Московский Авиационный Институт
(Государственный Технический Университет)

Факультет прикладной математики и физики.
Кафедра вычислительной математики и программирования.

Лабораторная работа №4
по курсу «Численные методы»

VI семестр

Студент Баскаков О.А.
Группа 08-306
Вариант 1

Москва, 2011.

Постановка задачи

1 Численные методы решения задачи Коши

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

2 Численные методы решение краевой задачи для ОДУ

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Теоретическая часть

ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ КОШИ

Задача Коши для одного обыкновенного дифференциального уравнения.

Рассматривается задача Коши для одного дифференциального уравнения первого порядка разрешенного относительно производной

$$\begin{aligned} y' &= f(x, y) \\ y(x_0) &= y_0 \end{aligned} \qquad (4.1)$$

Требуется найти решение на отрезке $[a,b]$, где $x_0 = a$.

Введем разностную сетку на отрезке $[a,b]$ $\Omega^{(k)} = \{x_k = x_0 + hk\}$, $k = 0,1,\ldots,N$, $h = |b-a|/N$.

Точки x_k - называются *узлами* разностной сетки, расстояния между узлами - *шагом* разностной сетки *(h)* , а совокупность значений какой либо величины заданных в узлах сетки называется *сеточной функцией* $y^{(h-1)} = \{y_k, k = 0,1,\ldots, N\}$.

Приближенное решение задачи Коши (4.1) будем искать численно в виде сеточной функции $y^{(h)}$. Для оценки погрешности приближенного численного решения $y^{(h)}$ будем рассматривать это решение как элемент N+1- мерного линейного векторного этого пространства $\delta^{(h)} = y^{(h)} - [y]^{(h)}$, где $[y]^{(h)}$ - точное решение задачи (1) в узлах расчетной сетки. Таким образом $\varepsilon_h = \|\delta^{(h)}\|$.

пространства с какой либо нормой. В качестве погрешности решения принимается норма элемента

4.1.2. Одношаговые методы

Метод Эйлера (явный).

Метод Эйлера играет важную роль в теории численных методов решения ОДУ, хотя и не часто используется в практических расчетах из-за невысокой точности. Вывод расчетных соотношений для этого метода может быть произведен несколькими способами: с помощью геометрической интерпретации, с использованием разложения в

ряд Тейлора, конечно разностным методом (с помощью разностной аппроксимации производной), квадратурным способом (использованием эквивалентного интегрального уравнения).

Рассмотрим вывод соотношений метода Эйлера геометрическим способом. Решение в узле x_0 известно из начальных условий рассмотрим процедуру получения решения в узле x_1 рис.4.1.

График функции $y^{(h)}$, которая является решением задачи Коши (1), представляет собой гладкую кривую, проходящую через точку (x_0, y_0) согласно условию $y(x_0) = y_0$, и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точке x_0 и равен значению правой части дифференциального уравнения в точке (x_0,y_0) согласно выражению $y'(x_0) = f(x_0,y_0)$. В случае небольшого шага разностной сетки h график функции и график касательной не успевают сильно разойтись друг от друга и можно в качестве значения решения в узле x_1 принять значение касательной y_1, вместо значения неизвестного точного решения y_{1ucm}. При этом допускается погрешность $|y_1 - y_{1ucm}|$ геометрически представленная отрезком CD на рис.4.1. Из прямоугольного треугольника ABC находим $CB=BA\ tg(CAB)$ или $\Delta y = hy(x_0)$. Учитывая, что $\Delta y = y_1 - y_0$ и заменяя производную $y^1(x_0)$ на правую часть дифференциального уравнения , получаем соотношение $y_1 = y_0 + hf(x_0,y_0)$. Считая теперь точку (x_1, y_1) начальной и повторяя все предыдущие рассуждения, получим значение y_2 в узле x_2.

Переход к произвольным индексам дает формулу метода Эйлера:

$$y_{k+1} = y_k + hf(x_k, y_k) \qquad (4.2)$$

Погрешность метода Эйлера.

На каждом шаге метода Эйлера допускается *локальная* погрешность по отношению к точному решению, график которого проходит через крайнюю левую точку отрезка. Геометрически локальная погрешность изображается отрезком CD на первом шаге, C'D' на втором и т. д. Кроме того, на каждом шаге, начиная со второго, накапливается

глобальная погрешность представляющая собой разность межу численным решением и точным решением исходной начальной задачи (а не локальной). Глобальная погрешность на втором шаге изображена отрезком C'E' на рис.4.1.

Локальная ошибка на каждом шаге выражается соотношением $\varepsilon_k^h = \frac{y''(\xi)}{2}h^2$, где $\xi \in [x_{k-1}, x_k]$. Глобальная погрешность метода Эйлера $\varepsilon_{LM}^h = Ch$ в окрестности h=0 ведет себя как линейная функция, и, следовательно, метод Эйлера имеет первый порядок точности относительно шага h.

Модификации метода Эйлера.

Неявный метод Эйлера

Если на правой границе интервала использовать точное значение производной от решения (т.е. тангенса угла наклона касательной), то получается неявный метод Эйлера первого порядка точности.

$$\begin{aligned} \tilde{y}_{k+1} &= y_k + hf(x_k, y_k) \\ y_{k+1} &= y_k + \frac{h(f(x_k, y_k) + f(x_{k+1}, \tilde{y}_{k+1})}{2} \\ x_{k+1} &= x_k + h \end{aligned} \qquad (4.4)$$

$$y_{k+1} = y_k + hf(x_{k+1}, y_{k+1}) \qquad (4.3)$$

В общем случае нелинейное относительно y_{k+1} уравнение (4.3) численно решается с помощью одного из методов раздела 2, например, методом Ньютона или его модификациями.

Метод Эйлера - Коши

В данном методе на каждом интервале расчет проводится в два этапа. На первом (этап прогноза) определяется приближенное решение на правом конце интервала по методу Эйлера, на втором (этап коррекции) уточняется значение решения на правом конце с использованием полусуммы тангенсов углов наклона на концах интервала

Этот метод имеет второй порядок точности.

Неявный метод Эйлера - Коши

Если на правой границе интервала использовать точное значение производной к решению (т. е. тангенса угла наклона касательной), то получается неявный метод Эйлера-Коши (метод трапеций) второго порядка точности.

$$y_{k-1} = y_k + \frac{h(f(x_k, y_k) + f(x_{k-1}, y_{k+1}))}{2} \qquad (4.5)$$
$$x_{k-1} = x_k + h$$

Метод Эйлера-Коши с итерационной обработкой

Комбинация (4.3), (4.4) и (4.5) дает метод формально второго порядка точности, но более точного в смысле абсолютной величины погрешности приближенного решения, чем исходные методы.

$$y_{k+1}^{(0)} = y_k + hf(x_k, y_k)$$
$$y_{k+1}^{(i)} = y_k + \frac{h(f(x_k, y_k) + f(x_{k+1}, y_{k-1}^{(i-1)}))}{2} \qquad (4.6)$$
$$x_{k+1} = x_k + h$$

В формуле (6) правые верхние индексы в круглых скобках обозначают номер итерации, при этом начальное приближение y[+[определяется по методу Эйлера.

Метод Эйлера-Коши с итерационной обработкой представляет собой реализацию метода простой итерации для решения нелинейного уравнения (5) в неявном методе Эйлера. Выполнять простые итерации до полной сходимости нет смысла, поэтому рекомендуется выполнять 3-4 итерации.

Первый улучшенный метод Эйлера

Данный метод использует расчет приближенного значения производной от решения в точке на середине расчетного интервала. Значение производной в середине получают применением явного метода Эйлера на половинном шаге по x.

$$y_{k+1/2} = y_k + \frac{h}{2} f(x_k, y_k)$$
$$y_{k-1} = y_k + hf(x_{k-1/2}, y_{k-1/2}) \qquad (4.7)$$
$$x_{k+1} = x_k + h$$
$$x_{k+1/2} = x_k + h/2$$

Данная модификация метода Эйлера имеет второй порядок точности.

Методы Рунге-Кутты

$$K_1^k = hf(x_k, y_k)$$
$$K_2^k = hf(x_k + \frac{1}{3}h, y_k + \frac{1}{3}K_1^k)$$
$$K_3^k = hf(x_k + \frac{2}{3}h, y_k + \frac{2}{3}K_2^k)$$

$$y_{k+1} = y_k + \Delta y_k$$
$$\Delta y_k = \frac{1}{4}(K_1^k + 3K_3^k) \qquad (4.9)$$

Все рассмотренные выше явные методы являются вариантами методов Рунге-Кутты. Семейство явных методов Рунге-Кутты *p-го* порядка записывается в виде совокупности формул:

$$y_{k-1} = y_k + \Delta y_k$$
$$\Delta y_k = \sum_{i=1}^{p} c_i K_i^k \qquad (4.8)$$

$$K_i^k = hf(x_k + a_i h, y_k + h\sum_{j=1}^{i-1} b_{ij} K_j^k)$$
$$i = 2,3.....p$$

Параметры a_i, b_{ij}, c_i подбираются так, чтобы значение y_{k-1}, рассчитанное по соотношению (4.8) совпадало со значением разложения в точке x_{k-1} точного решения в ряд Тейлора с погрешностью $O(h^{p-1})$

Метод Рунге-Кутты четвертого порядка точности

Метод Рунге-Кутты четвертого порядка

$$(p = 4, a_1 = 0, a_2 = \frac{1}{2}, a_3 = \frac{1}{2}, a_4 = 1, b_{21} = \frac{1}{2}, b_{31} = 0, b_{32} = \frac{1}{2}, b_{41} = 0, b_{42} = 0, b_{43} = \frac{1}{2}, c_1 = \frac{1}{6},$$

$$c_2 = \frac{1}{3}, c_3 = \frac{1}{3}, c_3 = \frac{1}{6})$$

является одним из самых широко используемых методов для решения Задачи Коши:

$$y_{k-1} = y_k + \Delta y_k$$

$$\Delta y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2K_3^k + K_4^k) \qquad (4.10)$$

$$K_1^k = hf(x_k, y_k)$$

$$K_2^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k)$$

$$K_3^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k)$$

$$K_4^k = hf(x_k + h, y_k + K_3^k)$$

Контроль точности на каждом шаге *h*.

Основным способом контроля точности получаемого численного решения при решении задачи Коши является методы основанные на принципе Рунге-Ромберга- Ричардсона.

Пусть y^h решение задачи Коши (1) полученое методом Рунге-Куттыp - го порядка точности с шагом *h* в точке x+2h . Пусть y^2 решение той же задачи в точке *x+2h,* полученное тем же методом, но с шагом 2h . Тогда выражение

$$\bar{y} = y^h + \frac{y^h - y^{2h}}{2^p - 1} \qquad (4.11)$$

аппроксимирует точное решение в точке $x + 2h$ $y(x + 2h)$ с $p + 1$-ым порядком.

Второе слагаемое в выражении (4.11) оценивает главный член в погрешности решения y^h, то есть $R^h = \frac{y^h - y^{2h}}{2^p - 1}$. Контроль точности может быть организован следующим образом. Выбирается значение шага *h* и дважды рассчитывается решение в точке *x+2h,* один раз с шагом h, другой раз с шагом 2h. Рассчитывается величина R^h и сравнивается с заданной точностью *s* . Если величина R^h меньше *s*, то можно продолжать вычисления с тем же шагом, в противном случае необходимо вернуться к решению в точке *x,* уменьшить шаг *h* и повторить вычисления.

Вычислительная стоимость такого контроля точности достаточно велика, особенно для многостадийных методов. Поэтому можно использовать более грубый способ контроля правильности выбора шага *h* . В случае метода Рунге-Кутты четвертого порядка точности следует на каждом шаге *h* рассчитывать параметр

$$E^K = \left| \frac{\text{K}_2 - \kappa^{\text{к}}_3}{\kappa\kappa - \kappa\,\kappa} \right| \qquad (4.12)$$

Если величина $0^к$ порядка нескольких сотых единицы, то расчет продолжается с тем же шагом, если $0^к$ больше одной десятой, то шаг следует уменьшить, если же $0^к$ меньше одной сотой, то шаг можно увеличить.

Таким образом с помощью определения величин $в^к$ или R^h можно организовать алгоритм выбора шага h для явного метода Рунге-Кутты.

4.1.3. Решение задачи Коши для системы обыкновенных дифференциальных уравнений.

Рассматривается задача Коши для системы дифференциальных уравнений первого

$$\begin{cases} y_1' = f_1(x, y_1, y_2, ..., y_n) \\ y_2' = f_2(x, y_1, y_2, ..., y_n) \\ ..... \\ y_n' = f_n(x, y_1, y_2, ..., y_n) \end{cases} \qquad (4.13)$$

$$y_1(x_0) = y_{01}$$
$$y_2(x_0) = y_{02}$$
$$.....$$
$$y_n(x_0) = y_{0n}$$

Система (4.13) в более компактном виде записывается в векторной форме

$$\begin{aligned} &\overline{y}' = \overline{F}(x, \overline{y}) \\ &\overline{y}(x_0) = \overline{y}_0 \end{aligned} \qquad (4.14)$$

Здесь $\overline{y}(x) = (y_1, y_2, ..., y_n)^T$ - вектор столбец неизвестных функций, $\overline{F} = (f_1, f_2, ..., f_n)^T$ - вектор функция правых частей.

порядка разрешенных относительно производной

К векторному дифференциальному уравнению (4.14) можно применить все методы рассмотренные выше в данном разделе (благодаря линейной структуре всех рассмотренных методов). При этом в формулах (4.2)-(4.14) все величины векторные кроме переменной x и шага h.

Рассмотрим задачу Коши для системы двух ОДУ первого порядка, где уравнения записаны в развернутом виде

$$\begin{cases} y' = f(x, y, z) \\ z' = g(x, y, z) \end{cases} \qquad (4.15)$$
$$y(x_0) = y_0$$
$$z(x_0) = z_0$$

Формулы метода Рунге-Кутты 4-го порядка точности для решения (4.15) следующие:

$$y_{k+1} = y_k + \Delta y_k$$
$$z_{k+1} = z_k + \Delta z_k$$
$$\Delta y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2K_3^k + K_4^k) \qquad (4.16)$$
$$\Delta z_k = \frac{1}{6}(L_1^k + 2L_2^k + 2L_3^k + L_4^k)$$

$$K_1^k = hf(x_k, y_k, z_k)$$
$$L_1^k = hg(x_k, y_k, z_k)$$
$$K_2^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k, z_k + \frac{1}{2}L_1^k)$$
$$L_2^k = hg(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k, z_k + \frac{1}{2}L_1^k)$$
$$K_3^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k, z_k + \frac{1}{2}L_2^k)$$
$$L_3^k = hg(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k, z_k + \frac{1}{2}L_2^k)$$
$$K_4^k = hf(x_k + h, y_k + K_3^k, z_k + L_3^k)$$
$$L_4^k = hg(x_k + h, y_k + K_3^k, z_k + L_3^k)$$

Контроль правильности выбора шага *h* в случае использования метода Рунге- Кутты четвертого порядка точности для системы (4.15) может быть организован с помощью вычисления на каждом шаге *h* параметров

$$\theta_1^k = \left| \frac{K_2^k - K_3^k}{K_1^k - K_2^k} \right|; \qquad (4.17)$$
$$\theta_2^k = \left| \frac{L_2^k - L_3^k}{L_1^k - L_2^k} \right|$$

Если величины Ok (i=1,2) порядка нескольких сотых единицы, то расчет продолжается с тем же шагом, если больше одной десятой, то шаг следует уменьшить, если же меньше одной сотой, то шаг можно увеличить.

Исходный код на языке Python

1 Численные методы решения задачи Коши

```
#! /usr/bin/python3.1
# -*- coding: utf-8 -*-

from math import sqrt,sin,cos,tan,pi,log
from copy import copy, deepcopy
from functools import reduce
eps = 0.000001

def EulerMethod( f, g, segment, h, y0, z0):
     """Solving ordinary differential equations"""
     p0 = (segment[0], y0, z0)
     (x1,y1,z1) = p0
     func = [p0]

     while( p0[0] < segment[1] + eps):
          x1 += h
          y1 += h * f( p0 )
          z1 += h * g( p0 )
          p0 = (x1,y1,z1)
          func.append( p0 )
     return func


def RungeKutta( f, g, segment, h, y0, z0):
     """Iterative methods for the approximation of solutions of ODE"""
     x0 = segment[0]
     func = [ (x0,y0,z0) ]

     while(x0 < segment[1] + eps):
          k1 = h * f((x0, y0, z0))
          l1 = h * g((x0, y0, z0))
          k2 = h * f((x0 + h/2, y0 + k1/2, z0 + l1/2));
          l2 = h * g((x0 + h/2, y0 + k1/2, z0 + l1/2));
          k3 = h * f((x0 + h/2, y0 + k2/2, z0 + l2/2));
          l3 = h * g((x0 + h/2, y0 + k2/2, z0 + l2/2));
          k4 = h * f((x0 + h, y0 + k3, z0 + l3));
          l4 = h * g((x0 + h, y0 + k3, z0 + l3));
          dy = (k1 + 2 * k2 + 2 * k3 + k4) / 6;
          dz = (l1 + 2 * l2 + 2 * l3 + l4) / 6;
          x0 += h
          y0 += dy
          z0 += dz
          func.append((x0,y0,z0))
     return func


def AdamsMethod( f, g, segment, h, y0, z0 ):
     x0 = segment[0]
     func = RungeKutta(f, g, (x0, x0 + 2*h + eps), h, y0, z0 )

     Xk = [ zz[0]for zz in func]
     Fk = [ f(p) for p in func]
     Gk = [ g(p) for p in func]
     (x0,y0,z0) = func[-1]

     while( x0 < segment[1] + eps):
          x0 += h
          y0 += h*(55*Fk[-1] - 59 * Fk[-2]  + 37 * Fk[-3] - 9 * Fk[-4]) / 24
          z0 += h*(55*Gk[-1] - 59 * Gk[-2]  + 37 * Gk[-3] - 9 * Gk[-4]) / 24
```

```
            Xk.append( x0 )
            Fk.append( f((x0,y0,z0)) )
            Gk.append( g((x0,y0,z0)) )
            func.append( (x0,y0,z0) )
#      func = [ pk for pk in zip(Xk,Fk,Gk) ]
      return func

def f(x):
      return x*x*x + 3*x + 1

def f1(p):
      return p[2]

def f2(p):
      (x,y,z) = p
      return y * cos(x)*cos(x) - z * tan(x)

def pr(X):
      for xx in X :
            print("%.2f\t"%xx, end = '')
      print()


def main():
      Ek = EulerMethod(f1, f2, (0, 1) , 0.1, 2, 0)
      Rk = RungeKutta(f1, f2, (0, 1) , 0.1, 2, 0)
      Ak = AdamsMethod(f1, f2, (0, 1) , 0.1, 2, 0)

      print("table function for Euler Method ")
      [ pr( xx ) for xx in Ek[-7:] ]
      print("table function for Runge Kutta ")
      [ pr( xx ) for xx in Rk[-7:] ]
      print("table function for Adams Method ")
      [ pr( xx ) for xx in Ak[-7:] ]
      print("x     y     z")

      return 0

main()
```

2 Численные методы решение краевой задачи для ОДУ

```
#! /usr/bin/python3.1
# -*- coding: utf-8 -*-

from math import sqrt,sin,cos,tan,pi,log,exp
from copy import copy, deepcopy
from functools import reduce
eps = 0.0001

def RungeKutta( f, g, segment, h, y0, z0):
      """Iterative methods for the approximation of solutions of ODE"""
      x0 = segment[0]
      func = [ (x0,y0,z0) ]

      while(x0 < segment[1] - eps):
            k1 = h * f((x0, y0, z0))
            l1 = h * g((x0, y0, z0))
            k2 = h * f((x0 + h/2, y0 + k1/2, z0 + l1/2));
            l2 = h * g((x0 + h/2, y0 + k1/2, z0 + l1/2));
            k3 = h * f((x0 + h/2, y0 + k2/2, z0 + l2/2));
```

```
          l3 = h * g((x0 + h/2, y0 + k2/2, z0 + l2/2));
          k4 = h * f((x0 + h, y0 + k3, z0 + l3));
          l4 = h * g((x0 + h, y0 + k3, z0 + l3));
          dy = (k1 + 2 * k2 + 2 * k3 + k4) / 6;
          dz = (l1 + 2 * l2 + 2 * l3 + l4) / 6;
          x0 += h
          y0 += dy
          z0 += dz
          func.append((x0,y0,z0))
     return func


class Tridiagonal_Matrix:
     def __init__(self, f = ""):
          self.a = []
          self.b = []
          self.c = []
          self.d = []
          self.n = 0

     def solve(self):
          """Method progonki"""
          a = self.a
          b = self.b
          c = self.c
          d = self.d
          n = self.n

          P = []
          Q = []
          P.append(-c[0]/b[0])
          Q.append( d[0]/b[0])

          for i in range(1, n):
               P.append( -c[i] / (b[i]+a[i]*P[i-1]) )
               Q.append( (d[i] - a[i]*Q[i-1]) / (b[i] + a[i]*P[i-1]) )

          x = [0]*n
          x[n-1] = Q[n-1]
          for i in range(n-2, -1, -1):
               x[i] = P[i]*x[i+1] + Q[i]

          return x


def f(x):
     return x*x*x + 3*x + 1

def y(p):
     return p[1]


def f1(p):
     (x,y,z) = p
     return z

def f2(p):
     (x,y,z) = p
     return exp(x) + sin(y)

def pr(X):
     for xx in X :
          print("%.2f\t"%xx, end = '')
     print()
```

```
def shoting(f, g, segment, h, y0, y1 ):
     eta = []
     eta.append(1.0)
     eta.append(0.8)
     Rk0 = Rk1 = RungeKutta(f, g, (0, 1) , 0.1, 1, eta[-2]) [-1]

     delta1 = 1
     count = 0
     while (delta1 > eps)and(count < 9000):
          count+=1
          Rk0 = Rk1
          Rk1 = RungeKutta(f, g, segment , h, y0, eta[-1]) [-1]

          delta1 = abs( y(Rk1) - y1)
          delta0 = abs( y(Rk0) - y1)

          new_eta = eta[-1]-(y(Rk1)-y1)*(eta[-1] - eta[-2])/(y(Rk1) - y(Rk0))
          eta.append(new_eta)


     return RungeKutta(f1, f2, segment , h, y0, eta[-1])


def finite_diff(X, p, q, f, h, n = 5):
     ya = 1.0

     #~ May be +h*h !!!
     Tridiag = Tridiagonal_Matrix("")
     a = [ 0.0 ]
     b = [ -2 + h*h * q(X[1]) ]
     c = [ 1 - p(X[1])*h/2 ]
     d = [ h*h * f(X[1]) - 1 - p(X[1])*h/2.0*ya ]

     for k in range(2, n):
          a.append( 1 + p(X[k])*h/2.0 )
          b.append( -2 + h*h * q(X[k]) )
          c.append( 1 - p(X[k])*h/2 )
          d.append( h*h * f(X[k]) )

     #~ for first border
     #~ a.append( 1 - p(X[-1])*h/2.0 )
     #~ b.append( -2 + h*h * q(X[-1]) )
     #~ c.append( 0.0 )
     #~ d.append( h*h * f(X[k]) - 1 - p(X[-1])*h/2 *yb )

     a.append( 5)
     b.append(-7)
     c.append( 0)
     d.append( 0)

     #~ pr(a)
     #~ pr(b)
     #~ pr(c)
     #~ pr(d)

     Tridiag.a = a
     Tridiag.b = b
     Tridiag.c = c
     Tridiag.d = d
     Tridiag.n = len(a)


     y = [ya] + Tridiag.solve()
     return y
```

```
def RRR(f1, f2, k, p):
     # k = h1/h2, p - interpolation level
     return (f1 - f2) / (k**p - 1)


def main():

     p = lambda x: -x
     q = lambda x: -1
     f = lambda x: 0
     h = 0.2


     X = [ h*i for i in range(6)]
     X2 = [ h*i/2 for i in range(11)]
     X4 = [ h*i/4 for i in range(21)]
     print( "X = ", X)

     Sh = shoting(f1, f2, [0, 1] , 0.1, 1.0, 2.0)
     print("--Shoot--")
     print(Sh[-1])


     y1 = finite_diff(X, p, q, f, h, 5)
     y2 = finite_diff(X2, p, q, f, h/2, 10)
     y4 = finite_diff(X4, p, q, f, h/4, 20)


     print("---finite_diff---")
     print("X =")
     pr(X)
     print("y1 =")
     pr(y1)
     print("y2 =")
     pr(y2)
     print("y4 =")
     pr(y4[::2])


     print("eps:", RRR(y4[-1], y2[-1], 1/2, 2) )

#   y'' + p(x)y' + q(x) = 0
#     y(a) = y0; y(b) = y1;

     return 0

main()
```

Протокол тестирования

1 Численные методы решения задачи Коши

```
oleg@debian:~/lab4_release$ python3.1 lab4_1.py
table function for Euler Method
0.50                                2.19   0.90
0.60                                2.28   1.02
0.70                                2.38   1.11
0.80                                2.50   1.15
0.90                                2.61   1.15
1.00                                2.73   1.11
1.10                                2.84   1.02
table function for Runge Kutta
0.50                                2.23   0.87
0.60                                2.33   0.98
0.70                                2.43   1.06
0.80                                2.54   1.09
0.90                                2.65   1.08
1.00                                2.75   1.02
1.10                                2.85   0.92
table function for Adams Method
0.50                                2.23   0.87
0.60                                2.33   0.98
0.70                                2.43   1.05
0.80                                2.54   1.09
0.90                                2.65   1.08
1.00                                2.75   1.02
1.10                                2.85   0.92
x                                   y      z
```

2 Численные методы решение краевой задачи для ОДУ

```
oleg@debian:~/lab4_release$ python3.1 lab4_2.py
X =  [0.0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]
--Shoot--
(0.9999999999999999, 1.9999999997893136, 2.473875822251543)
---finite_diff---
X =
0.00                                0.20   0.40   0.60   0.80   1.00
y1 =
1.00                                0.77   0.58   0.43   0.31   0.22
y2 =
1.00                                0.87   0.75   0.64   0.54   0.46   0.37   0.30
                                    0.24   0.18   0.13
y4 =
1.00                                0.87   0.74   0.63   0.52   0.43   0.34   0.26
                                    0.19   0.13   0.07
eps: 0.0778786693048
```

*Погрешность для h = 0.1, h = 0.05

Выводы

Дифференциальные уравнения применяются при рассмотрении большинства физических процессов и решении почти любых прикладных задач. Это объясняет огромную важность алгоритмов для их вычисления, которые были изучены в данной лабораторной работе.