Университет ИТМО

Вычислительная математика

Лабораторная работа №6

Численное решение обыкновенных дифференциальных уравнений

ФИО студента: Готовко Алексей Владимирович
Направление подготовки: 09.03.04 (СППО)
Учебная группа: P32101
ФИО преподавателей:
Малышева Т.А.
Рыбаков С.Д.

Санкт-Петербург
2023г.

1 Цель работы

Решить задачу интерполяции, найти значения функции при заданных значениях аргумента, отличных от узловых точек.

2 Программная реализация задачи

```
def step_euler(f: Function, h: float, x: float, y: float, twice=False) -> float:
    for i in range(2 if twice else 1):
        y += h * f.value_at(x, y)
        x += h
    return y


def step_runge_kutta(f: Function, h: float, x: float, y: float, twice=False) -> float:
    for i in range(2 if twice else 1):
        k1 = h * f.value_at(x, y)
        k2 = h * f.value_at(x + h / 2, y + k1 / 2)
        k3 = h * f.value_at(x + h / 2, y + k2 / 2)
        k4 = h * f.value_at(x + h, y + k3)
        y += (k1 + 2 * k2 + 2 * k3 + k4) / 6
        x += h
    return y


def milne_predict(f: Function, h: float, x_arr: list[float], y_arr: list[float]) -> float:
    return 4 * h / 3 * (2 * f.value_at(x_arr[-1], y_arr[-1]) - f.value_at(x_arr[-2], y_arr[-2])
                        + 2 * f.value_at(x_arr[-3], y_arr[-3])) + y_arr[-4]


def milne_correct(f: Function,
                  h: float,
                  x_arr: list[float],
                  y_arr: list[float],
                  pred_y: float) -> float:
    return h / 3 * (f.value_at(x_arr[-1] + h / 2, pred_y) + 4 * f.value_at(x_arr[-1], y_arr[-1])
                    + f.value_at(x_arr[-2], y_arr[-2])) + y_arr[-2]


class DiffEqSolver:
    def __init__(self,
                 functions: tuple[Function, Function, Function, Function],
                 h: float,
                 y0: float,
                 x0: float,
                 xn: float,
                 eps: float) -> None:
        self.functions = functions
        self.h = h
        self.y0 = y0
        self.x0 = x0
        self.xn = xn
        self.eps = eps

    def solve(self, func_idx: int, method_name: SolutionMethod) -> Solution:
        if method_name == SolutionMethod.MILNE:
            return self._solve_milne(func_idx)
        params = get_method_params(method_name)
        return self._solve_single_step(func_idx, params["order"], params["function"])

```

```
    def _solve_single_step(self, func_idx: int, order: int, step_method) -> Solution:
        f: Function
        f = self.functions[func_idx]

        cur_h = self.h

        x = self.x0
        y = self.y0

        exact_y_arr = [y]
        x_arr = [x]
        y_arr = [y]
        f_arr = []

        while x <= self.xn:
            y_halved_step = step_method(f, cur_h / 2, x, y, True)

            f_val = f.value_at(x, y)
            y = step_method(f, cur_h, x, y)
            x += cur_h

            if abs(y - y_halved_step) / (2 ** order - 1) > self.eps and cur_h > STEP_MIN:
                exact_y_arr = [f.exact_value_at(self.x0)]
                x_arr = [self.x0]
                y_arr = [self.y0]
                f_arr = []
                y = self.y0
                x = self.x0
                cur_h /= 2
                continue

            x_arr.append(x)
            y_arr.append(y)
            f_arr.append(f_val)
            exact_y_arr.append(f.exact_value_at(x))

        f_arr.append(0)
        return Solution(len(exact_y_arr), x_arr, y_arr, exact_y_arr, f_arr)

    def _solve_milne(self, func_idx: int) -> Solution:
        f: Function
        f = self.functions[func_idx]

        x = self.x0
        y = self.y0

        exact_y_arr = [f.exact_value_at(x)]
        x_arr = [x]
        y_arr = [y]

        for i in range(3):
            k1 = self.h * f.value_at(x, y)
            k2 = self.h * f.value_at(x + self.h / 2, y + k1 / 2)
            k3 = self.h * f.value_at(x + self.h / 2, y + k2 / 2)
            k4 = self.h * f.value_at(x + self.h, y + k3)

            y += (k1 + 2 * k2 + 2 * k3 + k4) / 6
            x += self.h
            exact_y_arr.append(f.exact_value_at(x))
            y_arr.append(y)
            x_arr.append(x)
```

```
115
116         while x <= self.xn:
117             eps = max([abs(y_arr[i] - exact_y_arr[i]) for i in range(len(y_arr))])
118
119             pred_y = milne_predict(f, self.h, x_arr, y_arr)
120             corr_y = milne_correct(f, self.h, x_arr, y_arr, pred_y)
121             while abs(pred_y - corr_y) > eps > STEP_MIN:
122                 pred_y = corr_y
123                 corr_y = milne_correct(f, self.h, x_arr, y_arr, pred_y)
124
125             x += self.h
126             exact_y_arr.append(f.exact_value_at(x))
127             y_arr.append(corr_y)
128             x_arr.append(x)
129
130         return Solution(len(exact_y_arr), x_arr, y_arr, exact_y_arr)
131
132
```

Полный код программы доступен по ссылке.

3 Результат работы программы

```
Select function
[1] y' = x - y
[2] y' = y + (1 + x) * y^2
[3] y' = x^2 - 2 * y
[4] y' = 3 * x^2 * y + x^2 * e^(x^3)
>> 3
Specify [x0, xn] interval and y0 = y(x0) value (format: 'y0 x0 xn'):
>> -1 1 7
Specify step (h) value:
>> 0.1
Specify accuracy (epsilon) value:
>> 0.2

Euler's method
```

i	x	y	f(x, y)	Exact y
0	1.00000	-1.00000	3.00000	-1.00000
1	1.10000	-0.70000	2.61000	-0.71841
2	1.20000	-0.43900	2.31800	-0.46790
3	1.30000	-0.20720	2.10440	-0.24101
4	1.40000	0.00324	1.95352	-0.03166
5	1.50000	0.19859	1.85282	0.16515
6	1.60000	0.38387	1.79225	0.35351
7	1.70000	0.56310	1.76380	0.53675
8	1.80000	0.73948	1.76104	0.71763
9	1.90000	0.91558	1.77883	0.89838
10	2.00000	1.09347	1.81307	1.08083
11	2.10000	1.27477	1.86045	1.26650
12	2.20000	1.46082	1.91836	1.45660
13	2.30000	1.65265	1.98469	1.65216
14	2.40000	1.85112	2.05775	1.85399
15	2.50000	2.05690	2.13620	2.06277
16	2.60000	2.27052	2.21896	2.27905
17	2.70000	2.49242	2.30517	2.50328
18	2.80000	2.72293	2.39414	2.73585
19	2.90000	2.96235	2.48531	2.97704
20	3.00000	3.21088	2.57825	3.22711
21	3.10000	3.46870	2.67260	3.48626
22	3.20000	3.73596	2.76808	3.75465
23	3.30000	4.01277	2.86446	4.03244
24	3.40000	4.29922	2.96157	4.31971
25	3.50000	4.59537	3.05926	4.61658
26	3.60000	4.90130	3.15740	4.92310
27	3.70000	5.21704	3.25592	5.23935
28	3.80000	5.54263	3.35474	5.56538
29	3.90000	5.87810	3.45379	5.90122
30	4.00000	6.22348	3.55303	6.24690
31	4.10000	6.57879	3.65243	6.60246
32	4.20000	6.94403	3.75194	6.96792
33	4.30000	7.31922	3.85155	7.34330
34	4.40000	7.70438	3.95124	7.72861
35	4.50000	8.09950	4.05099	8.12386
36	4.60000	8.50460	4.15080	8.52907
37	4.70000	8.91968	4.25064	8.94424
38	4.80000	9.34475	4.35051	9.36937
39	4.90000	9.77980	4.45041	9.80449
40	5.00000	10.22484	4.55033	10.24958

41	5.10000	10.67987	4.65026	10.70466
42	5.20000	11.14490	4.75021	11.16972
43	5.30000	11.61992	4.85017	11.64477
44	5.40000	12.10493	4.95013	12.12981
45	5.50000	12.59995	5.05011	12.62485
46	5.60000	13.10496	5.15009	13.12987
47	5.70000	13.61997	5.25007	13.64490
48	5.80000	14.14497	5.35005	14.16992
49	5.90000	14.67998	5.45004	14.70493
50	6.00000	15.22498	5.55003	15.24994
51	6.10000	15.77999	5.65003	15.80495
52	6.20000	16.34499	5.75002	16.36996
53	6.30000	16.91999	5.85002	16.94497
54	6.40000	17.50499	5.95001	17.52997
55	6.50000	18.09999	6.05001	18.12498
56	6.60000	18.70500	6.15001	18.72998
57	6.70000	19.32000	6.25001	19.34499
58	6.80000	19.94500	6.35001	19.96999
59	6.90000	20.58000	6.45000	20.60499
60	7.00000	21.22500	6.55000	21.24999
61	7.10000	21.88000	0.00000	21.90499

Runge-Kutta's method

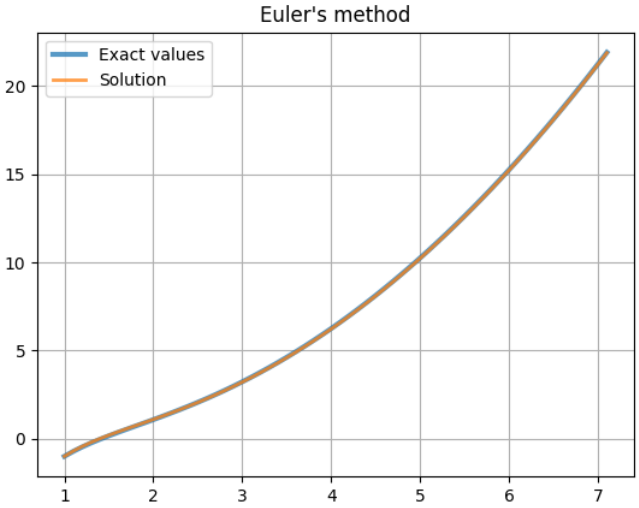
i	x	y	f(x, y)	Exact y
0	1.00000	-1.00000	3.00000	-1.00000
1	1.10000	-0.71842	2.64683	-0.71841
2	1.20000	-0.46790	2.37581	-0.46790
3	1.30000	-0.24102	2.17204	-0.24101
4	1.40000	-0.03167	2.02333	-0.03166
5	1.50000	0.16515	1.91971	0.16515
6	1.60000	0.35350	1.85299	0.35351
7	1.70000	0.53675	1.81650	0.53675
8	1.80000	0.71763	1.80475	0.71763
9	1.90000	0.89837	1.81325	0.89838
10	2.00000	1.08083	1.83834	1.08083
11	2.10000	1.26650	1.87701	1.26650
12	2.20000	1.45660	1.92680	1.45660
13	2.30000	1.65216	1.98568	1.65216
14	2.40000	1.85399	2.05202	1.85399
15	2.50000	2.06277	2.12446	2.06277
16	2.60000	2.27905	2.20190	2.27905
17	2.70000	2.50329	2.28343	2.50328
18	2.80000	2.73585	2.36830	2.73585
19	2.90000	2.97704	2.45592	2.97704
20	3.00000	3.22711	2.54578	3.22711
21	3.10000	3.48626	2.63748	3.48626
22	3.20000	3.75466	2.73069	3.75465
23	3.30000	4.03244	2.82512	4.03244
24	3.40000	4.31972	2.92057	4.31971
25	3.50000	4.61658	3.01684	4.61658
26	3.60000	4.92311	3.11378	4.92310
27	3.70000	5.23936	3.21128	5.23935
28	3.80000	5.56538	3.30924	5.56538
29	3.90000	5.90122	3.40756	5.90122
30	4.00000	6.24691	3.50619	6.24690
31	4.10000	6.60247	3.60506	6.60246
32	4.20000	6.96793	3.70415	6.96792

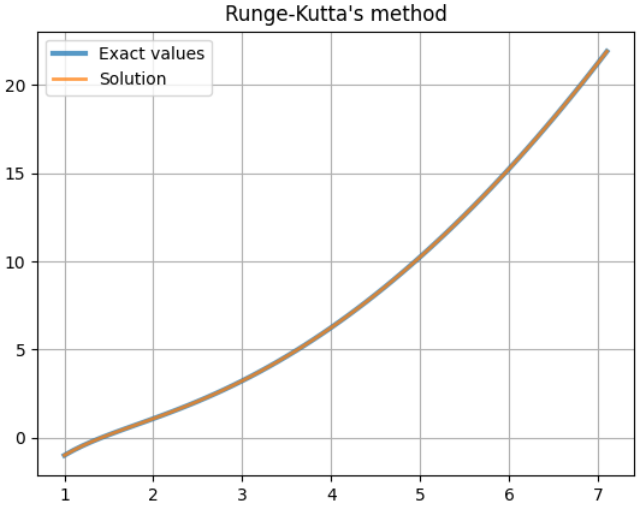
33	4.30000	7.34330	3.80339	7.34330
34	4.40000	7.72861	3.90278	7.72861
35	4.50000	8.12386	4.00227	8.12386
36	4.60000	8.52907	4.10186	8.52907
37	4.70000	8.94424	4.20152	8.94424
38	4.80000	9.36938	4.30124	9.36937
39	4.90000	9.80449	4.40102	9.80449
40	5.00000	10.24959	4.50083	10.24958
41	5.10000	10.70466	4.60068	10.70466
42	5.20000	11.16972	4.70055	11.16972
43	5.30000	11.64477	4.80045	11.64477
44	5.40000	12.12982	4.90037	12.12981
45	5.50000	12.62485	5.00030	12.62485
46	5.60000	13.12988	5.10024	13.12987
47	5.70000	13.64490	5.20020	13.64490
48	5.80000	14.16992	5.30016	14.16992
49	5.90000	14.70494	5.40013	14.70493
50	6.00000	15.24995	5.50010	15.24994
51	6.10000	15.80496	5.60008	15.80495
52	6.20000	16.36997	5.70007	16.36996
53	6.30000	16.94497	5.80005	16.94497
54	6.40000	17.52998	5.90004	17.52997
55	6.50000	18.12498	6.00003	18.12498
56	6.60000	18.72999	6.10002	18.72998
57	6.70000	19.34499	6.20002	19.34499
58	6.80000	19.96999	6.30001	19.96999
59	6.90000	20.60500	6.40001	20.60499
60	7.00000	21.25000	6.50001	21.24999
61	7.10000	21.90500	0.00000	21.90499

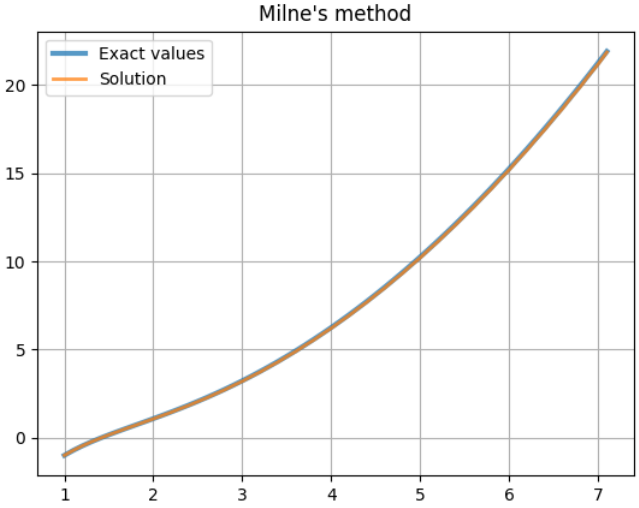
Milne's method

i	x	y	Exact y
0	1.00000	-1.00000	-1.00000
1	1.10000	-0.71842	-0.71841
2	1.20000	-0.46790	-0.46790
3	1.30000	-0.24102	-0.24101
4	1.40000	-0.03624	-0.03166
5	1.50000	0.16171	0.16515
6	1.60000	0.34548	0.35351
7	1.70000	0.53057	0.53675
8	1.80000	0.70665	0.71763
9	1.90000	0.88991	0.89838
10	2.00000	1.06721	1.08083
11	2.10000	1.25608	1.26650
12	2.20000	1.43973	1.45660
13	2.30000	1.64023	1.65216
14	2.40000	1.83391	1.85399
15	2.50000	2.04970	2.06277
16	2.60000	2.25581	2.27905
17	2.70000	2.48785	2.50328
18	2.80000	2.70976	2.73585
19	2.90000	2.96119	2.97704
20	3.00000	3.19797	3.22711
21	3.10000	3.46833	3.48626
22	3.20000	3.72271	3.75465
23	3.30000	4.01251	4.03244
24	3.40000	4.28513	4.31971

25	3.50000	4.59480	4.61658
26	3.60000	4.88598	4.92310
27	3.70000	5.21585	5.23935
28	3.80000	5.52577	5.56538
29	3.90000	5.87606	5.90122
30	4.00000	6.20484	6.24690
31	4.10000	6.57573	6.60246
32	4.20000	6.92342	6.96792
33	4.30000	7.31504	7.34330
34	4.40000	7.68166	7.72861
35	4.50000	8.09412	8.12386
36	4.60000	8.47967	8.52907
37	4.70000	8.91305	8.94424
38	4.80000	9.31751	9.36937
39	4.90000	9.77190	9.80449
40	5.00000	10.19523	10.24958
41	5.10000	10.67070	10.70466
42	5.20000	11.11286	11.16972
43	5.30000	11.60948	11.64477
44	5.40000	12.07041	12.12981
45	5.50000	12.58826	12.62485
46	5.60000	13.06789	13.12987
47	5.70000	13.60705	13.64490
48	5.80000	14.10533	14.16992
49	5.90000	14.66586	14.70493
50	6.00000	15.18271	15.24994
51	6.10000	15.76469	15.80495
52	6.20000	16.30004	16.36996
53	6.30000	16.90357	16.94497
54	6.40000	17.45733	17.52997
55	6.50000	18.08248	18.12498
56	6.60000	18.65457	18.72998
57	6.70000	19.30143	19.34499
58	6.80000	19.89176	19.96999
59	6.90000	20.56043	20.60499
60	7.00000	21.16889	21.24999
61	7.10000	21.85948	21.90499


Euler's method
Exact values
Solution
20
15
10
5
0
1
2
3
4
5
6
7


Runge-Kutta's method
Exact values
Solution
20
15
10
5
0
1
2
3
4
5
6
7


Milne's method
Exact values
Solution
20
15
10
5
0
1
2
3
4
5
6
7

4 Вывод

Для наиболее оптимального распределения человеческих ресурсов и поддержания количества нервных клеток в головном мозгу, настоятельно рекомендуется использовать готовые библиотеки, содержащие наиболее эффективные реализации алгоритмов, вместо самостоятельной реализации оных.

Иными словами, вообще лучше зачиллиться и не изобретать велосипед.