4997691804;2.4960430217

2.2000000000;-0.0099311815;3.1120010944;-0.0095179724;3.1112552785

2.6000000000;-0.6231329141;3.6246862576;-0.6224864518;3.6238806020

Тест 7

К сожалению для задания этого варианта, нет аналитического решения, поэтому основываясь на трех предыдущих тестах, положим, что алгоритм работает верно, и по этой системе приведем только решение.

Система:

$$u' = -2 \cdot x \cdot u^2 + v^2 - x - 1,$$

$$v' = \frac{1}{v^2} - u - \frac{x}{u},$$

Начальные условия:

$$(x_0, u_0, v_0) = (0,1,1),$$

Точное решение:

$$u = NOTFOUND,$$

$$v = NOTFOUND,$$

Вывод программы:

TYPE = 1 __ COUNT_GRID = 50

LENGTH = 1.00000 __ GRID = 0.02000

SECOND_ACCURE METHOD

$X = 0.0000000000$, $Y1 = 1.0000000000$, $Y2 = 1.0000000000$

$X = 0.0200000000$, $Y1 = 0.9994000000$, $Y2 = 0.9998000000$

$X = 0.0400000000$, $Y1 = 0.9975877675$, $Y2 = 0.9992388423$

$X = 0.0600000000$, $Y1 = 0.9945583156$, $Y2 = 0.9983524260$

$X = 0.0800000000$, $Y1 = 0.9903138316$, $Y2 = 0.9971738770$

$X = 0.1000000000$, $Y1 = 0.9848634419$, $Y2 = 0.9957335049$

$X = 0.1200000000$, $Y1 = 0.9782228849$, $Y2 = 0.9940587528$

$X = 0.1400000000$, $Y1 = 0.9704140986$, $Y2 = 0.9921741446$

$X = 0.1600000000$, $Y1 = 0.9614647321$, $Y2 = 0.9901012310$

$X = 0.1800000000$, $Y1 = 0.9514075928$, $Y2 = 0.9878585373$

$X = 0.2000000000$, $Y1 = 0.9402800407$, $Y2 = 0.9854615144$

$X = 0.2200000000$, $Y1 = 0.9281233425$, $Y2 = 0.9829224928$

$X = 0.2400000000$, $Y1 = 0.9149819992$, $Y2 = 0.9802506403$

$X = 0.2600000000$, $Y1 = 0.9009030592$, $Y2 = 0.9774519225$

$X = 0.2800000000$, $Y1 = 0.8859354301$, $Y2 = 0.9745290649$

$X = 0.3000000000$, $Y1 = 0.8701291989$, $Y2 = 0.9714815138$

$X = 0.3200000000$, $Y1 = 0.8535349725$, $Y2 = 0.9683053939$

$X = 0.3400000000$, $Y1 = 0.8362032450$, $Y2 = 0.9649934586$

$X = 0.3600000000$, $Y1 = 0.8181837997$, $Y2 = 0.9615350289$

$X = 0.3800000000$, $Y1 = 0.7995251506$, $Y2 = 0.9579159157$

$X = 0.4000000000$, $Y1 = 0.7802740272$, $Y2 = 0.9541183185$

$X = 0.4200000000$, $Y1 = 0.7604749031$, $Y2 = 0.9501206942$

$X = 0.4400000000$, $Y1 = 0.7401695696$, $Y2 = 0.9458975861$

$X = 0.4600000000$, $Y1 = 0.7193967528$, $Y2 = 0.9414194024$

$X = 0.4800000000$, $Y1 = 0.6981917707$, $Y2 = 0.9366521313$

$X = 0.5000000000$, $Y1 = 0.6765862266$, $Y2 = 0.9315569750$

$X = 0.5200000000$, $Y1 = 0.6546077339$, $Y2 = 0.9260898839$

$X = 0.5400000000$, $Y1 = 0.6322796659$, $Y2 = 0.9202009612$

$X = 0.5600000000$, $Y1 = 0.6096209226$, $Y2 = 0.9138337068$

X = 0.5800000000 , Y1 = 0.5866457073 , Y2 = 0.9069240521

X = 0.6000000000 , Y1 = 0.5633633025 , Y2 = 0.8993991265

X = 0.6200000000 , Y1 = 0.5397778350 , Y2 = 0.8911756743

X = 0.6400000000 , Y1 = 0.5158880175 , Y2 = 0.8821580057

X = 0.6600000000 , Y1 = 0.4916868523 , Y2 = 0.8722353280

X = 0.6800000000 , Y1 = 0.4671612798 , Y2 = 0.8612782272

X = 0.7000000000 , Y1 = 0.4422917518 , Y2 = 0.8491339768

X = 0.7200000000 , Y1 = 0.4170517031 , Y2 = 0.8356201856

X = 0.7400000000 , Y1 = 0.3914068893 , Y2 = 0.8205160494

X = 0.7600000000 , Y1 = 0.3653145488 , Y2 = 0.8035500557

X = 0.7800000000 , Y1 = 0.3387223341 , Y2 = 0.7843822949

X = 0.8000000000 , Y1 = 0.3115669386 , Y2 = 0.7625782984

X = 0.8200000000 , Y1 = 0.2837723203 , Y2 = 0.7375690672

X = 0.8400000000 , Y1 = 0.2552473901 , Y2 = 0.7085875837

X = 0.8600000000 , Y1 = 0.2258829938 , Y2 = 0.6745631224

X = 0.8800000000 , Y1 = 0.1955479895 , Y2 = 0.6339348149

X = 0.9000000000 , Y1 = 0.1640842962 , Y2 = 0.5842978547

X = 0.9200000000 , Y1 = 0.1313013046 , Y2 = 0.5216650767

X = 0.9400000000 , Y1 = 0.0969725713 , Y2 = 0.4387110992

X = 0.9600000000 , Y1 = 0.0608502531 , Y2 = 0.3196138370

X = 0.9800000000 , Y1 = 0.0227844487 , Y2 = 0.0961839859

TYPE = 1 __ COUNT_GRID = 50

LENGTH = 1.00000 __ GRID = 0.02000

FORTH_ACCURE METHOD

X = 0.0000000000 , Y1 = 1.0000000000 , Y2 = 1.0000000000

X = 0.0200000000 , Y1 = 0.9993976407 , Y2 = 0.9998065532

X = 0.0400000000 , Y1 = 0.9975835693 , Y2 = 0.9992515164

X = 0.0600000000 , Y1 = 0.9945527665 , Y2 = 0.9983707873

X = 0.0800000000 , Y1 = 0.9903073819 , Y2 = 0.9971974934

X = 0.1000000000 , Y1 = 0.9848564998 , Y2 = 0.9957619495

X = 0.1200000000 , Y1 = 0.9782158126 , Y2 = 0.9940916076

X = 0.1400000000 , Y1 = 0.9704072094 , Y2 = 0.9922110052

X = 0.1600000000 , Y1 = 0.9614582891 , Y2 = 0.9901417109

X = 0.1800000000 , Y1 = 0.9514018081 , Y2 = 0.9879022727

X = 0.2000000000 , Y1 = 0.9402750763 , Y2 = 0.9855081687

X = 0.2200000000 , Y1 = 0.9281193119 , Y2 = 0.9829717614

X = 0.2400000000 , Y1 = 0.9149789700 , Y2 = 0.9803022549

X = 0.2600000000 , Y1 = 0.9009010571 , Y2 = 0.9775056558

X = 0.2800000000 , Y1 = 0.8859344427 , Y2 = 0.9745847345

X = 0.3000000000 , Y1 = 0.8701291812 , Y2 = 0.9715389860

X = 0.3200000000 , Y1 = 0.8535358515 , Y2 = 0.9683645875

X = 0.3400000000 , Y1 = 0.8362049257 , Y2 = 0.9650543486

X = 0.3600000000 , Y1 = 0.8181861703 , Y2 = 0.9615976500

X = 0.3800000000 , Y1 = 0.7995280883 , Y2 = 0.9579803656

X = 0.4000000000 , Y1 = 0.7802774029 , Y2 = 0.9541847617

X = 0.4200000000 , Y1 = 0.7604785862 , Y2 = 0.9501893661

X = 0.4400000000 , Y1 = 0.7401734328 , Y2 = 0.9459687975

X = 0.4600000000 , Y1 = 0.7194006752 , Y2 = 0.9414935451

X = 0.4800000000 , Y1 = 0.6981956410 , Y2 = 0.9367296846

X = 0.5000000000 , Y1 = 0.6765899458 , Y2 = 0.9316385146

X = 0.5200000000 , Y1 = 0.6546112165 , Y2 = 0.9261760921

X = 0.5400000000 , Y1 = 0.6322828409 , Y2 = 0.9202926414

X = 0.5600000000 , Y1 = 0.6096237334 , Y2 = 0.9139318015

X = 0.5800000000 , Y1 = 0.5866481104 , Y2 = 0.9070296666

X = 0.6000000000 , Y1 = 0.5633652659 , Y2 = 0.8995135614

X = 0.6200000000 , Y1 = 0.5397793343 , Y2 = 0.8913004675

X = 0.6400000000 , Y1 = 0.5158890312 , Y2 = 0.8822949910

X = 0.6600000000 , Y1 = 0.4916873548 , Y2 = 0.8723867140

X = 0.6800000000 , Y1 = 0.4671612323 , Y2 = 0.8614467088

X = 0.7000000000 , Y1 = 0.4422910893 , Y2 = 0.8493228929

X = 0.7200000000 , Y1 = 0.4170503172 , Y2 = 0.8358337483

X = 0.7400000000 , Y1 = 0.3914046041 , Y2 = 0.8207596848

X = 0.7600000000 , Y1 = 0.3653110875 , Y2 = 0.8038309278

X = 0.7800000000 , Y1 = 0.3387172709 , Y2 = 0.7847101355

X = 0.8000000000 , Y1 = 0.3115596293 , Y2 = 0.7629667797

X = 0.8200000000 , Y1 = 0.2837617985 , Y2 = 0.7380381832

X = 0.8400000000 , Y1 = 0.2552322092 , Y2 = 0.7091680316

X = 0.8600000000 , Y1 = 0.2258609775 , Y2 = 0.6753049667

X = 0.8800000000 , Y1 = 0.1955158277 , Y2 = 0.6349262582

X = 0.9000000000 , Y1 = 0.1640368703 , Y2 = 0.5857110423

X = 0.9200000000 , Y1 = 0.1312305520 , Y2 = 0.5238876151

X = 0.9400000000 , Y1 = 0.0968657472 , Y2 = 0.4428246916

X = 0.9600000000 , Y1 = 0.0606893079 , Y2 = 0.3299327788

X = 0.9800000000 , Y1 = 0.0225704039 , Y2 = 0.1443215101

Заметим, что решения также обладают некоторым отклонением друг от друга, что естественно.

Приложение 3

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <math.h>
#include <tgmath.h>
#include <fcntl.h>
#include <unistd.h>

// test1
long double s_a_1 = 1.0; //alfa1
long double s_a_2 = -2.0; //alfa2
long double s_point_1 = 1.0; //point start fo frst edje
long double s_sig_1 = 0.6; //sig in start point 1
long double s_point_2 = 1.3;  //point start fo scnd edje
long double s_sig_2 = 1; //sig in start point 2
// test2
// long double s_a_1 = 1.0; //alfa1
// long double s_a_2 = 1.0; //alfa2
// long double s_point_1 = 0.5; //point start fo frst edje
// long double s_sig_1 = 1.0; //sig in start point 1
// long double s_point_2 = 1.5;  //point start fo scnd edje
// long double s_sig_2 = 3; //sig in start point 2
// test3
// long double s_a_1 = 1.0; //alfa1
// long double s_a_2 = 1.0; //alfa2
// long double s_point_1 = 1.0; //point start fo frst edje
// long double s_sig_1 = 1.0; //sig in start point 1
// long double s_point_2 = 1.5;  //point start fo scnd edje
```

```
// long double s_sig_2 = 4; //sig in start point 2


// y''-0.5y'+3y=2x^2, y(1) - 2y'(1) = 0.6, y(1.3) = 1 //test1
// y''+1.5y'-5y=3x-1, y(0.5) + y'(0.5) = 1, y(1.5) = 3 //test2
// y''+1.5y'-5y=3x^3 + 8, y(1) + y'(1) = 1, y(1.5) = 4 //test3



inline long double task_1_real_solv( long double x ); //real solution of DD you can made it
inline long double q( long double x ); //q function its y koef
inline long double p( long double x ); //q function its y' koef
inline long double f( long double x ); //right part of DD
void          task_1( int count_grid ); //solve function


int
main( int argc , char **argv )
{
   if ( argc <= 1 ) { //help
      printf( "!!!README!!!\n" );
      printf( "YOU SHOULD SWAP P , Q , F function\n" );
      printf( "START PARAMETRS LIKE IN COMMENTS\n" );
      printf( "arg1 = count_grid\n" );
      printf( "arg2 = 1 solve , 2 test all\n" );
      printf( "!!!README!!!\n" );
   }
   if ( strtol( argv[2] , NULL , 10 ) == 1 ) {
      int count_grid = (int) strtol( argv[1] , NULL , 10 );
      task_1( count_grid );
   } else {
      for ( int i = 10 ; i < 10000000 ; i *= 10 ) {
```

```
            task_1( i );
        }
    }
    return 0;
}

inline long double
task_1_real_solv( long double x )
{
    return 2.0/3.0 *(x * x + 1.0/3.0 *x + 0.704467 * exp(x / 4) *sin(sqrt(47)*x/4) +
    0.927242*exp(x/4)*cos(sqrt(47)*x/4) -0.611111); //test1
    /*
    return 0.02 + 0.324268 * expl( -3.1085 * x ) + 0.347255 * expl( 1.6085 * x ) - 0.6 * x;
    */
    /*
    return -2.1292 + 33.5105 * expl( -3.1085 * x ) + 0.951122 * expl( 1.6085 * x ) -
    1.044 * x - 0.54 * x * x - 0.6 * x * x * x; //test3
    */
}

inline long double
p( long double x )
{
    return (-1.0) / 2; //test1
    // return 3.0 / 2; //test2
    // return 1.5; //test3
}

inline long double
```

```
q( long double x )
{
    return 3.0; //test1
    // return -5.0; //test2
    // return -5.0; //test3
}

inline long double
f( long double x )
{
    return 2.0 * x * x; //test1
    // return 3.0 * x - 1;
    // return 3.0 * x * x * x + 8.0; //test3
}

void
task_1( int count_grid )
{
    long double A , B , C , D; //cur memory
    long double B_0 , C_0 , D_0; //cur memory
    long double A_F , B_F , C_F , D_F; //cur memory
    long double a = s_point_1 , b = s_point_2; //cur memory and set sig
    long double grid = ( b - a ) / count_grid; //cur memory and set sig
    long double x = a - grid / 2.0; //cur memory and set sig
    //__________________________________________
    long double *alfa = calloc( count_grid + 1 , sizeof( *alfa ) ); //get memory
    long double *beta = calloc( count_grid + 1 , sizeof( *beta ) ); //get memory
    long double *solv = calloc( count_grid + 1 , sizeof( *solv ) ); //get memory
    //__________________________________________
```

```
B_0 = ( ( s_a_1 / 2.0 ) + ( s_a_2 / grid ) ); //set sig
C_0 = ( ( s_a_1 / 2.0 ) - ( s_a_2 / grid ) ); //set sig
D_0 = s_sig_1; //set sig
alfa[0] = (-1) * B_0 / C_0; //write koef
beta[0] = D_0 / C_0; //write koef
//__________________________________________
for ( int i = 1 ; i < count_grid ; i++ ) { //calc koef
    x += grid; //next iteration
    A = 1.0 / ( grid * grid ) - p( x ) / ( 2.0 * grid ); //set sig
    B = 1.0 / ( grid * grid ) + p( x ) / ( 2.0 * grid ); //set sig
    C = q( x ) - 2.0 / ( grid * grid ); //set sig
    D = f( x ); //set sig
    alfa[i] = (-1) * B / ( A * alfa[i - 1] + C ); //write koef
    beta[i] = ( D - A * beta[i - 1] ) / ( A * alfa[i - 1] + C ); //write koef
}
//__________________________________________
x += grid; //next iteration
A_F = 1.0 / ( grid * grid ) - p( x ) / grid; //set sig
B_F = 2.0 / ( grid * grid ); //set sig
C_F = p( x ) / grid + q( x ) - 3.0 / ( grid * grid ); //set sig
D_F = f( x ); //set sig
alfa[count_grid] = (-1) * B_F / ( A_F * alfa[count_grid - 1] + C_F ); //write koef
beta[count_grid] = ( D_F - A_F * beta[count_grid - 1] ) / ( A_F * alfa[count_grid - 1] + C_F
); //write koef
solv[count_grid] = s_sig_2; //write solve
//__________________________________________
for ( int i = count_grid - 1 ; i >= 0 ; i-- ) {
    solv[i] = alfa[i + 1] * solv[i + 1] + beta[i + 1]; //write solve
}
```

```
//___________________________________________
printf( "X = %.10Lf __ Y = %.10Lf __ R = %.10Lf\n" , x + ( grid / 2 ) , solv[count_grid] ,
task_1_real_solv( x + ( grid / 2 ) ) );  //print info
long double ERROR = -1.0; //cur memory
for ( int i = count_grid - 1 ; i >= 0 ; i-- ) { //write all information
    long double cur = fabsl( solv[i] - task_1_real_solv( x ) );
    ERROR = cur > ERROR ? cur : ERROR;
    printf( "X = %.10Lf __ Y = %.10Lf __ R = %.10Lf\n" , x , solv[i] ,
task_1_real_solv( x ) ); //print info
    x -= grid; //pre iteration
}
printf( "COUNT_GRID :: %d\nGRID :: %.8Lf\nMAX ERROR :: %.20Lf\n" , count_grid , grid
, ERROR ); //print info
//___________________________________________
free( alfa ); //free memory
free( beta ); //free memory
free( solv ); //free memory
}
```

Приложение 4

Тест 1

Уравнение:

$$y'' - 0.5 \cdot y' + 3y = 2 \cdot x^2$$

Начальные условия:

$$y(1) - 2 \cdot y'(1) = 0.6$$

$$y(1.3) = 1$$

Решение *wolfram*:

$$y(x) = \frac{2}{3} \cdot (x^2 + \frac{1}{3} \cdot x + 0.704467 \cdot e^{\frac{x}{4}} \cdot sin(\sqrt{47} \cdot \frac{x}{4}) + 0.927242 \cdot e^{\frac{x}{4}} \cdot cos(\sqrt{47} \cdot \frac{x}{4}) - 0.611111)$$

Вывод программы:

COUNT_GRID :: 10
GRID :: 0.03000000
MAX ERROR :: 0.00004508541744907221
COUNT_GRID :: 100
GRID :: 0.00300000
MAX ERROR :: 0.00000116589146485920
COUNT_GRID :: 1000
GRID :: 0.00030000
MAX ERROR :: 0.00000068549740229508
COUNT_GRID :: 10000
GRID :: 0.00003000
MAX ERROR :: 0.00000068068083722847
COUNT_GRID :: 100000
GRID :: 0.00000300
MAX ERROR :: 0.00000068063574484975
COUNT_GRID :: 1000000
GRID :: 0.00000030
MAX ERROR :: 0.00000068063560075527

Стоит обратить внимание на то, что точность перестает расти, так как решение, предоставленное сервисом *wolframalpha* имеет приближенный вид.

Решение данного уравнения:

(x,y_m,y_w)
1.3000000000;1.0000000000;1.0000006806

1.2850000000;0.9983213452;0.9983664306

1.2550000000;0.9952862987;0.9953245136

1.2250000000;0.9924431109;0.9924745187

1.1950000000;0.9896636218;0.9896883377

1.1650000000;0.9868246246;0.9868428139

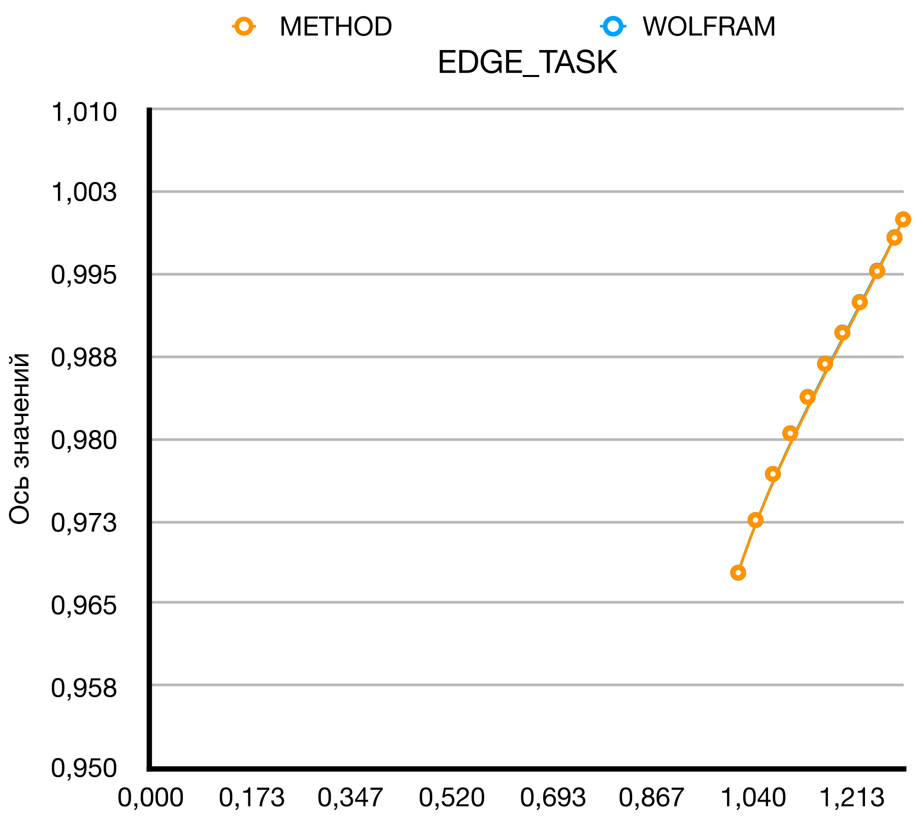
1.1350000000;0.9838081221;0.9838199981

1.1050000000;0.9805015657;0.9805073873

1.0750000000;0.9767980753;0.9767981450

1.0450000000;0.9725966421;0.9725913032

1.0150000000;0.9678023112;0.9677919451



Тест 2

Уравнение:

$y'' + 1.5 \cdot y' - 5 \cdot y = 3 \cdot x - 1$

Начальные условия:

$y(0.5) + y'(0.5) = 1$

$y(1.5) = 3$

Решение *wolfram*:

$y(x) = 0.02 + 0.324268 \cdot e^{-3.1085 \cdot x} + 0.347255 \cdot e^{1.6085 \cdot x} - 0.6 \cdot x$

Вывод программы:

COUNT_GRID :: 10

GRID :: 0.10000000

MAX ERROR :: 0.01413866240702971893

COUNT_GRID :: 100

GRID :: 0.01000000

MAX ERROR :: 0.00015938498418345049

COUNT_GRID :: 1000

GRID :: 0.00100000

MAX ERROR :: 0.00003316607177682065

COUNT_GRID :: 10000

GRID :: 0.00010000

MAX ERROR :: 0.00003195041034603796

COUNT_GRID :: 100000

GRID :: 0.00001000

MAX ERROR :: 0.00003194109822559515

COUNT_GRID :: 1000000

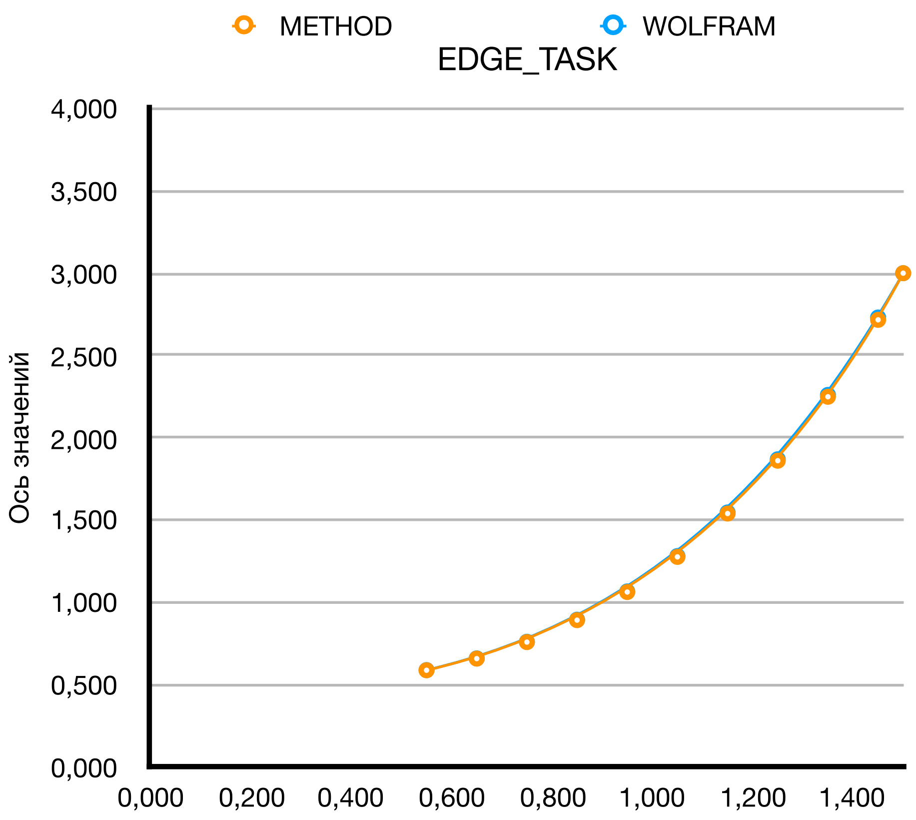
GRID :: 0.00000100

MAX ERROR :: 0.00003194128781393323

Стоит обратить внимание на то, что точность перестает расти, так как решение, предоставленное сервисом *wolframalpha* имеет приближенный вид.

Решение данного уравнения:

(x,y_m,y_w)
1.5000000000;3.0000000000;3.0000319413
1.4500000000;2.7168116602;2.7309503226
1.3500000000;2.2497103700;2.2607269941
1.2500000000;1.8614418636;1.8699534084
1.1500000000;1.5405591055;1.5470688267
1.0500000000;1.2774012572;1.2823191378
0.9500000000;1.0638611231;1.0675208510
0.8500000000;0.8931988658;0.8958719768
0.7500000000;0.7598994243;0.7618071366
0.6500000000;0.6595730153;0.6608962121
0.5500000000;0.5889003246;0.5897880893



Тест 3

Уравнение:

$$y'' + 1.5 \cdot y' - 5 \cdot y = 3 \cdot x^3 + 8$$

Начальные условия:

$$y(1) + y'(1) = 1$$

$$y(1.5) = 4$$

Решение *wolfram*:

$$y(x) = -2.1292 + 33.5105 \cdot e^{-3.1085 \cdot x} + 0.951122 \cdot e^{1.6085 \cdot x} - 1.044 \cdot x - 0.54 \cdot x^2 - 0.6 \cdot x^3$$

Вывод программы:

COUNT_GRID :: 10

GRID :: 0.05000000

MAX ERROR :: 0.00817187138498614760

COUNT_GRID :: 100

GRID :: 0.00500000

MAX ERROR :: 0.00014738582036768078

COUNT_GRID :: 1000

GRID :: 0.00050000

MAX ERROR :: 0.00007272320622895298

COUNT_GRID :: 10000

GRID :: 0.00005000

MAX ERROR :: 0.00007201750104327974

COUNT_GRID :: 100000

GRID :: 0.00000500

MAX ERROR :: 0.00007201396172044775

COUNT_GRID :: 1000000

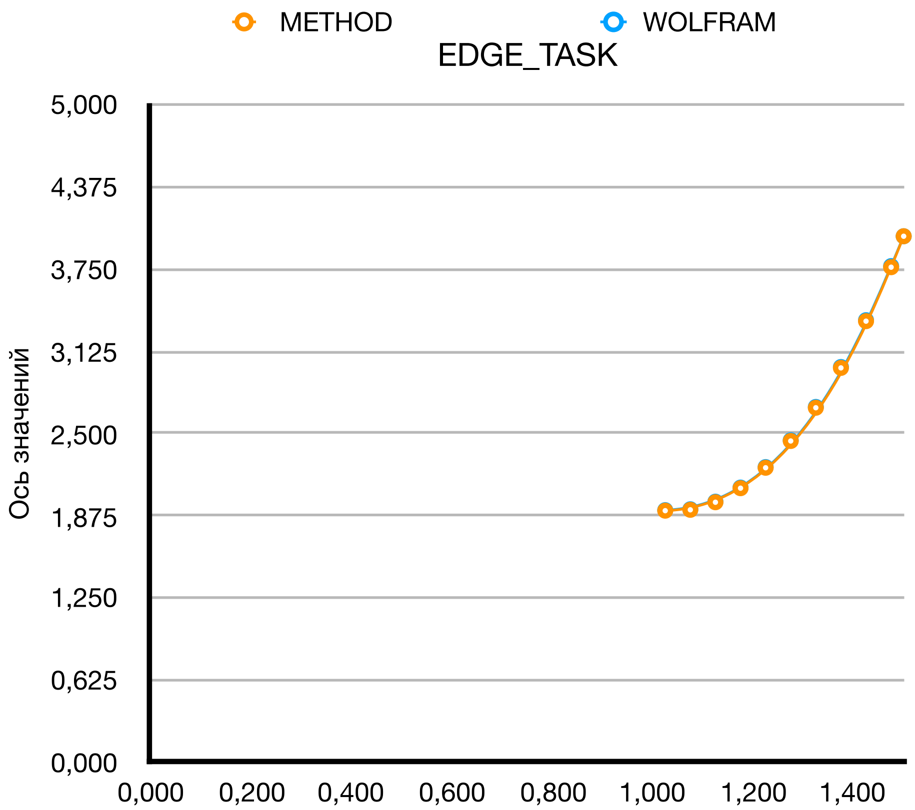
GRID :: 0.00000050

MAX ERROR :: 0.00007201427711185583

Стоит обратить внимание на то, что точность перестает расти, так как решение, предоставленное сервисом *wolframalpha* имеет приближенный вид.

Решение данного уравнения:

(x,y_m,y_w)
1.5000000000;4.0000000000;4.0000720143
1.4750000000;3.7647582244;3.7729300958
1.4250000000;3.3546433899;3.3619259952
1.3750000000;2.9994654224;3.0059771760
1.3250000000;2.6966012633;2.7024523243
1.2750000000;2.4440635740;2.4493572404
1.2250000000;2.2405186194;2.2453527004
1.1750000000;2.0853140319;2.0897822176
1.1250000000;1.9785175889;1.9827108396
1.0750000000;1.9209683642;1.9249763423
1.0250000000;1.9143418816;1.9182544514



Тест 4

Уравнение:

$y'' - y' = 0$

Начальные условия:

$-y(0) + y'(0) = 0$

$y(1) = 1$

Точное решение:

$y(x) = e^{x-1}$

Вывод программы:

COUNT_GRID :: 10

GRID :: 0.10000000

MAX ERROR :: 0.00100318015682260639

COUNT_GRID :: 100

GRID :: 0.01000000

MAX ERROR :: 0.00001223168018293149

COUNT_GRID :: 1000

GRID :: 0.00100000

MAX ERROR :: 0.00000012472941915371

COUNT_GRID :: 10000

GRID :: 0.00010000

MAX ERROR :: 0.00000000124972913138

COUNT_GRID :: 100000

GRID :: 0.00001000

MAX ERROR :: 0.00000000001250100078

COUNT_GRID :: 1000000

GRID :: 0.00000100

MAX ERROR :: 0.00000000000013595527

Этот пример позволяет убедиться в том, что алгоритм работает верно, и что приближение к точному решению уравнения с увеличением числа узлов сетки растет. А следовательно уменьшается погрешность решения.

(x,y_m,y_w)
1.0000000000;1.0000000000;1.0000000000
0.9500000000;0.9502262443;0.9512294245
0.8500000000;0.8597285068;0.8607079764
0.7500000000;0.7778496014;0.7788007831
0.6500000000;0.7037686870;0.7046880897
0.5500000000;0.6367430977;0.6376281516
0.4500000000;0.5761008979;0.5769498104
0.3500000000;0.5212341458;0.5220457768
0.2500000000;0.4715927985;0.4723665527
0.1500000000;0.4266791987;0.4274149319
0.0500000000;0.3860430845;0.3867410235