Project1

Shuang Hu

2021 年 4 月 28 日

1 Problem Statement

Consider the following **dynamic system**:

$$\begin{cases} u_1^{'} = u_4 \\ u_2^{'} = u_5 \\ u_3^{'} = u_6 \\ u_4^{'} = 2u_5 + u_1 - \dfrac{\mu(u_1+\mu-1)}{(u_2^2+u_3^2+(u_1+\mu-1)^2)^{\frac{3}{2}}} \\ u_5^{'} = -2u_4 + u_2 - \dfrac{\mu u_2}{(u_2^2+u_3^2+(u_1+\mu-1)^2)^{\frac{3}{2}}} - \dfrac{(1-\mu)u_2}{(u_2^2+u_3^2+(u_1+\mu)^2)^{\frac{3}{2}}} \\ u_6^{'} = -\dfrac{\mu u_3}{(u_2^2+u_3^2+(u_1+\mu-1)^2)^{\frac{3}{2}}} - \dfrac{(1-\mu)u_3}{(u_2^2+u_3^2+(u_1+\mu)^2)^{\frac{3}{2}}} \end{cases}$$

, my assignment is to create a C++ package to achieve

- Adams-Bashforce methods with precision p=1,2,3,4
- Adams-Moulton methods with precision p=2,3,4,5
- BDFs with precision p=1,2,3,4
- the classical Runge-Kutta method

Then test my program with the following two IVP cases:

Actual Case 1:

$$(u_1(0), u_2(0), u_3(0), u_4(0), u_5(0), u_6(0)) = (0.994, 0, 0, 0, -2.0015851063790825224, 0) \tag{1}$$

with period $T_1 = 17.06521656015796$

Actual Case 2:

$$(u_1(0), u_2(0), u_3(0), u_4(0), u_5(0), u_6(0)) = (0.87978, 0, 0, 0, -0.3797, 0) \tag{2}$$

with period $T_2 = 19.14045706162071$

2 Results And Some Straight-Forward Comparison

2.1 Results

The following two figures show the trajectories related to the above two test cases.

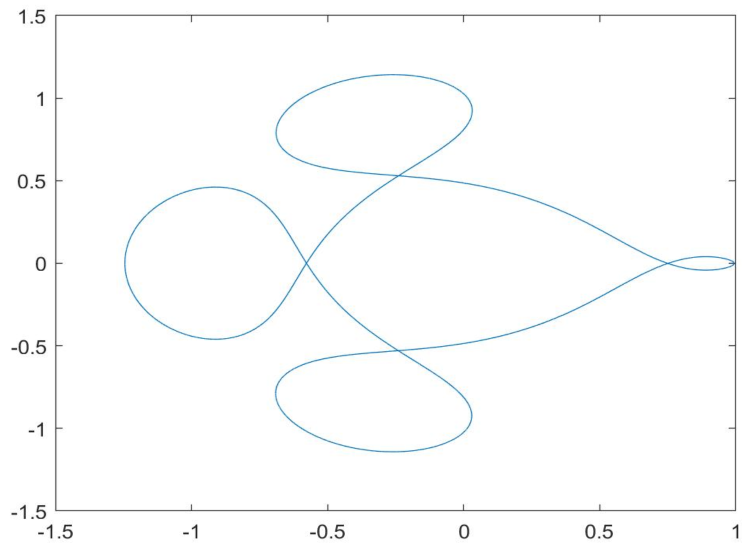

图 1: Result for actual case 1:Use Runge-Kutta method with 20000 steps.

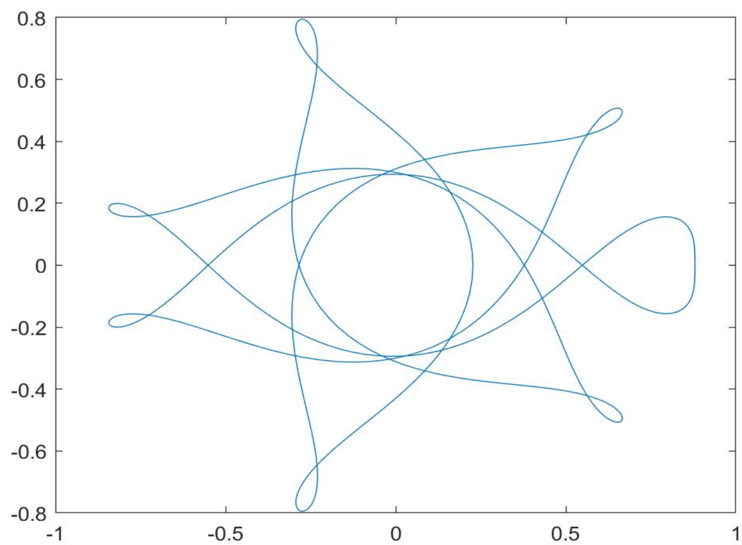

图 2: Result for actual case 2:Use Runge-Kutta method with 1000 steps.

2.2 Straight-Forward Comparison

Remark:All the ODE-solver algorithms are based on equidistant partition!So I use the parameter "Step number" instead of "Time step".The relationship:$k = \frac{T}{n}$, k:time step,n:step number,T:the period.

In this section,I make some comparison by straight-forward naked eye comparison.Rule:I will show the minimal stepnum to generate the figure similar to the correct ones.

Method	Precision	Stepnumber
Adam-Bashforce	1	>1000000
	2	near 250000
	3	near 60000
	4	near 50000
Adam-Moulton	2	near 125000
	3	near 40000
	4	near 25000
	5	near 18000
BDF	1	>1000000
	2	near 200000
	3	near 40000
	4	near 28000
RungeKutta	4	near 18000

表 1: The stepnum for actual case 1

Method	Precision	Stepnumber
Adam-Bashforce	1	>1000000
	2	near 8000
	3	near 6000
	4	near 2000
Adam-Moulton	2	near 5000
	3	near 3000
	4	near 1500
	5	near 1000
BDF	1	>1000000
	2	near 5000
	3	near 3500
	4	near 1350
RungeKutta	4	near 750

表 2: The stepnum for actual case 2

3 Analysis:Precision and Relative Error

3.1 Analyze the Relative Error

The definition of Relative Error comes as:

$$E = \frac{||I_E - I_A||_2}{||I_E||_2}$$

where I_E denotes the actual solution,and I_A denotes the solution approximated by the algorithm.Now, I will use actual problem (1) to do the test.

Here is the result.Timestep N gets 100000.

Method	Precision	E(N)	E(2N)	E(4N)
Adam-Bashforce	1	0.861674	0.829466	0.809348
	2	0.947447	0.784596	0.550436
	3	0.129373	0.012894	0.002843
	4	0.12506	0.019344	0.006423
Adam-Moulton	2	0.586927	0.322605	0.113165
	3	0.067277	0.029633	0.016107
	4	0.041897	0.026324	0.015617
	5	0.050341	0.026847	0.015557
BDF	1	0.952406	0.966053	0.94306
	2	0.735703	0.579029	0.312331
	3	0.091682	0.031460	0.015582
	4	0.036067	0.014236	0.013965
RungeKutta	4	0.208489	0.010726	0.000697

3.2 Richardson Extrapolation Method

Now, I will use **Richardson Extrapolation Method** to generate the convergent order.

In general problems, we can't get some information about the actual solution.So we need **Richardson extrapolation method.** For this problem, the equivalence comes as:

$$p \approx log_2\left(\frac{||U(h) - U(\frac{h}{4})||_2}{||U(\frac{h}{2}) - U(\frac{h}{4})||_2} - 1\right)$$

where $U(h)$ means the algorithm solution using time step h.The proof of this equation is shown in Levesque <Finite Differential Methods for Ordinary and Partial Differential Equations>, page 257.

In actual work,we can't set h too big or too small.If h is too big, the identity may not true, even the numerical form maybe unstable.If h is too small, it means we have to do something like "divide a small number", which is bad-conditioned.

Here are the results for the extrapolation method test:

Method	Precision	Stepnumber(N)	test p
Adam-Bashforce	1	80000	0.70
	2	16000	1.72
	3	8000	2.92
	4	2000	3.69
Adam-Moulton	2	1000	1.89
	3	850	2.79
	4	700	4.24
	5	180	5.21
BDF	1	50000	2.28(a bug)
	2	16000	1.78
	3	8000	2.71
	4	1500	4.68
RungeKutta	4	100	4.34

4 A contest:Forward Euler(1-order Adam-Bashforce) versus Classical Runge-Kutta

Forward Euler algorithm and Classical Runge-Kutta algorithm are two famous algorithms in ODE solver.Now, I will compare these two algorithms in two means.

4.1 Compare the result trajectory

First of all,I will compare the 6000-steps result by RungeKutta with the 24000-steps result by ForwardEuler.I use actual case 1 to make a test.

Here are the results:

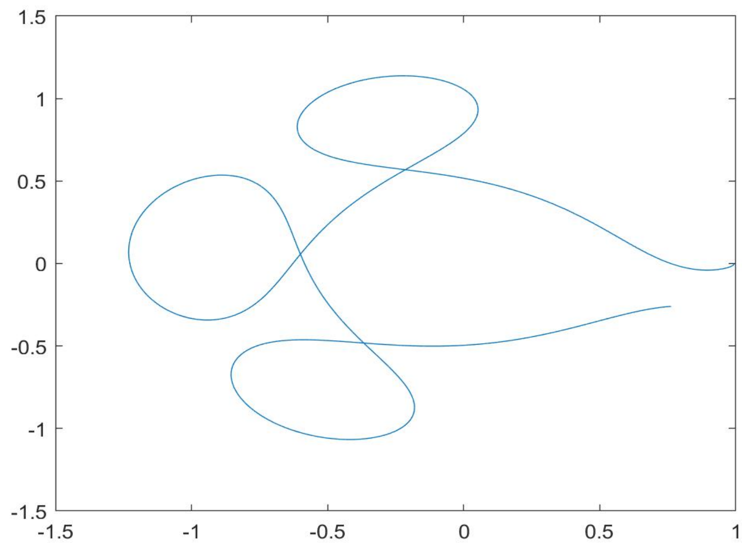

图 3: Result for actual case 1:Use Runge-Kutta method with 6000 steps.

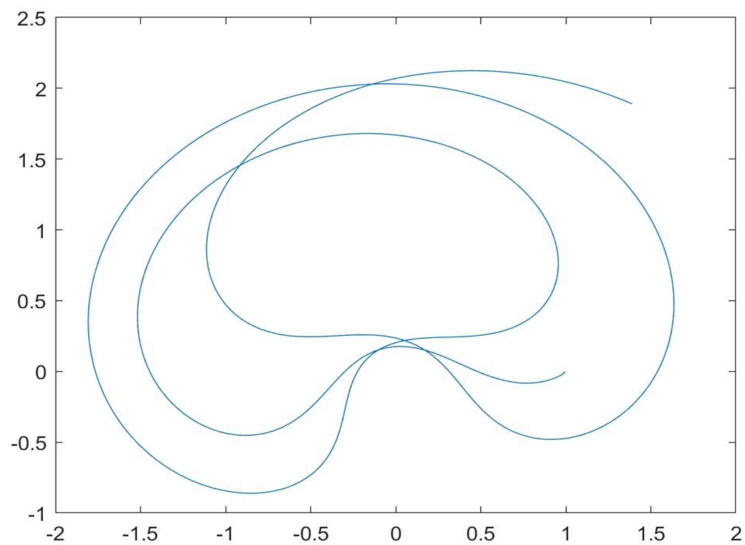

图 4: Result for actual case 1:Use Forward-Euler method with 24000 steps.

It's clear that:although the number of steps is different, Runge-Kutta algorithm performs far better than Forward-Euler algorithm.

4.2 Compare the total CPU time

Now, if we hope our result to achieve an error of 10^{-3} based on the maxnorm of the solution error, which algorithm will triamph?

```
Method          Order   Te                      Initial_value                           stepnum
AdamBashforce   1       17.06521656015796       0.994 0 0 0 -2.0015851063790825224 0    2000000
```

图 5: Input File for ForwardEuler

```
hsmath@ubuntu:~/nde2021/Project1$ ./ODESolver
position:[0.991696,0.0483944,0],velocity:[0.581179,-0.518757,0]
1.48283
Time:4.44241 seconds.
```

图 6: Result for the Forward Euler

What a pity!Although I set the step number to 2000000, the Forward Euler algorithm can't achieve my demand!The CPU time comes to 4.44241 seconds.

How about Runge-Kutta Method?

```
Method       Order   Te                      Initial_value                           stepnum
RungeKutta   4       17.06521656015796       0.994 0 0 0 -2.0015851063790825224 0    90000
```

图 7: Input File for ForwardEuler

```
hsmath@ubuntu:~/nde2021/Project1$ ./ODESolver
position:[0.993998,-6.07041e-06,0],velocity:[-0.000988999,-2.00188,0]
0.000988999
Time:0.981779 seconds.
```

图 8: Input File for ForwardEuler

We can see:Runge-Kutta Method achieves my demand perfectly!And its CPU time is only 0.981779 seconds.

So, Runge-Kutta method must be the winner in this problem.

5 Some shortcomings and doubts

My package has the following shortcomings:

- Only use trivial time step.
- The data structure isn't always same.In "Point" class, I use 3-dimension array to store the position and velocity, but when it comes to implicit difference scheme, I use Eigen::Matrix.
- I didn't use template, which makes my program hard to reuse.

In additional, during the process of testing and coding, I have some doubts about this project.

- First-order algorithm usually convergent to a wrong answer,why?
- The performance of Runge-Kutta method seems greater than LMM algorithms with precision = 4, why?
- First of all, I try to generate the initial-value of LMM by Euler's Method, it happens that the precision of LMM reduce rapidly.What's more, the algorithm with precision = 2 performs better than p = 3, and better than p = 4.After I have done the Runge-Kutta algorithm, I try to generate the initial-value with RK algorithm, then everything goes right.What happens?

6 Technical Details

See the file "Readme.md"