Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой Проект по курсу
«Численные Методы»

Студент: К. О. Вахрамян
Преподаватель: Д. Л. Ревизников
Группа: М8О-306Б
Дата:
Оценка:
Подпись:

Москва, 2021

«Решение краевых задач для нелинейных дифференциальных уравнений методом конечных разностей»

Описание

Рассмотрим постановку краевых задач для нелинейно ОДУ 2-го порядка с граничными условиями различных родов. Пусть на отрезке $x \in [a, b]$ определена дважды непрерывно дифференцируемая функция $y(x)$, поведение которой описывается нелинейным неоднородным ОДУ 2-го порядка. Принципиаль ным отличием краевой задачи от задачи Коши для ОДУ является задание дополнительных (краевых или граничных) условий более чем в одной точке независимой переменной (в задаче Коши дополнительные условия задаются в одной точке, называемой начальной).
Если на границах $x = a$ и $x = b$ заданы значения искомой функции $y(a), y(b)$, то такие условия называются граничными условиями первого рода.
Если на границах заданы линейные комбинации искомой функции и ее первой производной, то такие условия называются граничными условиями третьего рода

$$\begin{cases} F(y'', y', y, x) = 0, \\ a_0 y(a) + a_1 y(a)' = \alpha, \\ b_0 y(b) + b_1 y(b)' = \beta \end{cases}$$

Поскольку ОДУ описывает поведение функции $y(x)$ внутри расчётной области $x \in (a, b)$, то производные 1-го и 2-го порядка порядков можно аппроксимировать с помощью отношения центральных разностей со 2-м порядком аппроксимации:

$$y'_i = \frac{y_{i+1} - y_{i-1}}{2h} + O(h^2), i = 1, .., n-1;$$

$$y''_i = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + O(h^2), i = 1, .., n-1;$$

Подставим эту аппроксимацию в исходную функцию получим:

$$F\left(\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \frac{y_{i+1} - y_{i-1}}{2h}, y_i, x_i\right), i = 1, .., n-1$$

Т.о. получаем систему нелинейных уравнений.

Граничные условия имеют вид после аппроксимации:

$$y_0 = \frac{\alpha h}{a_0 h - a_1} - \frac{a_1}{a_0 h - a_1} y_1;$$

$$y_n = \frac{\beta h}{b_0 h + b_1} + \frac{b_1}{b_0 h + b_1} y_{n-1}$$

Для решения системы нелинейных уравнений воспользуемся методом Ньютона. Общая формула:

$$y^{(k+1)} = y^{(k)} - J^{-1}(y^{(k)})F(y^{(k)})$$

Где J - трехдиагональная матрица Якоби. Определим ее следующим образом:

$$J(y) = \begin{bmatrix} B(y_1) & C(y_1) & 0 & 0 & \dots & 0 \\ A(y_2) & B(y_2) & C(y_2) & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \dots & 0 \\ 0 & \dots & 0 & A(y_{n-2}) & B(y_{n-2}) & C(y_{n-2}) \\ 0 & \dots & \dots & 0 & A(y_{n-2}) & B(y_{n-2}) \end{bmatrix}$$

Где

$$A(y_i) = \frac{\partial F}{\partial y_{i-1}} = \frac{\left(F\left(\frac{y_{i+1}-2y_i+y_i}{h^2}, \frac{y_{i+1}-y_i}{2h}, y_i, x_i\right) - F\left(\frac{y_{i+1}-2y_i+y_{i-2}}{h^2}, \frac{y_{i+1}-y_{i-2}}{2h}, y_i, x_i\right)\right)}{2h}$$

$$B(y_i) = \frac{\partial F}{\partial y_i} = \frac{\left(F\left(\frac{y_{i+1}-2y_{i+1}+y_{i-1}}{h^2}, \frac{y_{i+1}-y_{i-1}}{2h}, y_{i+1}, x_i\right) - F\left(\frac{y_{i+1}-2y_{i-1}+y_{i-1}}{h^2}, \frac{y_{i+1}-y_{i-1}}{2h}, y_{i-1}, x_i\right)\right)}{2h}$$

$$C(y_i) = \frac{\partial F}{\partial y_{i+1}} = \frac{\left(F\left(\frac{y_{i+2}-2y_i+y_{i-1}}{h^2}, \frac{y_{i+2}-y_{i-1}}{2h}, y_i, x_i\right) - F\left(\frac{y_i-2y_i+y_{i-1}}{h^2}, \frac{y_i-y_{i-1}}{2h}, y_i, x_i\right)\right)}{2h}$$

Условия окончания итерационного процесса: $||y^{(k+1)} - y^{(k)}|| < \varepsilon = 0.01$.
Начальное приближение определяется программно.

Примеры работы программы

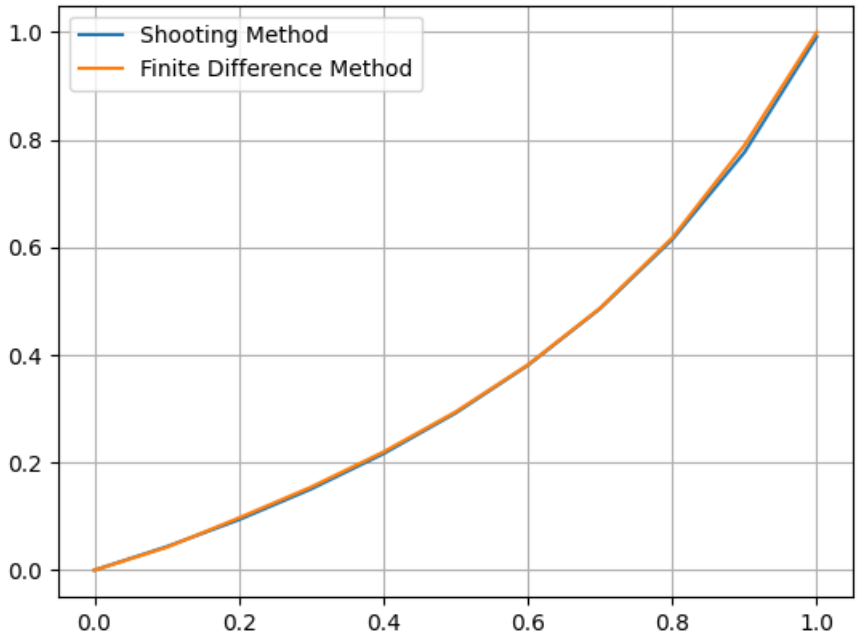
Для проверки корректности работы алгоритма исполузется метод стрельбы.

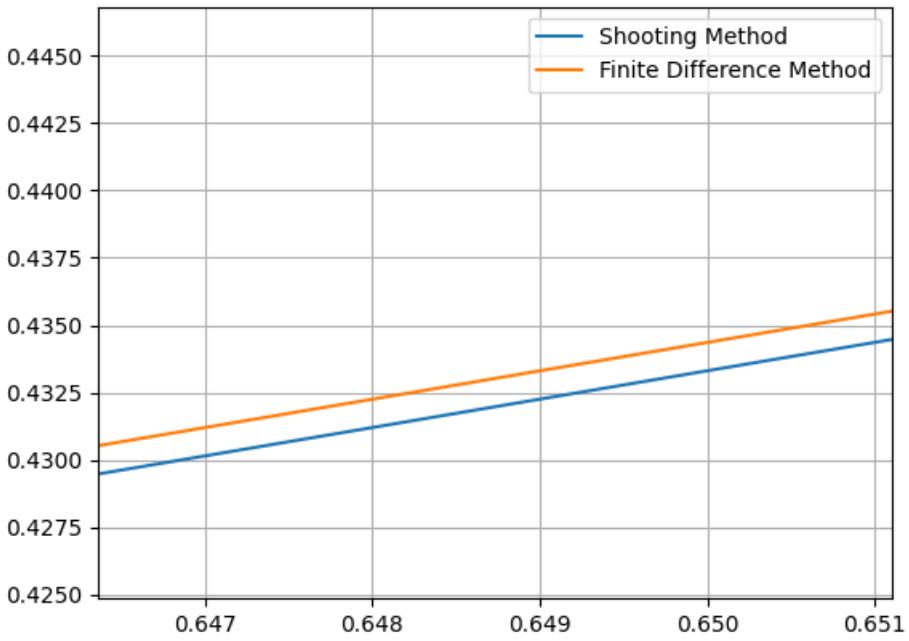
Пример 1.

$$F(y'', y', y, x) = y'' - (y')^2 - y' - y + x$$
$$y(0) = 0$$
$$y(1) = 1$$

```
Finite Difference Method:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 0.0423924 0.0971522 0.15456 0.219645 0.294223 0.381658 0.487061 0.617833
0.789857 1


Shooting Method:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 0.0436652 0.0936419 0.150776 0.216302 0.292024 0.380596 0.486021 0.614577
0.776705 0.991385
```



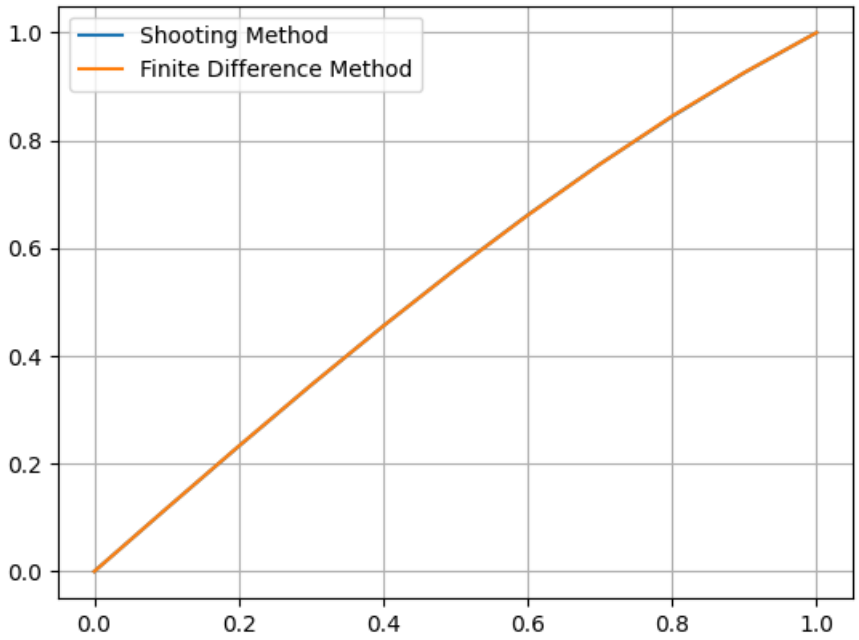

Shooting Method
Finite Difference Method
0.4450
0.4425
0.4400
0.4375
0.4350
0.4325
0.4300
0.4275
0.4250
0.647
0.648
0.649
0.650
0.651

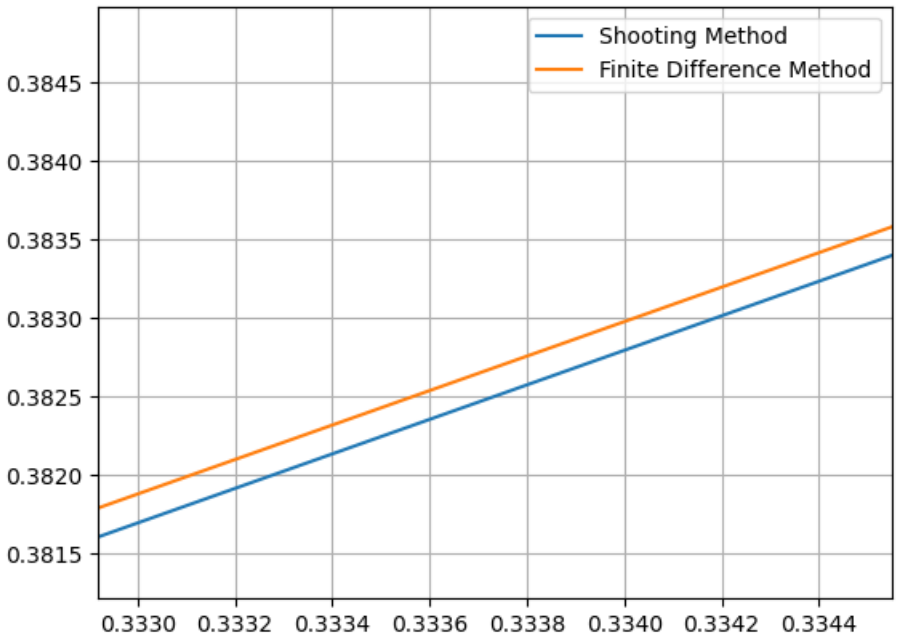
Пример 2.

$$F(y'', y', y, x) = y''y' + (y')^2 x = 0$$
$$y(0) = 0$$
$$y(1) = 1$$

```
Finite Difference Method:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 0.116842 0.232459 0.345667 0.455403 0.560735 0.660929 0.75564 0.845166 0.925954
1


Shooting Method:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 0.116679 0.232198 0.345432 0.455322 0.560906 0.661346 0.755941 0.844145 0.925574
1
```



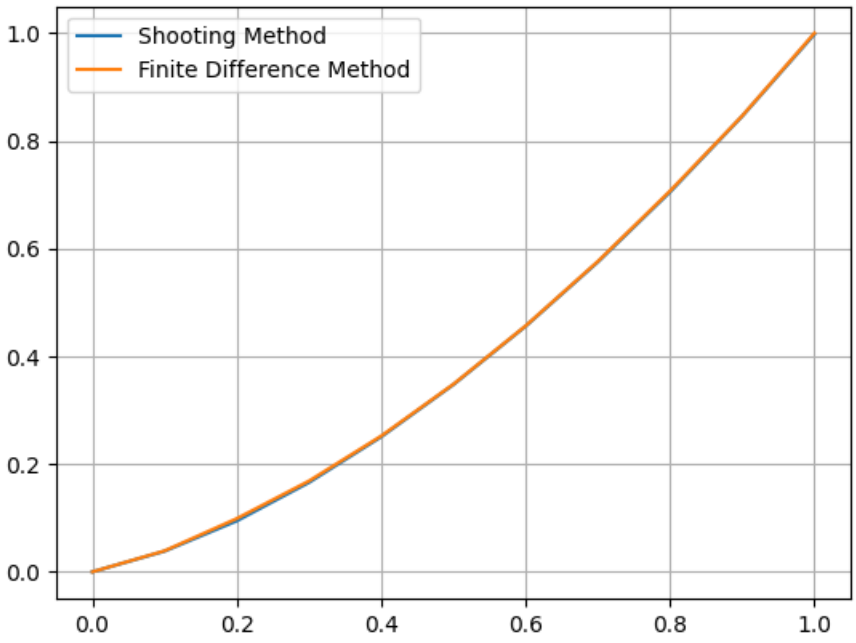

Shooting Method
Finite Difference Method
0.3845
0.3840
0.3835
0.3830
0.3825
0.3820
0.3815
0.3330
0.3332
0.3334
0.3336
0.3338
0.3340
0.3342
0.3344

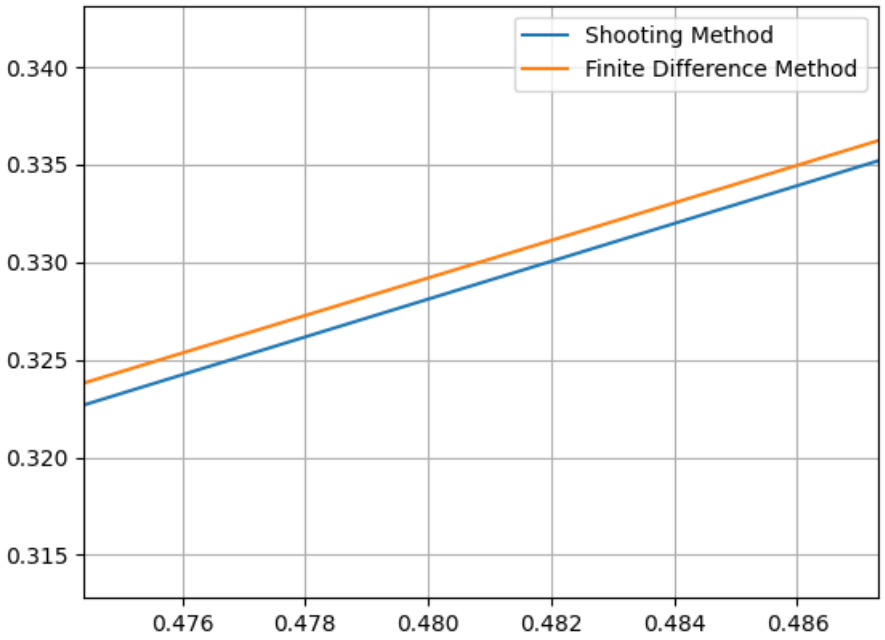
Пример 3.

$$F(y'', y', y, x) = y'' + \log(y') - e^y + x^2$$
$$y(0) = 0$$
$$y(1) = 1$$

```
Finite Difference Method:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 0.0390763 0.0989769 0.168725 0.252222 0.348438 0.456657 0.576321 0.707072
0.846563 1


Shooting Method:
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 0.0383552 0.0945493 0.165857 0.250565 0.347494 0.455833 0.575074 0.705017
0.845811 0.998041
```




Shooting Method
Finite Difference Method
0.340
0.335
0.330
0.325
0.320
0.315
0.476
0.478
0.480
0.482
0.484
0.486

Исходый код

```cpp
#include "calculation.hpp"


ld F(ld d2y, ld dy, ld y, ld x) {
    return d2y + log(dy) - std::exp(y) + x * x;
}


ld A(std::vector<ld> const & y, ld x, size_t i, ld h) {
    ld part1 = F((y[i + 1] - y[i]) / std::pow(h, 2.), (y[i + 1] - y[i]) / (2. * h), y[i
        ], x);
    ld part2 = F((y[i + 1] - 2. * y[i] + y[i - 2]) / std::pow(h, 2.), (y[i + 1] - y[i -
         2]) / (2. * h), y[i], x);
    return (part1 - part2) / (2. * h);
}

ld B(std::vector<ld> const & y, ld x, size_t i, ld h) {
    ld part1 = F((y[i - 1] - y[i + 1]) / std::pow(h, 2.), (y[i + 1] - y[i - 1]) / (2. *
         h), y[i + 1], x);
    ld part2 = F((y[i + 1] - y[i - 1]) / std::pow(h, 2.), (y[i + 1] - y[i - 1]) / (2. *
         h), y[i - 1], x);
    return (part1 - part2) / (2. * h);
}

ld C(std::vector<ld> const & y, ld x, size_t i, ld h) {
    ld part1 = F((y[i + 2] - 2. * y[i] + y[i - 1]) / std::pow(h, 2.), (y[i + 2] - y[i -
         1]) / (2. * h), y[i], x);
    ld part2 = F((y[i - 1] - y[i]) / std::pow(h, 2.), (y[i] - y[i - 1]) / (2. * h), y[i
        ], x);
    return (part1 - part2) / (2. * h);
}


TMatrix Jacobi_Matrix_Inverse(std::vector<ld> const & y, ld h, ld x_0, ld x_1) {
    size_t n = y.size() - 2;
    TMatrix J(n, n, 0.);
    J[0][0] = B(y, x_0 + h, 1, h);
    J[0][1] = C(y, x_0 + h, 1, h);

    for (size_t i = 1; i != n - 1; i++) {
        J[i][i - 1] = A(y, x_0 + (i + 1) * h, i + 1, h);
        J[i][i] = B(y, x_0 + (i + 1) * h, i + 1, h);
        J[i][i + 1] = C(y, x_0 + (i + 1) * h, i + 1, h);
    }
    J[n - 1][n - 2] = A(y, x_0 + n * h, n, h);
    J[n - 1][n - 1] = B(y, x_0 + n * h, n, h);

```

```
42      auto [L, U, P] = J.LUdecomposition();
43
44      return LU_Inverse_Matrix(L, U, P);
45  }
46
47  TMatrix F_column(std::vector<ld> const & y, ld h, ld x_0, ld x_1) {
48      size_t n = y.size() - 2;
49      TMatrix F_c(n, (size_t)1);
50      for (size_t k = 0; k != n; k++) {
51          size_t i = k + 1;
52          F_c[k][0] = F((y[i + 1] - 2. * y[i] + y[i - 1]) / std::pow(h, 2), (y[i + 1] - y
                [i - 1]) / (2. * h), y[i], x_0 + i * h);
53      }
54      return F_c;
55  }
56
57
58
59  std::vector<ld> Newton_Method(ld h, ld x_0, ld x_1, ld a_0, ld a_1, ld alpha, ld b_0,
        ld b_1, ld beta) {
60      ld eps = 0.01;
61      size_t n = (size_t)((x_1 - x_0) / h);
62
63      std::vector<ld> y(n + 1);
64
65      //initial approximation
66      ld s = (x_1 + x_0) / (n + 1);
67      y[1] = h;
68      for (size_t i = 2; i != n; i++)
69          y[i] = y[i - 1] + s;
70      y[0] = alpha * h / (a_0 * h - a_1) - a_1 / (a_0 * h - a_1) * y[1];
71      y[n] = beta * h /(b_0 * h + b_1) + b_1 / (b_0 * h + b_1) * y[n - 1];
72
73
74      TMatrix y_column(n - 1, (size_t)1);
75          for (size_t i = 0; i != n - 1; i++)
76              y_column[i][0] = y[i + 1];
77
78      while (true) {
79
80
81          TMatrix J_inv = Jacobi_Matrix_Inverse(y, h, x_0, x_1);
82          TMatrix F_c = F_column(y, h, x_0, x_1);
83
84          //Iteration
85          TMatrix y_next = y_column - J_inv * F_c;
86
87
88
```

```
89          for (size_t i = 0; i != n - 1; i ++)
90              y[i + 1] = y_next[i][0];
91          y[0] = alpha * h / (a_0 * h - a_1) - a_1 / (a_0 * h - a_1) * y[1];
92          y[n] = beta * h / (b_0 * h + b_1) + b_1 / (b_0 * h + b_1) * y[n - 1];
93
94          if (TMatrix(y_next - y_column).Norm() < eps)
95              break;
96
97          y_column = y_next;
98
99      }
100     return y;
101
102 }
103
104 std::tuple<std::vector<ld>, std::vector<ld>> Finite_Difference_Method(ld h, ld x_0, ld
        x_1, ld a_0, ld a_1, ld alpha, ld b_0, ld b_1, ld beta, std::ostream* log) {
105     size_t n = (size_t)((x_1 - x_0) / h);
106     std::vector<ld> x(n + 1);
107     x[0] = x_0;
108     for (size_t i = 1; i != n + 1; i++)
109         x[i] = x[i - 1] + h;
110     std::vector<ld> y = Newton_Method(h, x_0, x_1, a_0, a_1, alpha, b_0, b_1, beta);
111
112
113     if (log) {
114         for (auto a : x)
115             *log << a << ' ';
116         *log << '\n';
117         for (auto a : y)
118             *log << a << ' ';
119
120     }
121
122     std::cout << "Finite Difference Method is over\n";
123     return std::make_tuple(x, y);
124 }
125
126
127
128 ld f1(ld x, ld y, ld z) {
129     return z;
130 }
131
132 ld f2(ld x, ld y, ld z) {
133     return -log(z) + std::exp(y) - x * x;
134 }
135
136
```

```
137 std::tuple<std::vector<ld>, std::vector<ld>, std::vector<ld>> Runge_Kutta_Method(ld h,
        ld x_0, ld x_1, ld y_0, ld z_0, std::ostream* log = nullptr) {
138     size_t n = (size_t)((x_1 - x_0) / h);
139     std::vector<ld> x(n + 1);
140     std::vector<ld> y(n + 1);
141     std::vector<ld> z(n + 1);
142 
143     x[0] = x_0;
144     y[0] = y_0;
145     z[0] = z_0;
146 
147 
148     for (size_t i = 0; i != n; i++) {
149         ld k_1 = h * f1(x[i], y[i], z[i]);
150         ld l_1 = h * f2(x[i], y[i], z[i]);
151         ld k_2 = h * f1(x[i] + h / 2., y[i] + k_1 / 2., z[i] + l_1 / 2.);
152         ld l_2 = h * f2(x[i] + h / 2., y[i] + k_1 / 2., z[i] + l_1 / 2.);
153         ld k_3 = h * f1(x[i] + h / 2., y[i] + k_2 / 2., z[i] + l_2 / 2.);
154         ld l_3 = h * f2(x[i] + h / 2., y[i] + k_2 / 2., z[i] + l_2 / 2.);
155         ld k_4 = h * f1(x[i] + h, y[i] + k_3, z[i] + l_3);
156         ld l_4 = h * f2(x[i] + h, y[i] + k_3, z[i] + l_3);
157 
158         ld delta_y = (k_1 + (2. * k_2) + (2. * k_3) + k_4) / 6.;
159         ld delta_z = (l_1 + (2. * l_2) + (2. * l_3) + l_4) / 6.;
160 
161         x[i + 1] = x[i] + h;
162         y[i + 1] = y[i] + delta_y;
163         z[i + 1] = z[i] + delta_z;
164 
165     }
166 
167     if (log) {
168 
169         for (auto a : x)
170             *log << a << ' ';
171         *log << '\n';
172 
173         for (auto a : y)
174             *log << a << ' ';
175         *log << '\n';
176     }
177 
178     return std::make_tuple(x, y, z);
179 }
180 
181 
182 ld Phi (ld b_0, ld b_1, ld y_s, ld z_s, ld beta) {
183     return b_0 * y_s + b_1 * z_s - beta;
184 }
```

```
185
186
187 std::tuple<std::vector<ld>, std::vector<ld>> Shooting_Method(ld h, ld x_0, ld x_1, ld
        a_0, ld a_1, ld alpha, ld b_0, ld b_1, ld beta, std::ostream* log) {
188     std::vector<ld> x;
189     std::vector<ld> y;
190
191     ld eps = 0.01;
192     int n = (int)((x_1 - x_0) / h);
193     std::vector<ld> s;
194     ld y_0, z_0;
195
196     ld C_0, C_1;
197     if (a_0 == 0.) {
198         C_0 = -1. / a_1;
199         C_1 = 0.;
200     } else {
201         C_0 = 0.;
202         C_1 = -1. / a_0;
203     }
204
205     s.push_back((x_1 - x_0) / 2.);
206     s.push_back(s[0] / 2.);
207
208     std::vector<ld> y_s;
209     std::vector<ld> z_s;
210
211
212     y_0 = a_1 * s[0] - C_1 * alpha;
213     z_0 = a_0 * s[0] - C_0 * alpha;
214     auto ans1 = Runge_Kutta_Method(h, x_0, x_1, y_0, z_0);
215         y_s.push_back(std::get<1>(ans1)[n]);
216         z_s.push_back(std::get<2>(ans1)[n]);
217
218     y_0 = a_1 * s[1] - C_1 * alpha;
219     z_0 = a_0 * s[1] - C_0 * alpha;
220     auto ans2 = Runge_Kutta_Method(h, x_0, x_1, y_0, z_0);
221         y_s.push_back(std::get<1>(ans2)[n]);
222         z_s.push_back(std::get<2>(ans2)[n]);
223
224
225
226
227
228     size_t i = 2;
229     while (true) {
230         ld current_s = s[i - 1] - (s[i - 1] - s[i - 2]) / (Phi(b_0, b_1, y_s[i - 1],
            z_s[i - 1], beta) - Phi(b_0, b_1, y_s[i - 2], z_s[i - 2], beta)) * Phi(b_0,
             b_1, y_s[i - 1], z_s[i - 1], beta);
```

```
231         s.push_back(current_s);
232         y_0 = a_1 * s[i] - C_1 * alpha;
233         z_0 = a_0 * s[i] - C_0 * alpha;
234         auto ans = Runge_Kutta_Method(h, x_0, x_1, y_0, z_0);
235         y_s.push_back(std::get<1>(ans)[n]);
236         z_s.push_back(std::get<2>(ans)[n]);
237 
238         if (std::abs(Phi(b_0, b_1, y_s[i], z_s[i], beta)) < eps) {
239             x = std::get<0>(ans);
240             y = std::get<1>(ans);
241             break;
242         }
243 
244         i++;
245     }
246 
247     if (log) {
248         for (auto a : x)
249             *log << a << ' ';
250         *log << '\n';
251         for (auto a : y)
252             *log << a << ' ';
253 
254     }
255     std::cout << "Shooting Method is over\n";
256     return std::make_tuple(x, y);
257 }
```

Выводы

Решение краевых задач с нелинейными ОДУ в общем виде - задача непростая, так, например WolframAlpha не решил ни одной задачи в общем виде, однако численно, с достаточно малым шагом мы получаем точное решение. Часто я сталкивался с проблемой, что решение методом конечных разностей расходилось, это обусловленно тем, что, решая систему нелинейных уравнений методом Ньютона, необходимо выбрать начальное приблежение, которое нам не иззвестно. Метод стрельбы мне показался более надежным, ведь давал решение, даже когда таковое отсутствовало при решением методом Конечных разностей.