Technion – Israel Institute of Technology

HW4

Numerical Methods

019003

Alon Spinner	305184335	alonspinner@gmail.com
Oren Elmakis	311265516	orenelmakis@gmail.com

December 6, 2021

Question 1 – Implement your own ODE solver

(a) We were asked to implement Runge-Kutta-4

The RK-4 method can be written as such:

$$y_{k+1} = y_k + \sum_{i=1}^{4} \alpha_i k_i = y_k + \frac{h}{6}(k_0 + 2k_1 + 2k_2 + k_3)$$

Where:

$$k_0 = f\left(t_k, y_k + k_0 \cdot \frac{h}{2}\right)$$

$$k_1 = f\left(t_k + \frac{h}{2}, y_k + k_0 \cdot \frac{h}{2}\right)$$

$$k_2 = f(t_k + \frac{h}{2}, y_k + k_1 \cdot \frac{h}{2})$$

$$k_3 = f(t_k + h, y_k + k_2 \cdot h)$$

Our for-loop based implementation can be found in *MY_RK4.m*

(b) Preform order reduction to the pendulum equations to create a state derivative function.

We utilized Matlab's symbolic toolbox for the task.

First we defined the symbolic variabiles:

```
syms t th1 th2 dth1 dth2 ddth1 ddth2 real
```

First, we defined the state equations:

```
eq1 = (m1+m2)*l1*ddth1 + m2*l2*ddth2*cos(th2-th1) == m2*l2*dth2^2*sin(th2-
th1)-(m1+m2)*g*sin(th1);

eq2 = l2*ddth2 + l1*ddth1*cos(th2-th1) == -l1*dth1^2*sin(th2-th1) -
g*sin(th2);
```

Then we asked the symbolic engine to extract $\ddot{\theta}_1$ from the first equation: $\ddot{\theta}_1 = \ddot{\theta}_1(\theta_1, \theta_2, \dot{\theta}_2, \ddot{\theta}_2)$

```
ddth1_updated = solve(eq1,ddth1);
```

We substituted our new term for $\ddot{\theta}_1$ into equation 2:

```
eq2 = subs(eq2,ddth1,ddth1_updated)
```

And with a couple more operations we managed to extract our state derivative vector such that $\begin{bmatrix}\ddot{\theta}_1\\ \ddot{\theta}_2\end{bmatrix} = \begin{bmatrix}\ddot{\theta}_1(\theta_1,\theta_2,\dot{\theta}_1,\dot{\theta}_2)\\ \ddot{\theta}_2(\theta_1,\theta_2,\dot{\theta}_1,\dot{\theta}_2)\end{bmatrix}$

```
ddth2_updated = solve(eq2,ddth2);
ddth1_updated2 = subs(ddth1_updated,ddth2,ddth2_updated);

y = [th1,th2,dth1,dth2];
dy = [dth1,dth2,ddth1_updated2,ddth2_updated]';
```

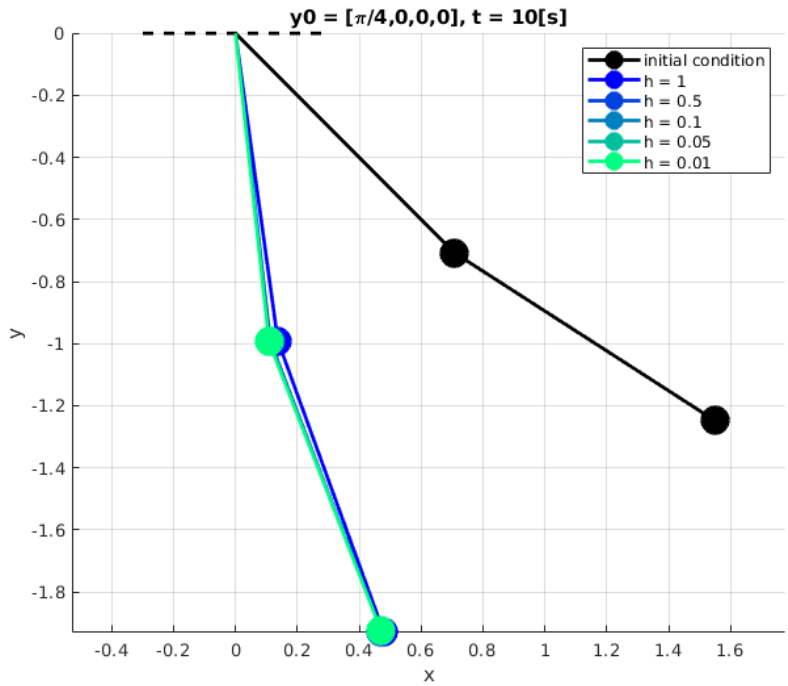
We than created the Matlab function *My_DoublePendulum* from our symbolic expression as follows:

```
txt = {'This function was created automatically from
funcbund.createMyDoublePendulum',...
    'accepts column vectors and returns column vectors',...
    't is entered to allow function to be called in ODEsolvers
even though its not used'};
fdy = matlabFunction(dy,'File','My_DoublePendulum',...
    'vars',{t,y'},...
    'comments',txt);
```

Question 2 – Solving for initial condition $[\theta_1, \theta_2\dot{\theta}_1, \dot{\theta}_2] = [\frac{\pi}{4}, 0,0,0]$

a) Solve for different time steps: $h = [0.01,0.05,0.1,0.5,1]$ and draw the solutions at $t = 10$.

We plotted the pendulum's state at $t = 10[s]$ for the given initial state, and different step sizes. The script can be found in *HW4.m* section *Q2*.

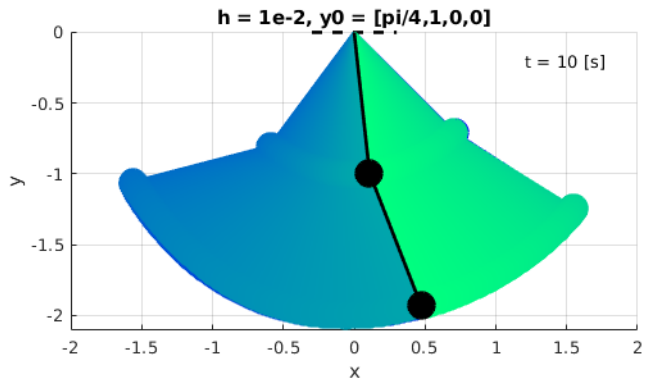


b) Analyze the results

The most precise solution is the one with the smallest time step. Still, we can see that all solutions are really "close" (depending on the application). Extremely stable systems tend to make an easy life with increased time steps, such is our case here.

c) Make a video

We made a cool video with flashy colors and all! Watch it in *h1e-2.avi*.
Here's a taste:



Question 3 – Adding a wall to the simulation at $x_{wall} = -0.5$

a) Update the solver so it stops if one of the balls hits the wall (distance less than 10^{-8})

We expanded upon the function from Question 1 with the following adjustments:

1. turning the *for* loop into a *while* loop (unknown amount of time steps due to reducing time step when approaching obstacles)
2. Adding a *break* rule for collision

   If $\left\|x_{ball_1} - x_{wall}\right\|_2 < 10^{-8}$ *or* $\left\|x_{ball_2} - x_{wall}\right\|_2 < 10^{-8}$, break out of the solver's loop.

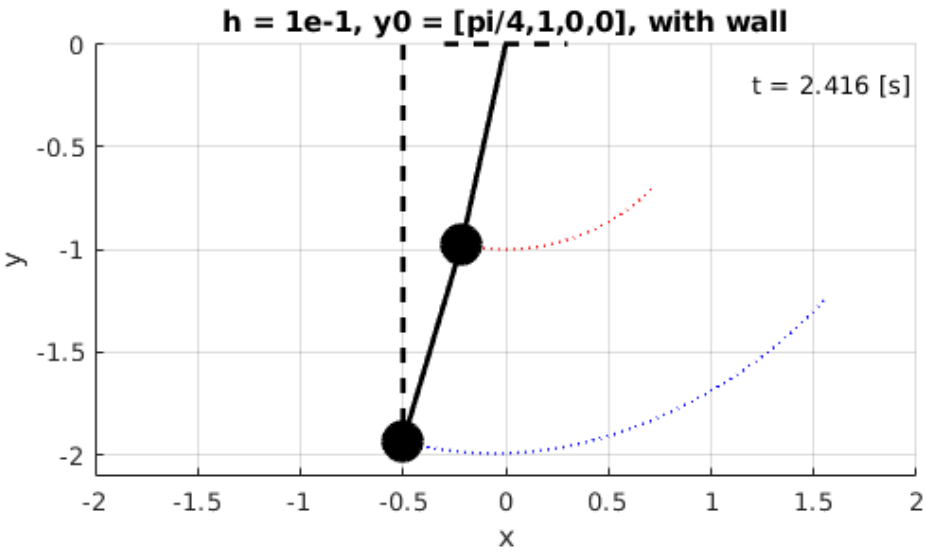
3. Adding a *continue* rule for reducing step size when approaching obstacle

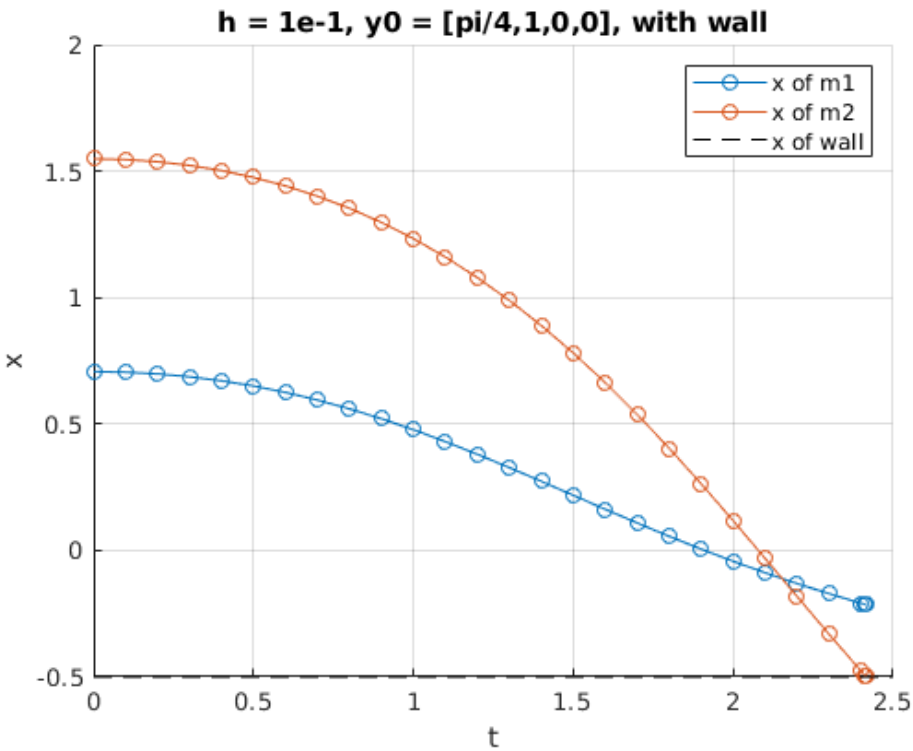
   If $\left(x_{ball_1} - x_{wall}\right) < 0$ *or* $\left(x_{ball_2} - x_{wall}\right) < 0$, go back to the previous time step and try to continue loop with time step cut in half.

You can view the function in *MY_RK4_event.m*

b) Plot the solution with the wall

We add two plots below, and a movie called *h1e-1wall.avi*
There you can verify that the simulation stopped after the second ball hit the wall, and that upon collision our solver reduced its time steps (more dots per dt at end of simulation, most visible in second plot).





You can re-create the plots via the third section in *HW4.m*

Question 4 – Shooting method theory

a) Finding $\theta_1|_{collision}$

Simple geometry would show that $\theta_1|_{collision} = \operatorname{asin}\left(\frac{x_{wall}}{l_1}\right) = 30°$

b) Explain the shooting method in your own words:

The shooting method incorporates our intelligent function zero-crossing or function minimization solvers into our ODE solvers to relax two-sided boundary problems into one-

sided. For example, if we are given an end state which we must achieve, and some condition on the initial state:

1. Start with a variable conditioned initial state (for example, $\dot{y}(0) = a$)
2. Evolve your ODE in time until you've reached the final time step
3. Calculate the distance between your solution and the wanted end state (You must invent some distance function, for example, $D = y(t_{final}) - 5$
4. Change the variable of your initial state such that the distance will be zero between the solution at the final time step and the wanted end state, still ensuring that it keeps the condition.
5. Repeat steps 2-4 while choosing initial conditions intelligently with a function minimizing (on the distance function) or zero crossing solver.

c) Explain the difficulties in using the shooting method in the case of 4th order differential equation

Boundary conditions take form in an algebraic equation composed of the state and its derivatives. For example:

$$y(0) + \dot{y}(0) = 0$$

Or

$$\dot{y}(t_{final}) + \sin\left(y(t_{final})\right) = 0$$

These equations are constraints that we must adhere to, and as in any problem, too many constraints and a solution will not exist. In a fourth degree ODE one can write up to four constraints equations on one boundary. To meet 4 constraints via the shooting method, one can employ 3 variables to optimize on in the other boundary.
Employing a function minimizer or zero crossing solver on three variables is very difficult. Often all we can do is use gradient decent methods and hope for the best. If it was just one variable, one would employ the bisection method.

d) Explain the difficulties in using the shooting method in case of the requested collision, regardless of the difficulty in section 3.

The requested collision end state can be for one of the balls only (as we have 4th order ODE comprising two states).
That is, our distance function whose roots we try to find is essentially broken into two.

$$D_{m_1 collision} = f(\theta_1, f\theta_2)$$

$$D_{m_2 collision} = f(\theta_1, f\theta_2)$$

In this case, one needs to 'watch out' for the sensitivity of the unconstrained state. In simple words: We found a solution such that m_1 collided with the wall, as we wanted. Was m_2 sent to the moon in the process?
Here we have a system that is extremely stable and easy to graph. What if we had a system of chemical responses?
Things get even more difficult when our simulation halts when m_2 hits the wall before m_1 does. How do we optimize our initial condition gusses from there? (We just start from a new initial guess, honestly).

Question 5 – Shooting method practice

We decided to use Matlab's *fsolve* to find the root of our distance function.
Code which runs and plots (movie style) all simulations can be found in *HW4.m* section 5.

Note:

$$x_{m_1} = l_1 \cos\left(\theta_1 - \frac{\pi}{2}\right)$$

$$x_{m_2} = x_{m_1} + l_2 \cos\left(\theta_2 - \frac{\pi}{2}\right)$$

a) Find an initial state $[\theta_1, \theta_2, 0,0]$ so that the pendulum's second link will collide flat with the wall.

Our distance function:

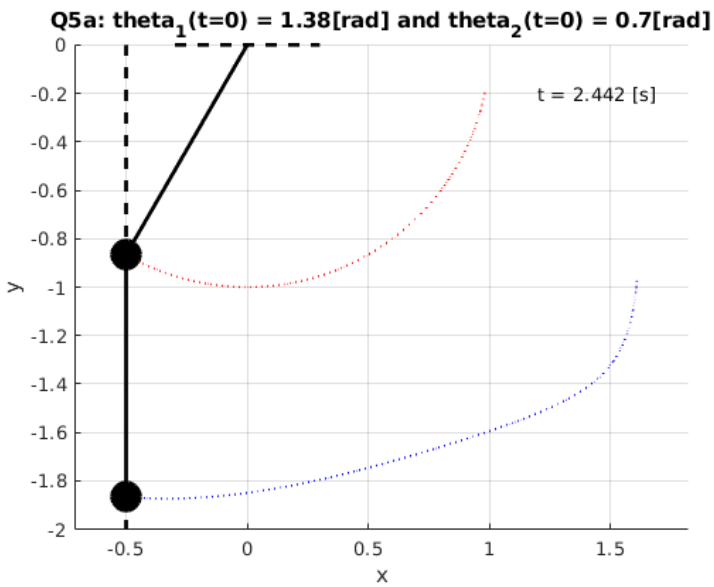
$$D_a(\theta_1, \theta_2) = \begin{bmatrix} [x_{m_1}(t_{final}) - x_{wall} \\ [x_{m_2}(t_{final}) - x_{wall} \end{bmatrix}$$

And Solution:

$$\theta_1 = 1.38[rad]$$

$$\theta_2 = 0.7[rad]$$

Implementation of the distance function can be found in code under *funcbund.f5a*



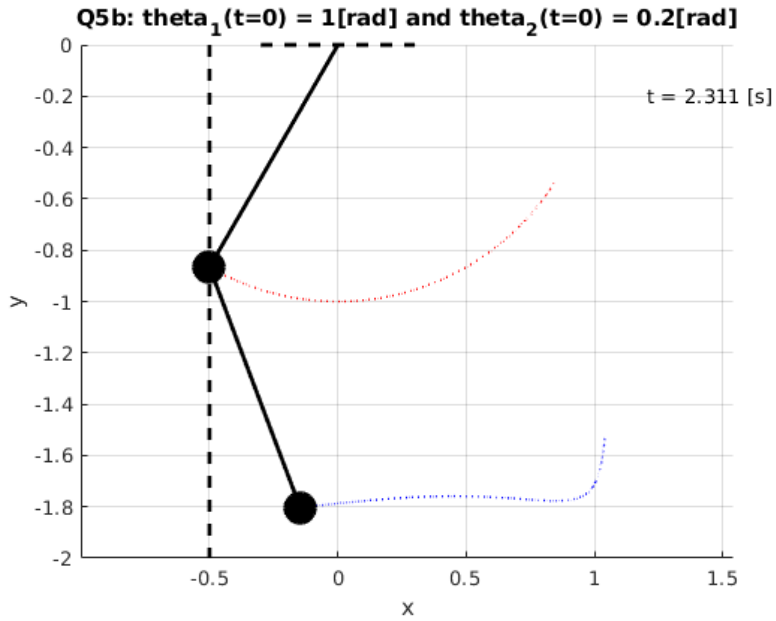
b) Find an initial state $[1, \theta_2, 0,0]$ so that m_1 will hit the wall

Our distance function:

$$D_b(\theta_2) = x_{m_1}(t_{final}) - x_{wall}$$

$$\theta_2 = 0.2[rad]$$

Implementation of the distance function can be found in code under *funcbund.f5b*



c) Find an initial state $[1, \theta_2, 0,0]$ so that m_2 will hit the wall

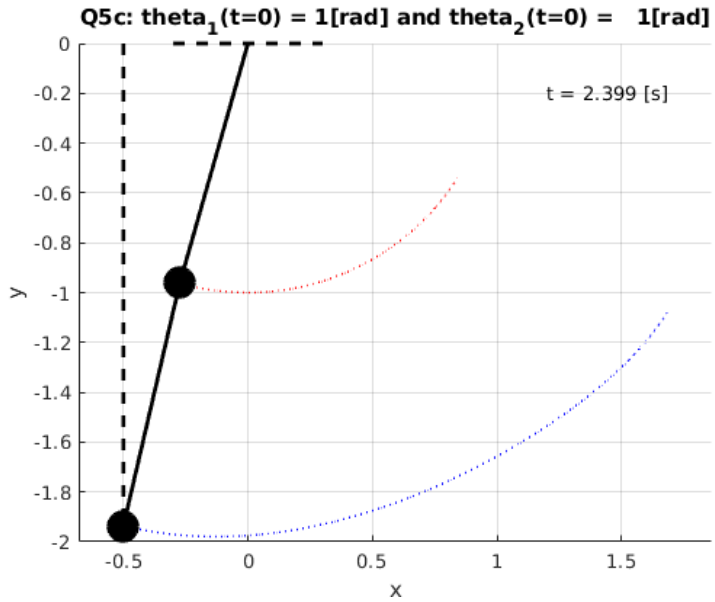
Our distance function:

$$D_c(\theta_2) = x_{m_2}(t_{final}) - x_{wall}$$

$$\theta_2 = 1[rad]$$

classic solution!

Implementation of the distance function can be found in code under *funcbund.f5c*



d) Are all solutions unique?

Of course not! There are infinite solutions to each of the problems in this section.
We have only one or two conditions on the state in t_{final} for a double state and continuous 4th order ODE. We could go up to four such conditions and then we would have been looking for a unique solution.