Weekly Report – W10 Fall 2022

Problem

1. Understanding of stiff systems;
2. The application areas for different ODE solvers in Simulink/MATLAB;
3. Continue with the falling SRA simulation based on the governing equations derived in Mathematica.

Solution

1. Definition and some of my understandings towards stiff systems

(1). Basic concepts of stiff equations

As the simulation tools like MATLAB/Simulink solve ODEs using numerical integration method which is quite different from the analytical/exact solutions computed manually, clearly utilizing different ODE solvers might lead to significant different solutions and simulation results, and the selection of ODE solver largely depends on whether the system is stiff or not, thereby, it's crucial important for us to understand the concept of stiff systems.

A stiff system can be composed of multiple differential equations, so we probably need to start from a single differential equation to learn about what is a stiff equation firstly. Generally, if the solution of a differential equation (worked out by numerical integration) is quite unstable unless a pretty small step size was chosen, the equation is so-called stiff. Or we can express it in a more understandable way, if the motion of a certain object can be written into a set of governing equations, there maybe exist several subsystems that have **significant variation rate**, this process or motion can be deemed to be "stiff".

If it is still too hard to understand this concept, the following examples might better demonstrate it. Consider we have two differential equations as follows,

$$\frac{dy_1}{dt} = -1000y_1 \qquad (W10-1)$$

$$\frac{dy_2}{dt} = -0.001y_2 \qquad (W10-2)$$

The exact solutions of them are $y_1 = e^{-1000t}$ and $y_2 = e^{-0.001t}$ respectively, however, these are done manually, it will be another story for numerical method, like Euler's method. According to the basic principles of that method, we have

$$y_{n+1} = y_n + hf(t_n, y_n) \qquad (W10-3)$$

where y_{n+1} and y_n stand for the estimated value of $(n+1)$th and nth steps of iteration of y, h is the step size can be selected according to our needs and $f(t_n, y_n)$ can be seen as a simple expression of the differentiation $\frac{dy}{dt}$, which is a function of both y and sampling time t. If we set the initial conditions for both the Eqn. (W10-1) and (W10-2) are $y_0 = y(0) = y(t_0) = y(t_0 = 0) = 1$, and the test values for h are 0.00001, 0.0005 , 0.0001 respectively, the simulation results in Simulink will be shown as follows,

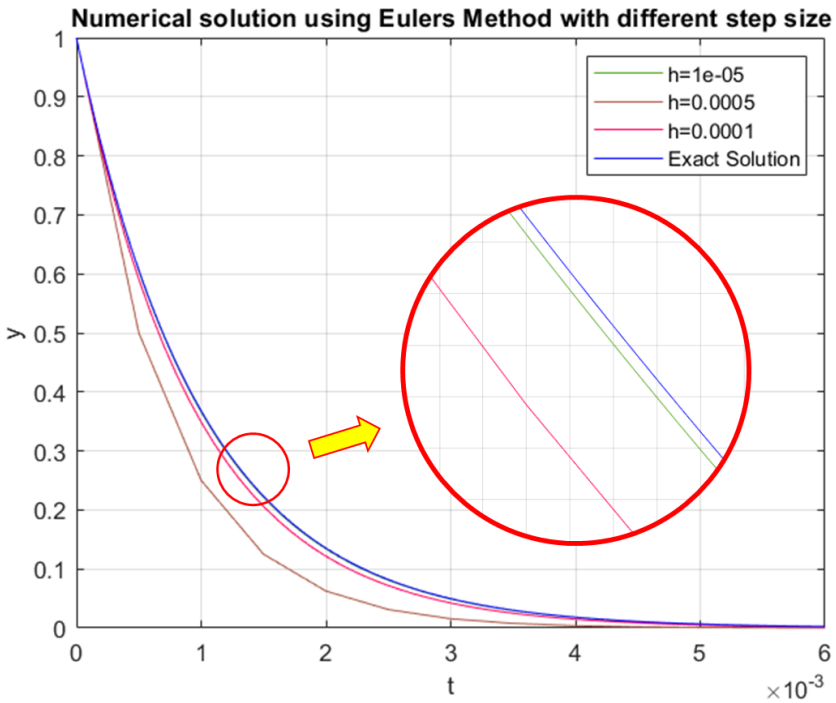

Fig. W10-1 The comparison of solutions of equation $\frac{dy_1}{dt} = -1000y_1$ with different step size

However, when the step size is relatively large, the computed results will not be accurate anymore, and moreover, with the increasing of the simulation time, the system which should converge now becomes unstable, the proof is shown as follows,

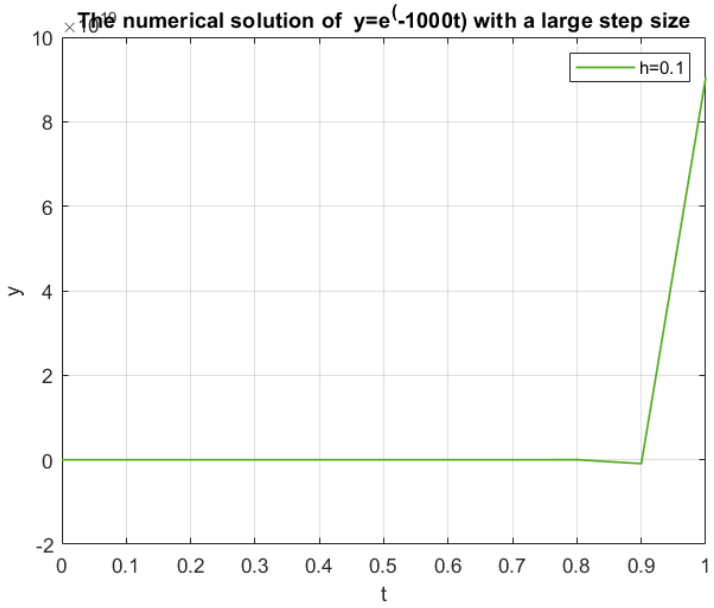

Fig. W10-2 The inaccurate solution for rapid decay system with large step size

Since smaller step size will lead to more accurate solution, we will try what happens for slow decay system y_2 in Eqn. (W10-2) with larger step size, $h = 1, 0.01, 0.005$ respectively. From the plot below, we can see that even with pretty large step size compared with that used in y_1, the difference between the numerical and exact solutions can be as minor as 0.002, which is acceptable.

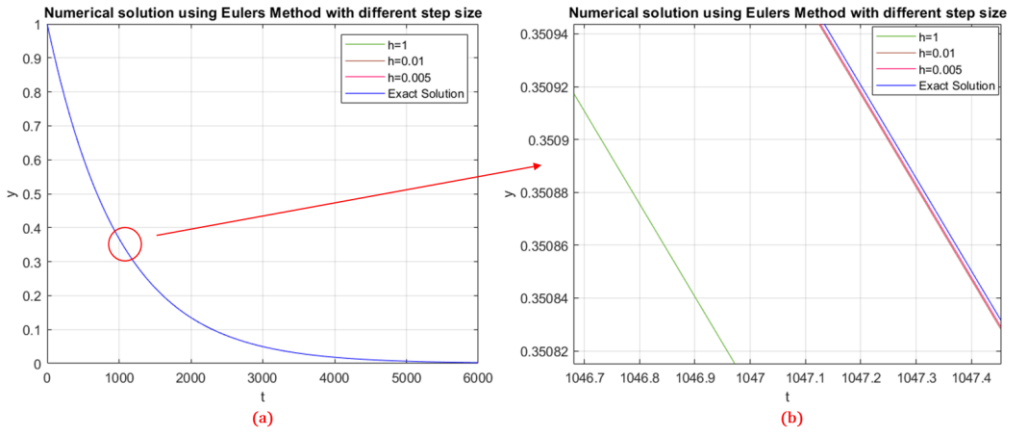

Fig. W10-3 The comparison between exact and numerical solutions with different values of step size h. (a) full scale solution in the simulation time. (b) partial enlarged drawing of circled position in (a).

Again, to testify the influence of step size on the accuracy of the computed solution, in this turn, a very small step size 0.00001 was chosen for simulation, the results will be displayed below.

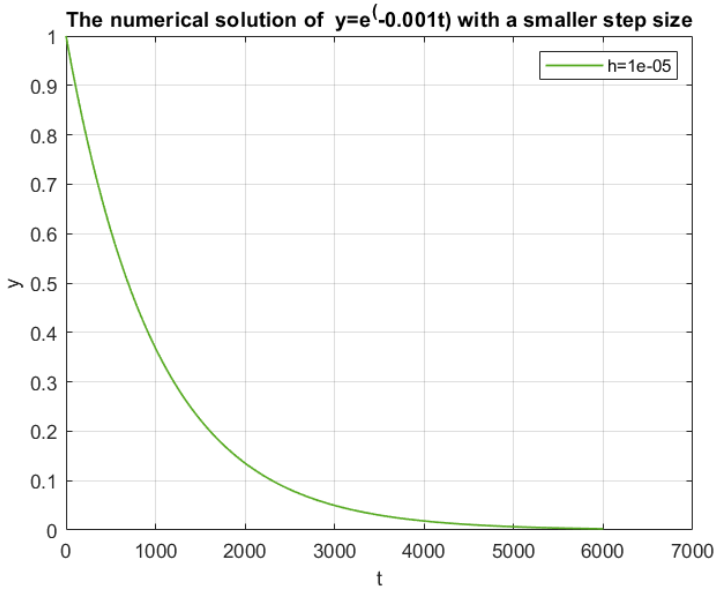

Fig. W10-4 The computed solution by Euler's Method of equation $\frac{dy_2}{dt} = -0.001y_2$ with a even smaller step size

From the simulation results above, we can conclude that smaller step size will only make the computed solution of ODE more accurate sacrificing the computation time, it took around 3 mins for my PC with 32 GB memory to simulate it. And we find that for fast decay or convergence system (with large $|\lambda|$ value, which is the coefficient before y_1 in Eqn. (W10-1)), the change is so large within a tiny time span, numerical solution with large step size will not reflect the real results, which means that the equation is stiff.

(2). Basic concepts about stiff systems

Slightly different from the definition of stiff ODE, a stiff system should be composed of at least one

equation that has very fast and very slow variation term (the $|\lambda|$ value) simultaneously, if we still use the example y_1 and y_2 in the previous part, now we can forge a new expression

$$y_3 = y_1 + y_2 = e^{-1000t} + e^{-0.001t} \qquad (W10-4)$$

And we find that if we set the step size as $h = 0.1$, the numerical solution for the fast variation part e^{-1000t} will be ignored, when the step size is chosen to be smaller and smaller, the characteristic of the equation will be reflected. In the Eqn. (W10-4), we can define that $\lambda_1 = -1000$, $\lambda_2 = -0.001$, a new concept, stiffness ratio can be brought in as follows,

$$\eta_s = \frac{|\lambda_1|}{|\lambda_2|} = 10^6 \qquad (W10-5)$$

And we find the ratio is very large, so the system is very stiff. And then another question might emerge, how could we know if a system is stiff or not, is there any rubric for the stiffness ratio? Before explaining this, let us review the general solution for a second order ODE,

$$y = \begin{cases} C_1 e^{r_1 x} + C_2 e^{r_2 x}, r_1 \neq r_2, \text{both of them are real roots} \\ C_1 e^{r_1 x} + C_2 x e^{r_2 x}, r_1 = r_2, \text{both of them are real roots} \\ e^{\alpha x}(C_1 \cos \beta x + C_2 \sin \beta x), r_1 \text{ and } r_2 \text{ are conjugate roots} \end{cases} \qquad (W10-6)$$

The stiffness ratio only has practical meaning when the system has two different real roots (for a second order system), it is sufficient for analysing most of the dynamics. For higher order systems, they will have more characteristic roots or eigenvalues, to express the stiffness ratio in a more general and rigorous way, we have

$$\eta_s = \frac{max\{|Re(\lambda_i)|\}}{min\{|Re(\lambda_i)|\}} \qquad (W10-7)$$

That is because only the real part of the roots domains the decay or growth rate. And now we consider an even more complex dynamic system with multiple state variables $q_1, q_2, \cdots, q_n$, we will have the following governing equations in the matrix format,

$$\underbrace{\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{bmatrix}_{n\times n}}_{\text{M matrix}} \cdot \underbrace{\begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \vdots \\ \ddot{q}_n \end{bmatrix}_{n\times 1}}_{\text{state accelerations}} + \underbrace{[C]_{n\times n}}_{\text{Coriolis matrix}} \cdot \underbrace{\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix}_{n\times 1}}_{\text{state velocities}} + \underbrace{\begin{bmatrix} P_1(q,\dot{q}) \\ P_2(q,\dot{q}) \\ \vdots \\ P_n(q,\dot{q}) \end{bmatrix}_{n\times 1}}_{\text{potential energy}} = \underbrace{\begin{bmatrix} \tau_1 \\ \tau_2 \\ \vdots \\ \tau_n \end{bmatrix}_{n\times 1}}_{\text{joint torque}} \qquad (W10-8)$$

And the exact solutions of the state variables q will be in the form of Eqn. (W10-6), in this case, there will be a stiffness ratio for each variable, compare each of them and then we can determine if the system is stiff or not. However, it can be extremely hard to obtain the exact solution of such system as the stiffness ratio varies with time continuously, the best way for doing simulation about system alike is to follow the suggestions for solving ODE (in MATLAB/Simulink), try ode45 first, because it can handle most of the problems, if it doesn't work, we can move on to try other implicit solvers, the specific information about ODE solvers will be introduced in Section 2.

Now we turn back to discuss the rubric to judge whether a system is stiff or not. According to the theory proposed by Lambert in 1973, a system can be classified as follows,

$$\eta_s: \begin{cases} 1 \leq \eta_s \leq 20, \text{non} - \text{stiff} \\ 20 < \eta_s \leq 10^3, \text{slightly stiff} \\ 10^3 < \eta_s \leq 10^6, \text{stiff} \\ \eta_s > 10^6, \text{very stiff} \end{cases} \qquad (W10-9)$$

If the ratio is greater than 1000 for a system, it is recommended that a smaller step size should be applied to guarantee the precision of the numerical solution.

2. Specifications of different ODE solvers in MATLAB/Simulink

Since in Section 1, we have spent a huge paragraph to demonstrate the definitions of stiff differential equations and systems, we know that the step size largely influences the accuracy of the numerical solution for a stiff system, meanwhile it's hard to determine if a complex system is stiff or not because the stiffness ratio changes with time, which is super dynamic, in this case, selection of proper ODE solver is extremely important for precise simulation results.

In general, the ODE solvers can be classified by explicit and implicit ones, the explicit ones are good at coping with non-stiff system; on the contrary, the explicit solvers are excel in dealing with stiff systems, which have the letter s,t or tb after the number of order, like ode 15s, ode 23t. For a relatively simple model, we can follow the steps to choose the ODE solver.

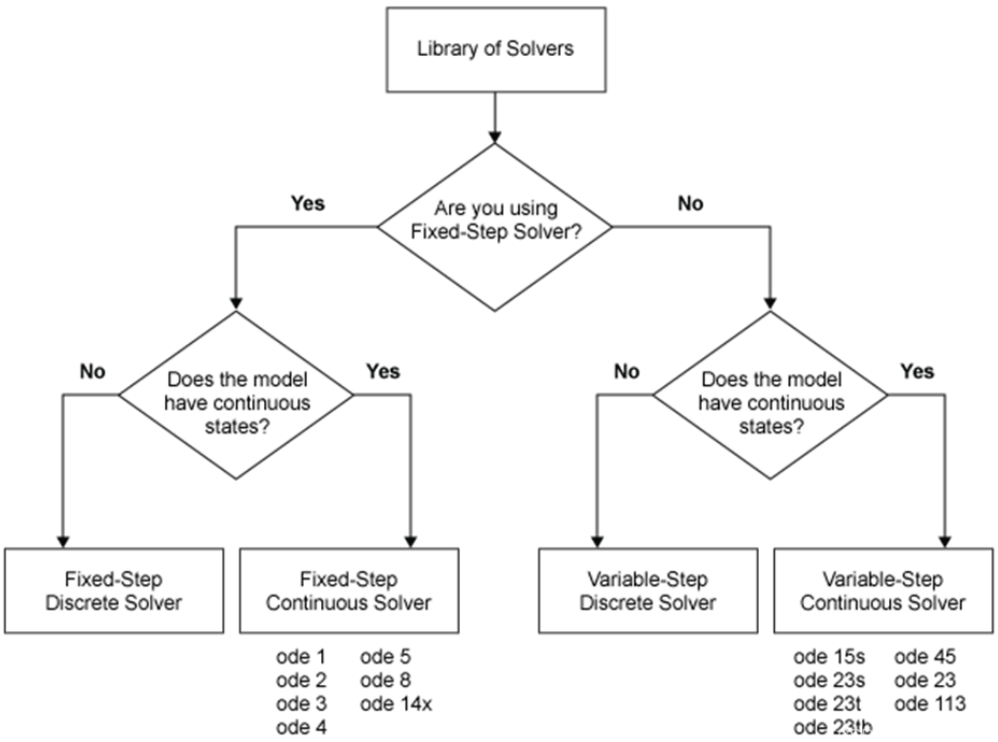

Fig. W10-5 The general procedure to choose proper ODE solver

However, when a model becomes more complex, we need to evaluate each solver.

(1). Fixed step ODE solvers

Theoretically if the step size was set to be small enough, any model can be simulated accurately, the only drawback is the computation cost. But it doesn't mean that it is of no use for fixed step solvers. For code generation used in embedded platform, fixed step solver must be used. And compared with variable step solvers, the fixed ones do not have to compute the step size for each step individually, which will take up additional memory of the computer, usually we need to do some tests using other solvers and analysis to confirm final step size on basis of making a balance between simulation precision and computation cost. The procedure to determine the step size will be shown as follows,

Step 1: Select at least three variable ODE solvers to get a rough baseline result;

Step 2: Select one of them as standard result which can be used to compared with the fixed step solution;

Step 3: Confirm an acceptable tolerance and step size;

Step 4: Perform the fixed step simulation and compare with the standard result you choose for the variable step solver in Step 2;

Step 5: Repeat the Step 3 and Step 4 until we have found the result of fixed step is within the tolerance, and confirm the maximum step size.

(2). Variable Step ODE Solvers (most frequently used)

1). Explicit solvers

ode 45: First choice for most problems with medium accuracy, if the computation is slow, we should try to use some implicit solvers because the system might be stiff.

ode 23: If the error tolerance is cruder than ode 45, and the system is slightly stiff, the efficiency can be improved using this solver, but the accuracy might be a little lower.

ode 113: Multiple step solver, if the error tolerance rule is strict according to our simulation needs, it is recommended to use this one, the accuracy can be variable from low to high.

2). Implicit solvers

ode 15s: Multiple step solver, it can handle stiff system, though with the increasing of orders, the accuracy will be improved, the stability will be decreased; if the system is stiff and stable, the highest order should be set to 2 or use ode 23s instead. (accuracy: low to medium)

ode 23s: Single step solver, applicable for moderate error tolerance, second choice for solving stiff system problems.

ode 23t & ode 23tb: Use trapezoid rule to realize, the accuracy is low.

3. Falling SRA simulation

To avoid the inconvenience of substitution of symbolic variables in MATLAB/Simulink, this week I turned back to use Mathematica to derive the governing equations of the system from the very beginning, the rotation matrix of each link. The whole process can be seen in Appendix, there were too many formatting issues when converting the expressions into MATLAB even with existing add-on and subscript "ToMatlab.m", it took me quite a long time to fix them, but there are still some errors when running the Simulink model, currently I'm still working on it and I hope I can get some simulation results tomorrow.

Difficulty

1. I think I'm just in the entrance level for understanding the ODE solvers in MATLAB/Simulink, I probably need more practical use of different ODE solvers to accumulate the experience of choosing solvers for proper scenarios. Since there are no straightforward formulas for us to choose the best solver, contemporarily the most ideal way is to compare first;
2. For verification of the inertia matrix of the SRA simulation, all the expressions or matrices derived this week were by Mathematica, using the method of judging the sign of trace and determinant, I found that there existed even higher order terms, which can be up to 20^{th}, I have no idea how to cope with that. One of the reasons for its emergence could be that our model is a soft and elastic model rather than a rigid body.

Appendix

```
R1 = {{Cos[θ1], Sin[θ1], 0, 2 * L1 * (Sin[θ1 / 2]) ^2 / θ1},
    {-Sin[θ1], Cos[θ1], 0, L1 * Sin[θ1] / θ1}, {0, 0, 1, 0}, {0, 0, 0, 1}};

R2 = {{Cos[θ2], Sin[θ2], 0, 2 * L2 * (Sin[θ2 / 2]) ^2 / θ2},
    {-Sin[θ2], Cos[θ2], 0, L2 * Sin[θ2] / θ2}, {0, 0, 1, 0}, {0, 0, 0, 1}};

Rintm1 = Simplify[R1];
Rintm2 = Simplify[R1.R2];

Rzdown = Simplify[{{Cos[-Pi / 2], -Sin[-Pi / 2], 0, pfx},
     {Sin[-Pi / 2], Cos[-Pi / 2], 0, pfy}, {0, 0, 1, 0}, {0, 0, 0, 1}}];
Rpos1 = Simplify[Rzdown.Rintm1];
Rpos2 = Simplify[Rzdown.Rintm2];

px1 = Rpos1[[1, 4]]; py1 = Rpos1[[2, 4]];
px2 = Rpos2[[1, 4]]; py2 = Rpos2[[2, 4]];
deltapx1 = py1 - pfy; deltapy1 = px1 - pfx;
deltapx2 = py2 - py1; deltapy2 = px2 - px1;
potentialgy1 = deltapy1 / 2; potentialgy2 = py1 + deltapy2 / 2;
Rcoord1 = Rpos1[[1 ;; 3, 4]];
Rcoord2 = Rpos2[[1 ;; 3, 4]];

Jv1 = Simplify[D[{Rcoord1}, {{θ1, θ2}}]];
Jv2 = Simplify[D[{Rcoord2}, {{θ1, θ2}}]];
Jv1 = Join[{Jv1[[1, 1]], Jv1[[1, 2]], Jv1[[1, 3]]}];
Jv2 = Join[{Jv2[[1, 1]], Jv2[[1, 2]], Jv2[[1, 3]]}];

I1 = 1 / 12 * m1 * (L1) ^2; I2 = 1 / 12 * m2 * (L2) ^2;
Rtwist1 = Rpos1[[1 ;; 3, 3]]; Rtwist2 = Rpos2[[1 ;; 3, 3]];
Jw1 = Simplify[D[{Rtwist1 * θ1 / 2}, {{θ1, θ2}}]];
Jw2 = Simplify[D[{Rtwist1 * θ1 / 2 + Rtwist2 * θ2 / 2}, {{θ1, θ2}}]];
Jw1 = Join[{Jw1[[1, 1]], Jw1[[1, 2]], Jw1[[1, 3]]}];
Jw2 = Join[{Jw2[[1, 1]], Jw2[[1, 2]], Jw2[[1, 3]]}];

(*Using the Jacobian matrices derived above,
we can work out the inertia matrix and Coriolis matrix respectively.*)

M = Simplify[m1 * Transpose[Jv1].Jv1 +
     I1 * Transpose[Jw1].Jw1 + m2 * Transpose[Jv2].Jv2 + I2 * Transpose[Jw2].Jw2];

(*Simplify[Eigenvalues[M]];*)
(*If the Coriolis-Centripedal maatrix was directly defined as C in Mathematica,
after computing the cell, it will display C is protected in workspace,
it's probably because captial C is a function,
so shall change the name of C matrix into "Cmatrix" instead.*)
Cmatrix = {{0, 0}, {0, 0}};
```

In[ ]:=
```
Cmatrix[[1, 1]] = (D[M[[1, 1]], {θ1}] + D[M[[1, 1]], {θ1}] - D[M[[1, 1]], {θ1}]) * θ1dot / 2 +
    (D[M[[1, 1]], {θ2}] + D[M[[1, 1]], {θ1}] - D[M[[2, 1]], {θ1}]) * θ2dot / 2;
Cmatrix[[1, 2]] = (D[M[[1, 2]], {θ1}] + D[M[[1, 1]], {θ2}] - D[M[[1, 2]], {θ1}]) * θ1dot / 2 +
    (D[M[[1, 2]], {θ2}] + D[M[[1, 2]], {θ2}] - D[M[[2, 2]], {θ1}]) * θ2dot / 2;
Cmatrix[[2, 1]] = (D[M[[2, 1]], {θ1}] + D[M[[2, 1]], {θ1}] - D[M[[1, 1]], {θ2}]) * θ1dot / 2 +
    (D[M[[2, 1]], {θ2}] + D[M[[2, 2]], {θ1}] - D[M[[2, 1]], {θ2}]) * θ2dot / 2;
Cmatrix[[2, 2]] = (D[M[[2, 2]], {θ1}] + D[M[[2, 1]], {θ2}] - D[M[[1, 2]], {θ2}]) * θ1dot / 2 +
    (D[M[[2, 2]], {θ2}] + D[M[[2, 2]], {θ2}] - D[M[[2, 2]], {θ2}]) * θ2dot / 2;
```

In[ ]:=
```
(*Export the expressions of M and C matrices to MATLAB file*)
(*Sometimes it doesn't work for the following line of code,
we have to upload the external code manually by clicking "file → install"*)
(*Import[
  "D:\\R - Clemson\\SRA Program\\Weekly Report\\W10 - Dynamcis Derivation\\ToMatlab.m",
  "Package"];*)
PrintMatlab[M]
PrintMatlab[Cmatrix]
```

```
[(1/48).*L1.^2.*m1+(1/48).*L2.^2.*m2+L1.^2.*m1.*θ1.^(-4).*( ...
  2+θ1.^2+(-2).*cos(θ1)+(-2).*θ1.*sin(θ1))+m2.*θ1.^(-4).*θ2.^( ...
  -2).*((θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+(-1).*L2.*θ1.^2.* ...
  cos(θ1+θ2)+L1.*θ2.*sin(θ1)).^2+(L1.*θ2+(-1).*L1.*θ2.*cos(θ1) ...
  +θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+(-1).*L2.*θ1.^2.*sin(θ1+ ...
  θ2)).^2),(1/48).*L2.*m2.*θ1.^(-2).*θ2.^(-2).*((-48).*L1+48.* ...
  L2.*θ1.^2+L2.*θ1.^2.*θ2.^2+48.*L1.*cos(θ1)+48.*(L1+(-1).* ...
  L2.*θ1.^2+L1.*θ1.*θ2).*cos(θ2)+(-48).*L1.*cos(θ1+θ2)+(-48).* ...
  L1.*θ1.*sin(θ2)+48.*L1.*θ2.*sin(θ2)+(-48).*L1.*θ2.*sin(θ1+ ...
  θ2));(1/48).*L2.*m2.*θ1.^(-2).*θ2.^(-2).*((-48).*L1+48.*L2.* ...
  θ1.^2+L2.*θ1.^2.*θ2.^2+48.*L1.*cos(θ1)+48.*(L1+(-1).*L2.* ...
  θ1.^2+L1.*θ1.*θ2).*cos(θ2)+(-48).*L1.*cos(θ1+θ2)+(-48).*L1.* ...
  θ1.*sin(θ2)+48.*L1.*θ2.*sin(θ2)+(-48).*L1.*θ2.*sin(θ1+θ2)),( ...
  1/48).*L2.^2.*m2.*θ2.^(-4).*(96+48.*θ2.^2+θ2.^4+(-96).*cos( ...
  θ2)+(-96).*θ2.*sin(θ2))];


[(1/2).*θ1dot.*(L1.^2.*m1.*θ1.^(-4).*(2.*θ1+(-2).*θ1.*cos( ...
  θ1))+(-4).*L1.^2.*m1.*θ1.^(-5).*(2+θ1.^2+(-2).*cos(θ1)+(-2) ...
  .*θ1.*sin(θ1))+(-4).*m2.*θ1.^(-5).*θ2.^(-2).*((θ1.*(L2.*θ1+( ...
  -1).*L1.*θ2).*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.* ...
  sin(θ1)).^2+(L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).* ...
  L1.*θ2).*sin(θ1)+(-1).*L2.*θ1.^2.*sin(θ1+θ2)).^2)+m2.*θ1.^( ...
  -4).*θ2.^(-2).*(2.*(θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+(-1) ...
  .*L2.*θ1.^2.*cos(θ1+θ2)+L2.*θ1.*sin(θ1)+L1.*θ2.*sin(θ1)+( ...
  L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+(-2).*L2.*θ1.*sin(θ1+θ2)).*( ...
  L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin( ...
  θ1)+(-1).*L2.*θ1.^2.*sin(θ1+θ2))+2.*(θ1.*(L2.*θ1+(-1).*L1.* ...
  θ2).*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.*sin(θ1)).*( ...
  L2.*θ1.*cos(θ1)+L1.*θ2.*cos(θ1)+(L2.*θ1+(-1).*L1.*θ2).*cos( ...
  θ1)+(-2).*L2.*θ1.*cos(θ1+θ2)+(-1).*θ1.*(L2.*θ1+(-1).*L1.*θ2) ...
  .*sin(θ1)+L2.*θ1.^2.*sin(θ1+θ2)))))+(1/2).*θ2dot.*(L1.^2.* ...
  m1.*θ1.^(-4).*(2.*θ1+(-2).*θ1.*cos(θ1))+(-4).*L1.^2.*m1.* ...
```

```
θ1.^(-5).*(2+θ1.^2+(-2).*cos(θ1)+(-2).*θ1.*sin(θ1))+(-1/48) ...
.*L2.*m2.*θ1.^(-2).*θ2.^(-2).*(96.*L2.*θ1+2.*L2.*θ1.*θ2.^2+ ...
48.*((-2).*L2.*θ1+L1.*θ2).*cos(θ2)+(-48).*L1.*θ2.*cos(θ1+θ2) ...
+(-48).*L1.*sin(θ1)+(-48).*L1.*sin(θ2)+48.*L1.*sin(θ1+θ2))+( ...
1/24).*L2.*m2.*θ1.^(-3).*θ2.^(-2).*((-48).*L1+48.*L2.*θ1.^2+ ...
L2.*θ1.^2.*θ2.^2+48.*L1.*cos(θ1)+48.*(L1+(-1).*L2.*θ1.^2+ ...
L1.*θ1.*θ2).*cos(θ2)+(-48).*L1.*cos(θ1+θ2)+(-48).*L1.*θ1.* ...
sin(θ2)+48.*L1.*θ2.*sin(θ2)+(-48).*L1.*θ2.*sin(θ1+θ2))+(-2) ...
.*m2.*θ1.^(-4).*θ2.^(-3).*((θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos( ...
θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.*sin(θ1)).^2+(L1.*θ2+ ...
(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+( ...
-1).*L2.*θ1.^2.*sin(θ1+θ2)).^2)+(-4).*m2.*θ1.^(-5).*θ2.^(-2) ...
.*((θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+(-1).*L2.*θ1.^2.*cos( ...
θ1+θ2)+L1.*θ2.*sin(θ1)).^2+(L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+ ...
θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+(-1).*L2.*θ1.^2.*sin(θ1+ ...
θ2)).^2)+m2.*θ1.^(-4).*θ2.^(-2).*(2.*(L1+(-1).*L1.*cos(θ1)+( ...
-1).*L2.*θ1.^2.*cos(θ1+θ2)+(-1).*L1.*θ1.*sin(θ1)).*(L1.*θ2+( ...
-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+(-1) ...
.*L2.*θ1.^2.*sin(θ1+θ2))+2.*(θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos( ...
θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.*sin(θ1)).*((-1).* ...
L1.*θ1.*cos(θ1)+L1.*sin(θ1)+L2.*θ1.^2.*sin(θ1+θ2)))+m2.* ...
θ1.^(-4).*θ2.^(-2).*(2.*(θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+ ...
(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L2.*θ1.*sin(θ1)+L1.*θ2.*sin(θ1)+ ...
(L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+(-2).*L2.*θ1.*sin(θ1+θ2)).*( ...
L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin( ...
θ1)+(-1).*L2.*θ1.^2.*sin(θ1+θ2))+2.*(θ1.*(L2.*θ1+(-1).*L1.* ...
θ2).*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.*sin(θ1)).*( ...
L2.*θ1.*cos(θ1)+L1.*θ2.*cos(θ1)+(L2.*θ1+(-1).*L1.*θ2).*cos( ...
θ1)+(-2).*L2.*θ1.*cos(θ1+θ2)+(-1).*θ1.*(L2.*θ1+(-1).*L1.*θ2) ...
.*sin(θ1)+L2.*θ1.^2.*sin(θ1+θ2))),(1/2).*θ2dot.*((1/24).* ...
L2.*m2.*θ1.^(-2).*θ2.^(-2).*(2.*L2.*θ1.^2.*θ2+48.*L1.*θ2.* ...
cos(θ2)+(-48).*L1.*θ2.*cos(θ1+θ2)+48.*L1.*sin(θ2)+(-48).*( ...
L1+(-1).*L2.*θ1.^2+L1.*θ1.*θ2).*sin(θ2))+(-1/12).*L2.*m2.* ...
θ1.^(-2).*θ2.^(-3).*((-48).*L1+48.*L2.*θ1.^2+L2.*θ1.^2.* ...
θ2.^2+48.*L1.*cos(θ1)+48.*(L1+(-1).*L2.*θ1.^2+L1.*θ1.*θ2).* ...
cos(θ2)+(-48).*L1.*cos(θ1+θ2)+(-48).*L1.*θ1.*sin(θ2)+48.* ...
L1.*θ2.*sin(θ2)+(-48).*L1.*θ2.*sin(θ1+θ2)))+(1/2).*θ1dot.*(( ...
-2).*m2.*θ1.^(-4).*θ2.^(-3).*((θ1.*(L2.*θ1+(-1).*L1.*θ2).* ...
cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.*sin(θ1)).^2+( ...
L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin( ...
θ1)+(-1).*L2.*θ1.^2.*sin(θ1+θ2)).^2)+m2.*θ1.^(-4).*θ2.^(-2) ...
.*(2.*(L1+(-1).*L1.*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+(-1) ...
.*L1.*θ1.*sin(θ1)).*(L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.* ...
θ1+(-1).*L1.*θ2).*sin(θ1)+(-1).*L2.*θ1.^2.*sin(θ1+θ2))+2.*( ...
θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+ ...
θ2)+L1.*θ2.*sin(θ1)).*((-1).*L1.*θ1.*cos(θ1)+L1.*sin(θ1)+ ...
L2.*θ1.^2.*sin(θ1+θ2))));(1/2).*θ1dot.*((1/24).*L2.*m2.* ...
θ1.^(-2).*θ2.^(-2).*(96.*L2.*θ1+2.*L2.*θ1.*θ2.^2+48.*((-2).* ...
L2.*θ1+L1.*θ2).*cos(θ2)+(-48).*L1.*θ2.*cos(θ1+θ2)+(-48).* ...
L1.*sin(θ1)+(-48).*L1.*sin(θ2)+48.*L1.*sin(θ1+θ2))+(-1/12).* ...
L2.*m2.*θ1.^(-3).*θ2.^(-2).*((-48).*L1+48.*L2.*θ1.^2+L2.* ...
```

```
θ1.^2.*θ2.^2+48.*L1.*cos(θ1)+48.*(L1+(-1).*L2.*θ1.^2+L1.* ...
θ1.*θ2).*cos(θ2)+(-48).*L1.*cos(θ1+θ2)+(-48).*L1.*θ1.*sin( ...
θ2)+48.*L1.*θ2.*sin(θ2)+(-48).*L1.*θ2.*sin(θ1+θ2))+2.*m2.* ...
θ1.^(-4).*θ2.^(-3).*((θ1.*(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+( ...
-1).*L2.*θ1.^2.*cos(θ1+θ2)+L1.*θ2.*sin(θ1)).^2+(L1.*θ2+(-1) ...
.*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+(-1).*L1.*θ2).*sin(θ1)+(-1).* ...
L2.*θ1.^2.*sin(θ1+θ2)).^2)+(-1).*m2.*θ1.^(-4).*θ2.^(-2).*( ...
2.*(L1+(-1).*L1.*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+(-1).* ...
L1.*θ1.*sin(θ1)).*(L1.*θ2+(-1).*L1.*θ2.*cos(θ1)+θ1.*(L2.*θ1+ ...
(-1).*L1.*θ2).*sin(θ1)+(-1).*L2.*θ1.^2.*sin(θ1+θ2))+2.*(θ1.* ...
(L2.*θ1+(-1).*L1.*θ2).*cos(θ1)+(-1).*L2.*θ1.^2.*cos(θ1+θ2)+ ...
L1.*θ2.*sin(θ1)).*((-1).*L1.*θ1.*cos(θ1)+L1.*sin(θ1)+L2.* ...
θ1.^2.*sin(θ1+θ2)))),(1/2).*θ2dot.*((1/48).*L2.^2.*m2.*θ2.^( ...
-4).*(96.*θ2+4.*θ2.^3+(-96).*θ2.*cos(θ2))+(-1/12).*L2.^2.* ...
m2.*θ2.^(-5).*(96+48.*θ2.^2+θ2.^4+(-96).*cos(θ2)+(-96).*θ2.* ...
sin(θ2)))];
```