**NATIONAL UNIVERSITY OF SINGAPORE**

**EE4307: Control System Design and Simulation**

**Project**

This project constitutes the first part of project (40%) of the final module score, while the second part of the project (40%) will be given to you in due course.

Students are to work on the project independently, although parts of this manual are designed for their guidance. Laboratory assistants will also be available to provide limited guidance. The objectives of this project are:

1. to provide hands-on experience in the acquisition and processing of real data from real physical systems and the use of computer-based data acquisition devices;
2. to familiarize with key non-parametric approaches to system identification for control design purposes;
3. to familiarize with some common and important functions collected within the *System Identification ToolBox* of MATLAB and Python
4. to provide learning experience over the entire cycle of parametric system identification;
5. to provide the experience with computer simulation of dynamical systems; and
6. to simulate and implement real-time control systems which are designed based on the models obtained earlier.

**1. Introduction**

Controller designs for dynamical systems are centred around a system model, which can be in a non-parametric or parametric form. There are systematic steps which should be undertaken before a control system can be designed and finally implemented.

First, control variables and objectives are defined, based on which the relevant information of the system should be collected in the form of raw data. Modern computer-based data acquisition devices can be used efficiently for this data logging process. The raw data collected may need to be further conditioned, in view of noise and other disturbances inevitably present, before it can be used for the next phase of system modelling.

System modelling can be carried out with a physical approach, using fundamental first principles to relate the variables of interests. However, this physical approach is typically constrained to small and relatively simple systems. Even then, there are unknown physical parameters to be determined. Another approach is based on system identification which uses only the logged data of the system and a pre- selected model structure to yield a model. In a way, this can be considered a “black- box” approach as the system, to be identified, is treated as an unknown black box to be fitted to the model structure based on observations of the system only. In this project, the two approaches of system identification mentioned above will be explored.

System identification approaches can be broadly classified under a non-parametric or parametric approach. A non-parametric approach will yield a non-parametric mode of the system, unconstrained to any parametric structure. For simple control design, such information, which can be in the time or frequency domain, is sufficient. A parametric approach, on the other hand, will yield a full parametric model and it involves an iterative loop to arrive at the final model. This process involves decision making on the part of the person in search of models, as well as fairly demanding computations to furnish bases for these decisions. A user will typically go through several iterations in the process of arriving at a final model, where at each step, earlier decisions are revised. Model-based control designs can be carried out when the parametric model is obtained.

For system identification, interactive software is a natural tool for approaching system identification. It is also a convenient way to package the rather extensive theory, making it available to the user. MATLAB is an excellent environment for such interactive calculations, with its workspace concept, graphing facilities, and easy data import and export. The System Identification Toolbox is a collection of M-files that implement the most common and useful parametric and non-parametric techniques available. It is specifically aimed at giving the user help, not only at computing, but also at evaluating models. Through this project, students will realize how easily a system identification task may be accomplished through the use of such modern interactive software.

Once a model becomes available and the controller designed, a simulation study on the adequateness of the model and the control system can be done prior to actual implementation on the real system. Simulation is usually done for prediction, control or even training purposes. The simulation study will allow fine adjustment to the model and/or controller until a satisfactory level of performance is achieved. This step is useful and necessary in many case, as it is not always possible to do an experiment directly on the actual system over a prolonged period for time, due to cost and safety concerns, or due to the fact that the actual system simply have not yet being developed.

The project will bring the students through all the phases of control design and simulation. In Section 2, you will learn different non-parametric approaches. In Section 3, you will be familiar with some control methods. In Section 4, some mini-projects are given, you are required to do system simulation.

**2. System Identification: Non-Parametric Approach**

In this part of the project, the students will familiarize themselves with the different non-parametric approaches in identifying a system. They will be guided through a few examples using MATLAB before applying it on their assigned system.

***2.1 A Guided Example (MATLAB)***

In this example, you will be guided through MATLAB to plot a given set of data and using the tools in MATLAB to condition the data for analysis.

Load the data file that was saved above using the following command

load *c:\EE4307\your\_name\part4\_1.dat*

You will get an *n* by 2 matrix in the workspace. The name of the matrix will be

*part4\_1*

Rename the matrix by assigning

*X = part4\_1*

Using the commands below you will be able to obtain a plot of the input and output.

*plot(X(:,1))* – Gives a plot of the input vs. sample number

*plot(X(:,2))* – Gives a plot of the output vs. sample number

*plot(a, b(:,1:2)))* – Gives a plot of b(1) and b(2) vs. a

Before the data can be used for non-parametric or parametric analysis, it has to be post treated and conditioned.

Post-treatment of the data can be done by choosing a part of the data which is suitable for analysis, removing the offset and high frequency noise.

Certain data range can be chosen by using the following command. Assume the index of the start sample of the range to be Start and the index of the end sample to be *Final*.

*Y = X(:,Start:Final);*

You can remove the offset by deducting all the output values by a constant, min, which is to be determined by you.

*for i = Start:Final*

*Y(i – Start + 1,2)=X(i,2) – min;*

*end*

After removing the offset, the data can be filtered to remove high frequency noise by using the following command. Assume the cut-off frequency in terms of the Nyquist frequency is given by w and the order of filter required is *N.*

*W = idfilt(Y(:,2), N, w, ‘noncausal’);*

The high frequency noise can also be removed by resampling at *m* times the original sampling period.

*j = 1;*

*for i = Start:Final*

*if(rem(i,m) == 0)*

*Y(j,1) = X(i,1);*

*Y(j,2) = X(i,2);*

*j = j+1;*

*end*

*end*

The data will be ready for analysis after conditioning. List and explain any additional conditioning that you have done on the data.

Plot the input and output data using the data file obtained in Part 4.1 and conditioned using the methods described above.

***2.2 Exercise on Data Conditioning***

A set of data is given in the file ***cond.dat*** in the folder ***EE4307/Part4***, load the file into MATLAB. Using the methods described above or otherwise obtains a conditioned data set.

The following conditions are given:

* The data is of the form:

|  |  |  |
| --- | --- | --- |
| Time | Input | Output |

* A step analysis of the system is to be done using the data,
* The output is contaminated with high frequency noise with zero mean. Plot the conditioned input and output data.

***2.2 A Guided Example (Time-Domain Method)***

In this example, you will be guided through the process of using time-domain methods of non-parametric identification using MATLAB. Given a digital system described below

Using MATLAB, define the system using the following command

*G = tf([1 0],[1 0.5], 1);*

Get the impulse response of the system by

*X = impulse(G, [0:1:15])*

As discussed in the lectures, the impulse-response coefficients of the system is given by

For a unit impulse input to the system. Therefore the impulse-response coefficients are given by the matrix X.

Given a system described below

Using MATLAB, define the system using the following command

*G = tf([1],[1 1 2],’inputdelay’, 0.3);*

Get the step response of the system by

*step(G)*

Find the parameters τ, time delay, , settling time, and *M* from the figure.

***2.3 Non-Parametric Identification (Time-Domain Methods)***

Given two sets of data, you are to perform conditioning and identify the system using non-parametric approach, based on the following requirements:

* The data is corrupted at the output with high frequency noise,
* The data is of the form:

|  |  |  |
| --- | --- | --- |
| Time | Input | Output |

* The data is stored in two separate files, ***EE4307/Part5/set1.dat*** and ***EE4307/Part5/set2.dat***, and they are based on two different sets of input to the same system.

You are required to;

* Perform an identification, either by finding the impulse response coefficients or/and fitting a transfer function, on the system using both sets of data. The sampling period for the required system model is 1 sec,
* Perform the analysis with conditioning and compare the results,
* Find the time domain equations of the system.

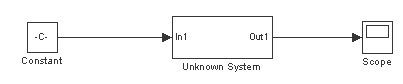
***2.4 A Guided Example (Defining Operating Point)***

In this part, we will need to define the operating point for a given plant. Open SIMULINK in MATLAB by typing the following command in the MATLAB command window.

*simulink*

Open the following SIMULINK file ***EE4307/Part5/operating.mdl***

Given that an unknown nonlinear system below, due to the operation requirements, we want to identify the system about a few particular operating points.

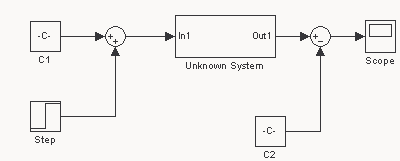


Unknown System Block Diagram

Given that the first operating point is to be at

Find the corresponding .

Change the above block diagram to the following form, where *C1* is given by and *C2* is given by .

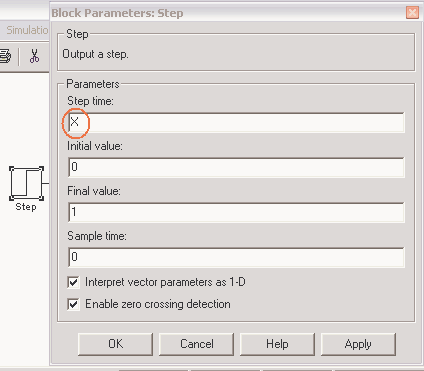


Setting Operating Point

Save the SIMULINK file under

*c:\EE4307\your\_name\part5\_3.mdl*

Perform a step response identification, for both operating points, on the unknown system and record your results. Allow the system to reach a steady state at the operating point before performing the step response analysis. Find a suitable step time for the identification process and change it in the step block accordingly.



Setting Step Time

***2.5 Non-Parametric Identification (Frequency-Domain Method)***

You are given a set of data, perform conditioning and find the gain and phase of the system at the given frequency, based on the following requirements:

* The data is corrupted at the output with high frequency noise.
* The data is of the form:

|  |  |  |
| --- | --- | --- |
| Time | Input | Output |

* The data is stored in ***EE4307/Part5/set3.dat***

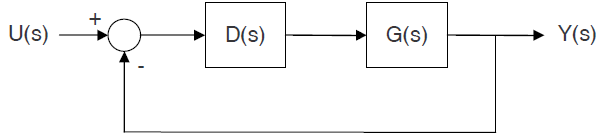
**3. Control Design and Implementation**

In this part of the experiment, the student will be required to test the designed controller on the assigned systems. Guided examples on the use of digital-to-analog converters and control design process will be provided.

***3.1 Design of Continuous-time PID Control Systems***

The plant dynamics is given by the transfer function below

Design a continuous-time controller for the plant using unity feedback and the corresponding PID gains such that the plant has dominant roots at s = -5. The block diagram of the system is given by



Continuous-time Block Diagram

Simulate the system using ***SIMULINK***. Define D(s) in your report and plot the output for an unit step input for a suitable time range.

***3.2 Design of Continuous-time Servo Control Systems***

The MIMO system is described by the state-space model subject to the external disturbances

The control objective is to design a linear servo controller that makes the output track the reference  while rejecting the disturbance .

Here is a guided example in which a servo controller is designed, step by step, for a spring-damper system. In the end of this example, there is a practice to help understand the design procedure. The system dynamics take the following form:

where is the displacement of the point mass, and are the spring stiffness and damping coefficient respectively, is the mass and is the control force. By defining, the state-space model is:

To reject the disturbance, we design the state-feedback control law that incorporates the integral of tracking error.

where is the reference to be tracked.

By treating the integral as an extended state, the control gains can be determined using pole-placement, which stabilizes the following augmented closed-loop system.

Suppose the dominant dynamics can be described by a second-order system with the damping ratio of and the natural frequency of . The rest pole should be 2~5 faster than the dominant poles, for example .

Given the above settings, compute the corresponding control gains.

The Python code for this example is shown in the Appendix.

***3.3 Design of Continuous-time LQR Control Systems***

In addition to the control of system states, we would also like to ‘control’ the control effort itself due to some practical reasons, for example, the physical limitation of actuators. As such, we need optimal control and a simple way to realize it for LTI systems is linear quadratic regulator (LQR). LQR makes a trade-off between control performance and control effort by solving the linear quadratic programming problem.

Below is a guided example showing how the control gains of servo control can be computed using LQR technique, followed by a Python programming practice. We still use the spring-damper system to demonstrate the design procedure. The augmented state-space model is:

where

The LQR-based control law is given as:

where is the solution of the following Algebraic Riccati Equation (ARE)

In this example, we define , the corresponding control gain matrix can be achieved by the following Python code:

*from control import \**

*K, S, E = lqr(A\_a, B\_a, Q, R)*

**Use the above Python code to compute the control gain matrix, and choose the value of and as or to observe the control performance. You are encouraged to choose different and to explore their effects.**

***3.4 Design of Discrete-time MPC Control Systems***

LQR minimizes the control effort by just penalizing it in the cost function, which cannot guarantee the control always satisfies the constraints. Model predictive control (MPC) solves this problem by explicitly taking into account the constraints. At each sampling time, MPC predicts the state trajectory over one horizon based on the system model and implements the first component of control trajectory to the system. The procedure is repeated again at the next sampling time with a moving forward prediction window. Thus, MPC is also named receding horizon control. The optimal control is obtained by solving the following optimization problem.

Here is an example showing how MPC is coded in Python using the CasADi optimization package. We still use the spring-damper system as the plant and follow the servo-control strategy. As such, the control objective is to solve the above optimization problem for the augmented state-space model.

First, we need to import the CasADi package into the main script by typing the command.

*from casadi import \**

Second, build a *class mpc()* in which to discretize the augmented state-space model,

*dyn = mtimes(A\_a,state)+mtimes(B\_a,ctrl)*

*Dyn = state + dt\*dyn*

*Ddyn\_f=Function('Dyn',[state,ctrl],[Dyn],['s','ctrl'],['Dynf'])*

and to define the cost dynamics whose summation is the cost function .

*dJ=mtimes(transpose(state),mtimes(Q, state))+R\*ctrl\*\*2  
dJ\_fn=Function('dJ',[state,ctrl],[dJ], ['s','ctrl'],['dJf'])*

The command *Function()* is to build the CasADi function which supports both the numerical and symbolic computation. Next, build an iteration over one horizon in which to define the state and control variables to be optimized,

*Uk=SX.sym('u\_'+ str(t), n\_ctrl, 1)*

*J+=dJ\_fn(s=Xk,ctrl=Uk)['dJf']  
Xnext = Ddyn\_f(s=Xk, ctrl=Uk)['Dynf']*

and to define the equality constraint *g* that respects the system dynamics.

*Xk=SX.sym('X\_'+ str(t+1), n\_state, 1)*

*g+=[Xnext - Xk]*

The complete Python code is presented in the Appendex.

**Use the above Python code for simulation, choose different control constraints e.g. , different horizon e.g. and different weighting matrices e.g. to explore their effects.**

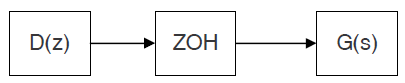
***3.5 Digital-to-Analog Converters***

The continuous-time control systems are implemented in the past using analog devices such as resistors, capacitors, inductors and operational amplifiers. These devices are not economical or durable. It is much more efficient and economical to implement control systems using computers which operates in a discrete-time setting. Therefore there is a need to design digital control systems so that it can be realized using computers.

There are two methods to design the digital controller. A continuous-time PID controller, , can be designed first and then transformed into its digital counterpart, . The second method is to discretize the plant and design a digital controller for the discrete-time plant.

Find , the digital counterpart of the controller, you have designed in the previous part by using the transformation given below.

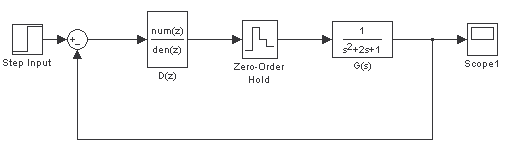
To convert the digital output signal to a continuous-time signal, a digital-to-analog converter can be used as illustrated in the following diagram:



Zero-order Hold

where ZOH represents a zero-order hold digital-to-analog converter.

Simulate the system using ***SIMULINK***. The block diagram should be of the following form.



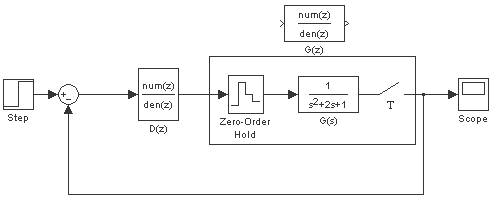
Continuous-time Plant with Discrete Controller

There are many ways to transform to . For example, using bilinear (Tustin’s) transformation where

You can use either of the methods described above to find ). Plot your results in your report and compare the results of the continuous-time controller and the discrete-time controller. Repeat for a number of different sampling periods, T, and compare your results.

***3.6 Control Design and Simulation (A Guided Example)***

Another method to design *D(z)* is to discretize the continuous-time plant



Equivalent Discrete Plant

A continuous time domain transfer function with a zero-order hold can be approximated by a discrete transfer function using the following transformation

where stands for the z-transform. can be designed using pole placement after is found.

In this example, the plant dynamics, digitized using zero-order hold, is given by the transfer function below

Design a PID controller such that the dominant poles of the system under unity feedback is at .

Since we are using a PID controller, define the PID controller as

where the proportionate gain, integral gain and derivative gain is given by , and respectively.

The system transfer function under unity feedback is given by

Calculate the gains of the PID controller by comparing coefficients and assign extra poles such that they are faster then the dominant poles.

Simulate your system with ***SIMULINK***, giving the system a step reference. Record your observations.

***4 Simulation of Control Design for Assigned Systems***

There are three assigned systems, i.e. ground vehicle, underwater robots and unmanned aerial vehicle. There will be different control tasks for each systems, ranging from stabilization to trajectory tracking.

***4.1 Motion Control Design for Ground Vehicle***

In this part, the motion control problem is implemented for ground vehicle with model predictive control.

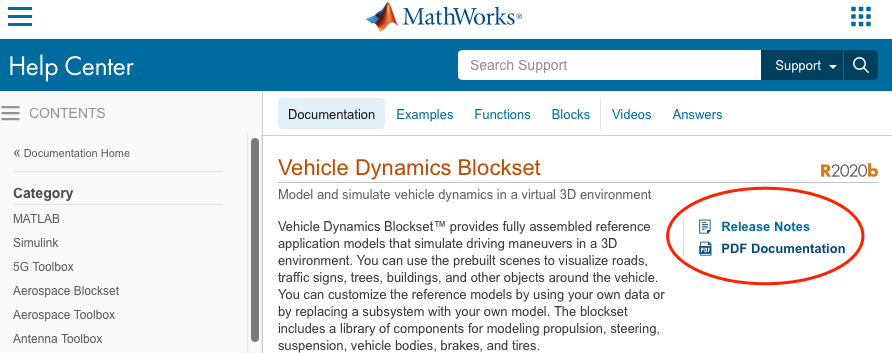
You are required to

* Establish the simulation environment using Simulink, which should clearly show the plant model part, controller part and input/output channel.
* Familiar with model predictive control method, given three state-space vehice model for different control problem, like basic 2-DOF, 3DOF considering yawing moment and so on. Also explain the reson for such modeling.
* Realize the provided control algorithm on motion planning for ground vehicle and choose the suitable parameters,
* Use another control method or change the form of the optimal problem to compare the performance of these two different algorithm. Given and explain your observations

***4.1.1 Required Software Platform***

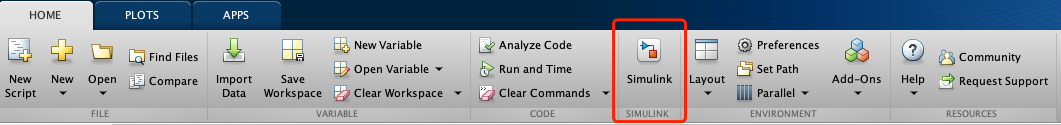
In this project, you need to use Matlab and Simulink to realize software-in-the loop control. You need to familiar with basic language in Matlab coding (M language), basic block, like gain, scope, etc. in Simulink modeling and some specific functional blockset, like Vehicle Dynamics Blockset. Then you can realize some control algorithms by simulation.

You need to use **Vehicle Dynamics Blockset** in Matlab/Simulink, which is only published in version ranging from 2018a to 2020b. So please install the suitable version on your computer. The detail of this blockset can be find in the Help Center, you can search to download the PDF documentation.

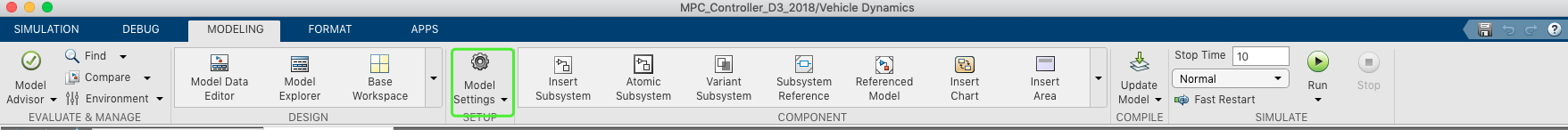


***4.1.2 Simulation Model Establishment***

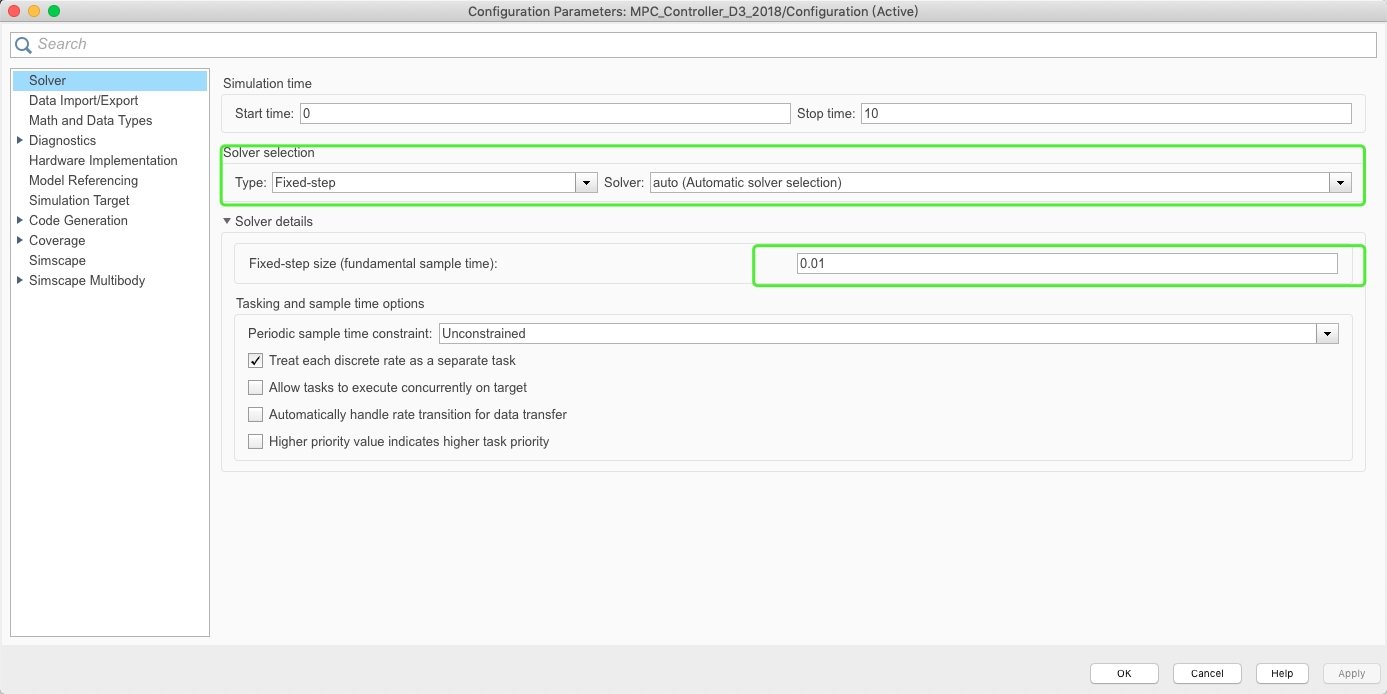
The model of ***Simulation*** is established in Matlab/Simulink. You can find the entrance in the highlight red square and create a blank model. Set parameters for this model.



Click Model Settings to set solver information.

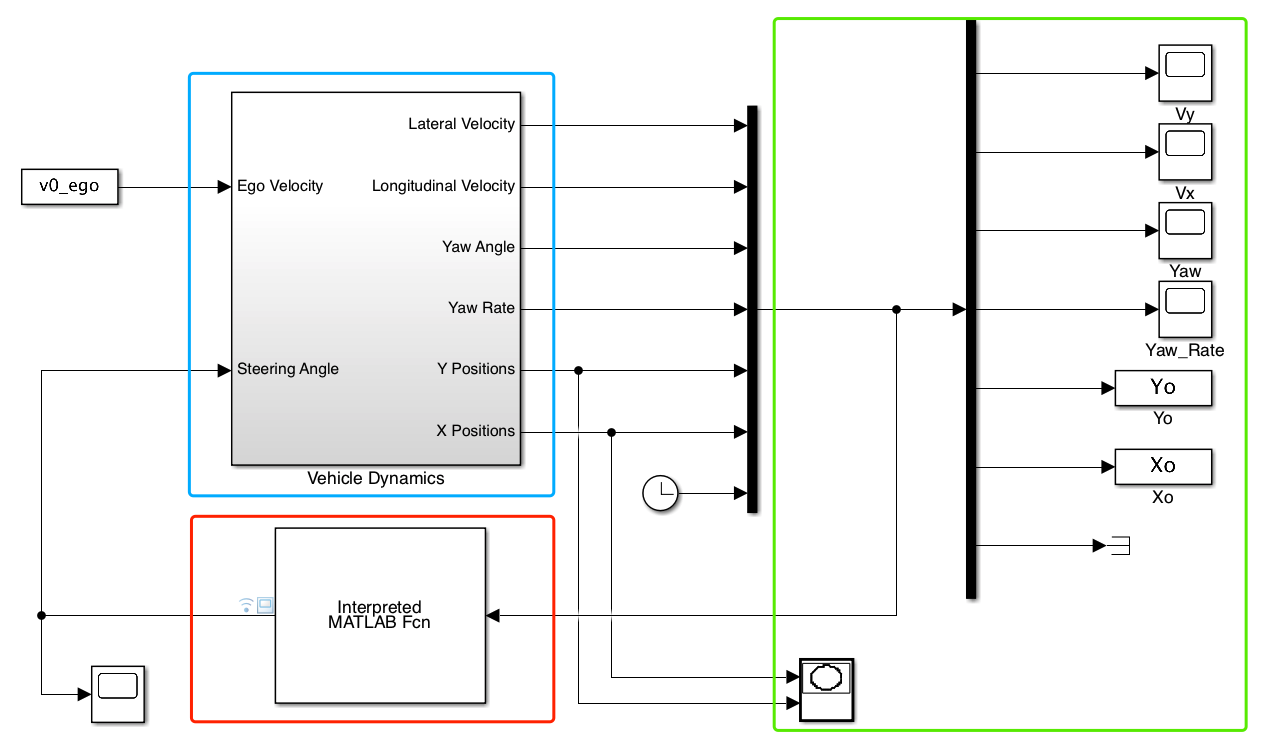


**Solver selection:** Type: Fixed-step / Solver: auto/ Fixed-step size: 0.01

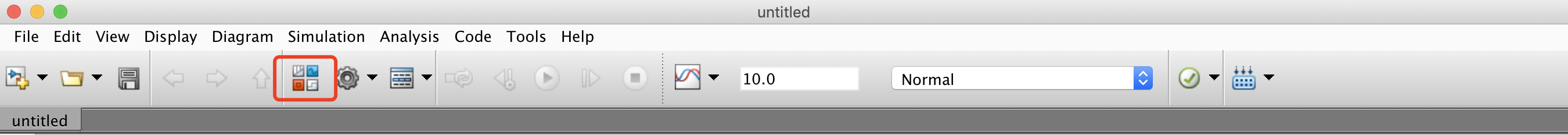


Here is an example to realize ground vehicle control.

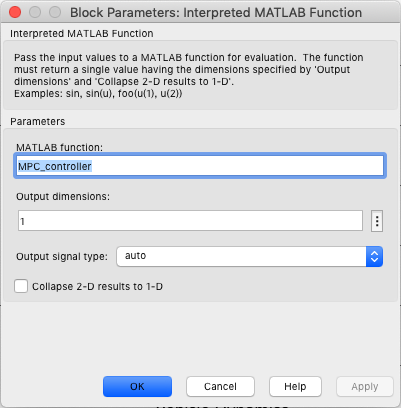
The diagram of ***Simulation*** model is shown as follow.



The area of red square is the controller. Use Interpreted MATLAB Fcn block, which can be found in Simulink Library Browser on the top of the interface.



Double click this block and assign the name of .m document for the coding of algorithm. In this example, we will use “MPC\_controller.m” . Set Output dimentions to fit your algorithm.



In the function, the example for input and output variable is

*function y = MPC\_controller(x)*

*% input variable*

*vy=x(1); % lateral velocity， m/s*

*vx=x(2); % longitudinal velocity, m/s*

*fy=x(3); % Yaw angle，rad*

*wr=x(4); % Yaw rate rad/s*

*Y=x(5); % Y position， m*

*X=x(6); % X position，m*

*t=x(7); % Time, s*

*% main code*

*……*

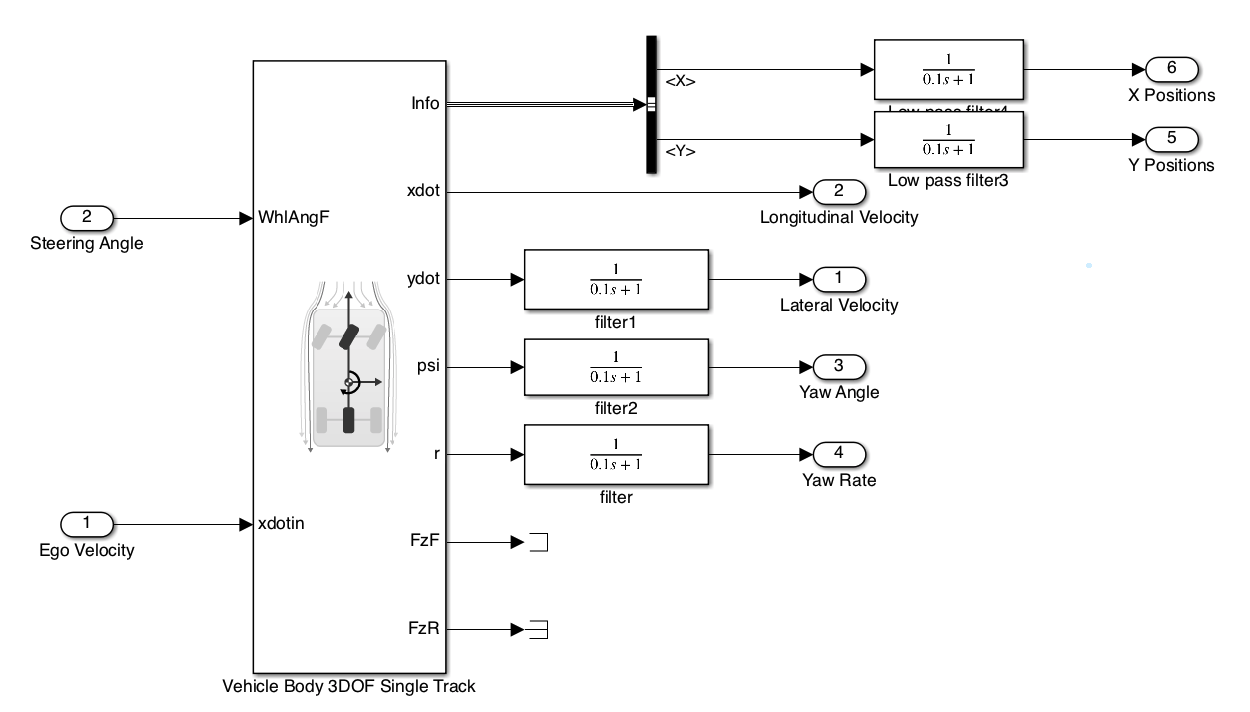
*% output variable*

*y(1)=uu(1); % the output variable only contains 1 dimention; the example only use the first variable in the control horizon, uu is the whole optimized control sequence in MPC*

*end*

The area of green square is the scope to show the results of simulation. Use Scope block to show the simulation curve. Use To workspace block to save variables to the matlab workspace.

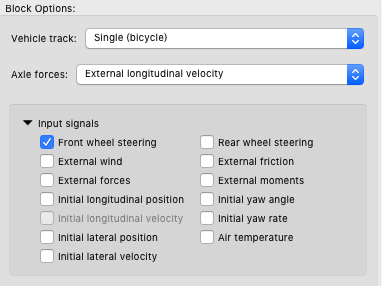
The area of blue square is the subsystem of plant model. The inside of blue square is shown as follow.



Use 3DOF Vehicle Body as the plant model from Vehicle Dynamics Blockset. All the blocks for Plant Model Establishment will be added from Simulink Library Browser.

If the signal fluctuates drastically, try to use **low pass filter** (as shown in the figure) and adjust the value of filter time constant and gain to obtain better performance.

The plant model has block option and vehicle parameters. Use external longitudinal velocity and Front wheel steering to set the model as a basic model. You can also try to add other options to obtain a more complicated plant model.



Here is an example about the main parameters of vehicle

Table: vehicle parameters

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol** | **Description** | **Value** | **Unit** |
|  | Vehicle mass | 1270 | kg |
|  | Front tire corner stiffness | 1.4324e+05 | [N/rad]: |
|  | Rear tire corner stiffness | 8.6517e+04 | [N/rad]: |
| ­ | Yaw polar inertia | 1536.7 | [kg\*m^2] |
|  | Longitudinal distance from center of mass to front axle | 1.015 | m |
|  | Longitudinal distance from center of mass to rear axle | 1.895 | m |

Remind to change these parameters in the block of ‘Vehicle Parameters’ in the vehicle model.

Some output signal of the plant model subsystem that will use in algorithm design is summarized as

Table: output signal

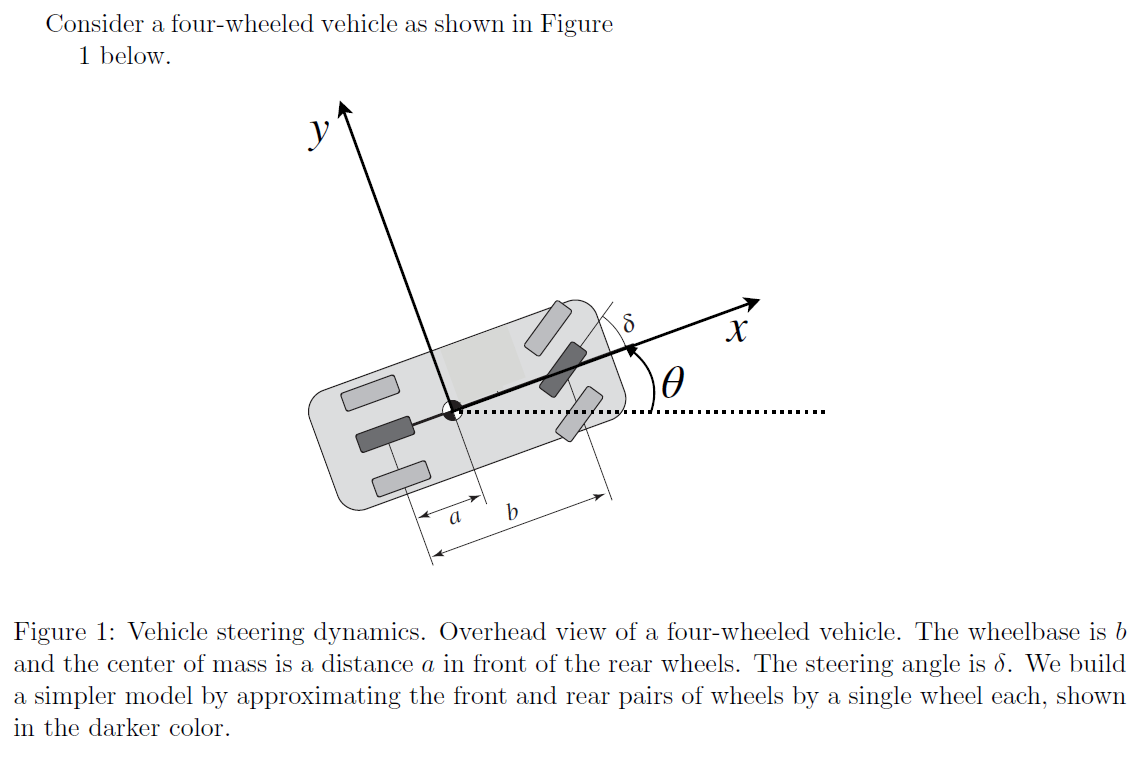
|  |  |  |
| --- | --- | --- |
| **Source-Symbol** | **Description** | **Unit** |
| Info- InertFrm-Disp- | Vehicle CG displacement along the earth-fixed X-axis | m |
| Info- InertFrm-Disp- | Vehicle CG displacement along the earth-fixed Y-axis | m |
| ­ | Vehicle CG velocity along the vehicle-fixed x-axis | m/s |
|  | Vehicle CG velocity along the vehicle-fixed y-axis | m/s |
|  | Rotation of the vehicle-fixed frame about the earth-fixed Z-axis (yaw) | rad |
|  | Vehicle angular velocity, r, about the vehicle-fixed *z*-axis (yaw rate) | rad/s |

Select these signal from the plant model. (Mind the first two signal, select the right signal with ‘BusSelector’ from the output ‘Info’ )

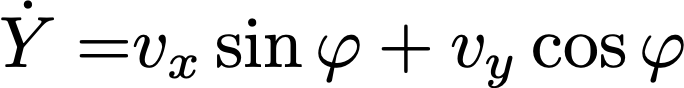
**Show and explain your customized simulation model and evaluate your plant model with different Test Inputs.**

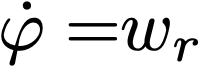
***4.1.3 Control Model Establishment***

For motion planning, there are two main characteristic, kinematic and dynamic, should be considered in the model of controller.



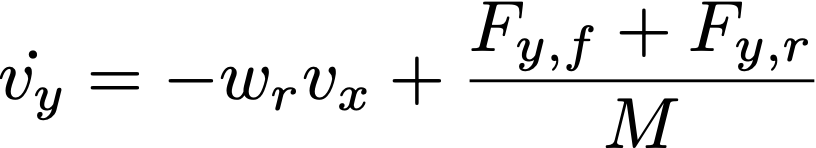
For the kinematic part, the equation can be expressed as

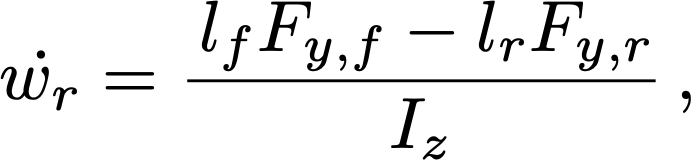




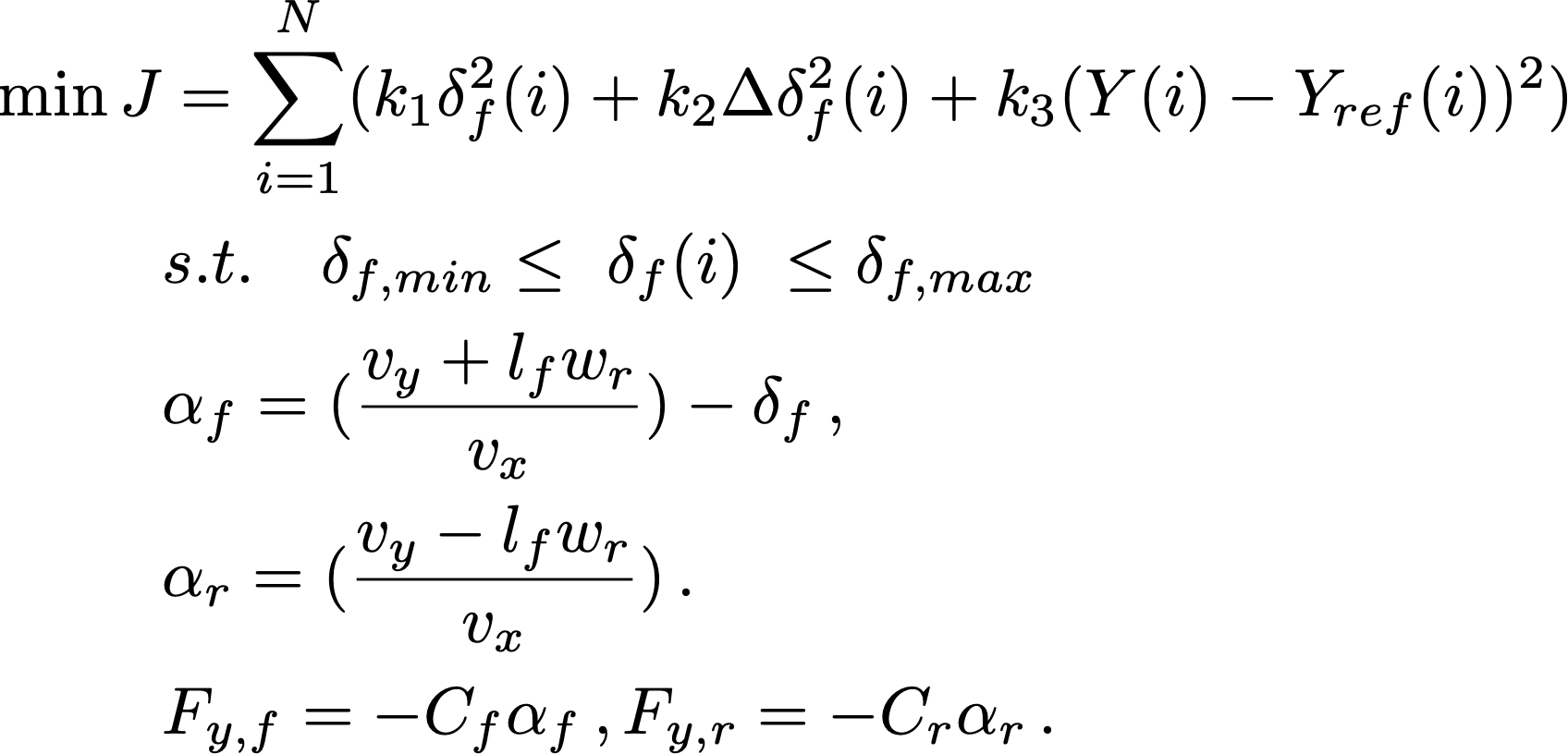
Where is the coordinates of directions in the global coordinate system; is the longitudinal velocity, is the lateral velocity, is the heading angle in the global coordinate system; is the yaw rate.

For the dynamic part, the equation can be expressed as





where is the mass of the vehicle; is the yaw rate; is the moment of inertia of the vehicle about the -axis, and and are the distances from the center of gravity (CoG) to the front and rear axles, respectively;



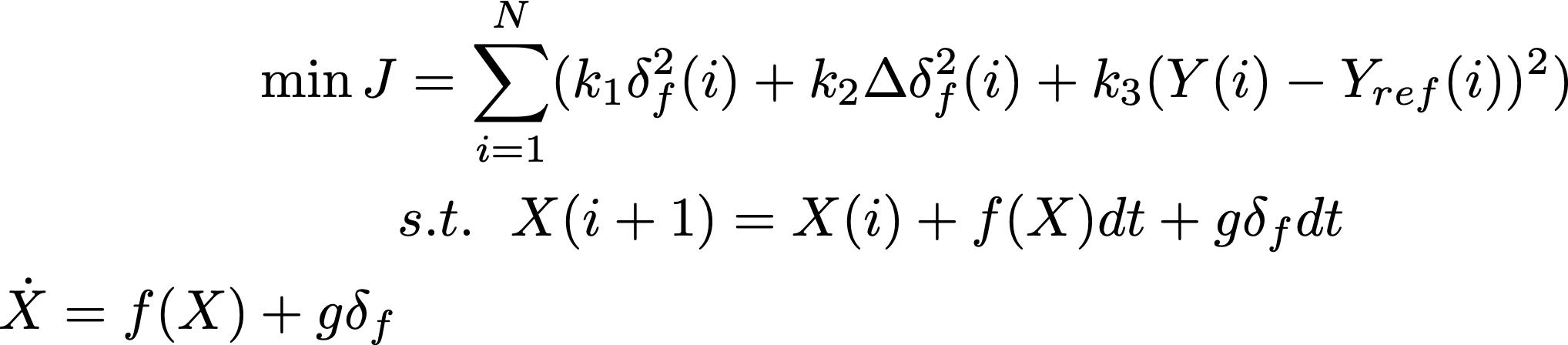
is the steering-wheel angle; and are the lateral force with front and rear tires, and are the cornering stiffness values of the front and rear tires, respectively.

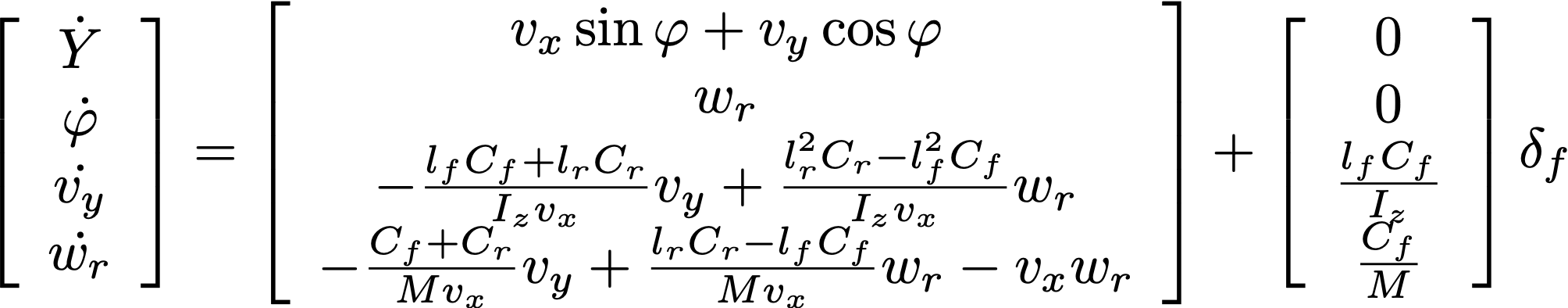
This is a simple example about 2-DOF ground vehicle model. There are also many complicate model can be used to establish controller.

## Formulate 2-DOF ground vehicle model to a state-space model. Given other two state-space vehice model that consideres other sepecific characteristic, like yawing moment and so on. Also explain the reason for such modeling.

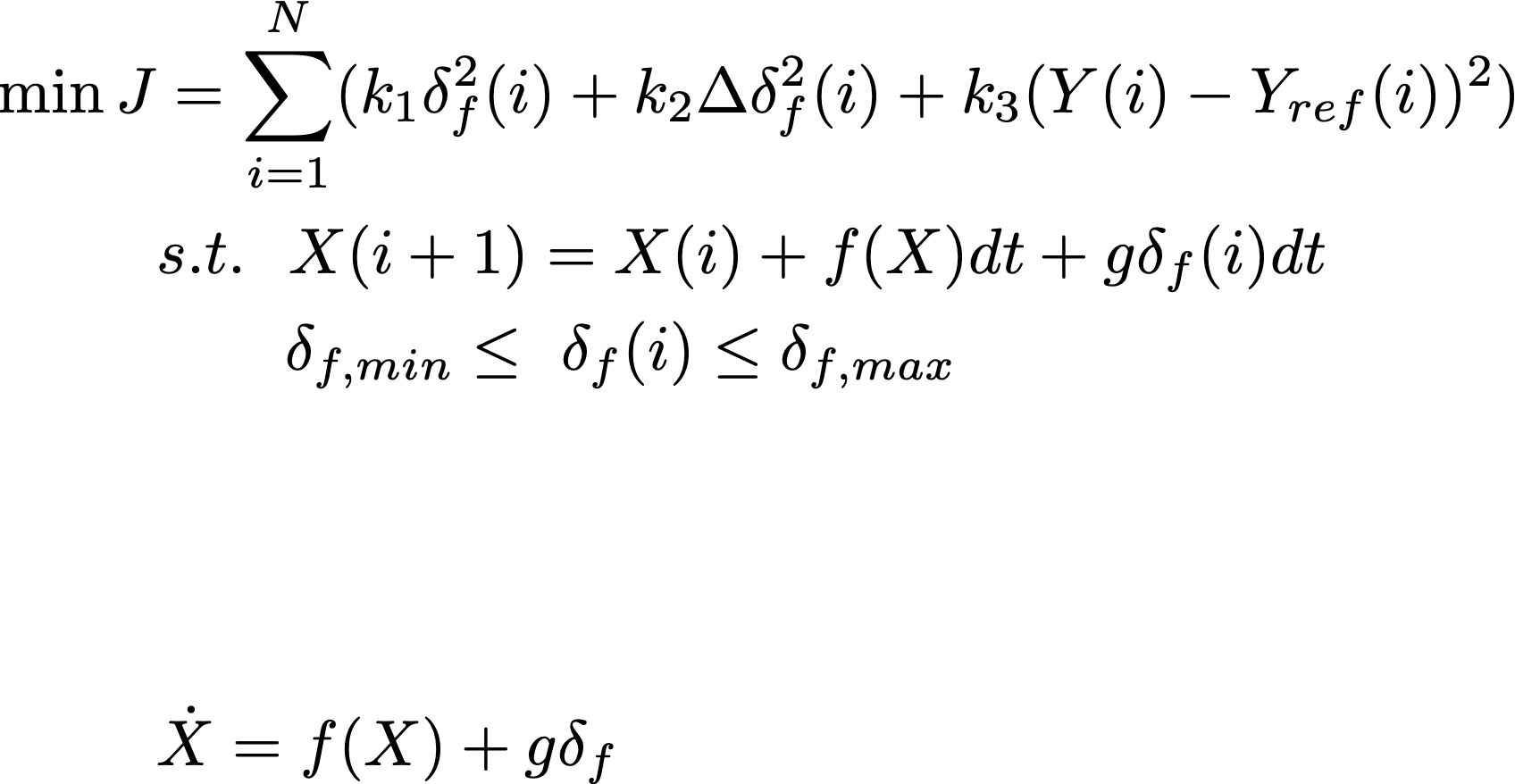
***4.1.4 Control Algrithm Design and Coding***

The controller in this example is designed with model predictive control. The state-space model used in this example can be written as





This state-space model is a nonlinear model. For model predictive control, you can linearize the nonlinear model or directly discrete the model to formulate an optimal problem. Here, we directly discrete the model to formulate an optimal problem. The discrete time step is . Then, the optimal control problem is to find the optimal control sequence such that



You can code with Matlab to solve this optimal control problem with ***fmincon*** solver in ***Optimization Toolbox***. Learn its function in ***help ceter*** of Matlab. Fmincon is a nonlinear programming solver, which can find the minimum of a problem specified by

such that

You need to initialize control variables values, which can be set as 0. Set the option for optimization and also the objective function and constrainted fuction in separated functions, like ‘*myobj1*’, ‘*mycon1*’.

*u=ones(N,1)\*0; % control variables--delta\_f, N is the length of predictive horizon*

*options=optimset('Algorithm','interior-point','TolFun',1e-6,'LargeScale','on','MaxFunEvals',1e10, 'MaxIter',1e10);*

*[uu,fval,exitflag]=fmincon(@myobj1,u,[],[],[],[],[],[],@mycon1,options);*

In objective function, a example can be

*function f=myobj1(u)*

*global Ty M Izz lf lr dt xx vx N X % global variables*

*k1=0.2;k2=0.002;k3=0.2; %weights for optimization*

*f=k1\*u(1)^2; % u^2+delta\_u^2*

*for i=1:N-1*

*f=f+k1\*u(i+1)^2+k3\*(u(i+1)-u(i))^2;*

*end*

*shape=10; dx1=50; dx2=4; dy1=Ty; Xs1=2.3\*vx;*

*vy0=xx(3,1); wr0=xx(4,1); fy\_0=xx(2,1); s\_y0=xx(1,1); X\_predict=X;*

*for i=1:N*

*X\_DOT=vx\*cos(fy\_0)-vy0\*sin(fy\_0);*

*X\_predict=X\_predict+X\_DOT\*dt;*

*z1=shape/dx1\*(X\_predict-Xs1)-shape/dx2;*

*Y\_ref=dy1/2\*(1+tanh(z1));*

*alpha1=-((vy0+lf\*wr0)/vx-u(i))\*180/pi;*

*alpha2=-(vy0-lr\*wr0)/vx\*180/pi;*

*Fy1=alpha1\*1250;*

*Fy2=alpha2\*755;*

*vy=vy0+(2\*Fy2/M+2\*Fy1/M-vx\*wr0)\*dt;*

*wr=wr0+(lf\*2\*Fy1/Izz-lr\*2\*Fy2/Izz)\*dt;*

*fy=fy\_0+wr0\*dt;*

*sy=s\_y0+vy0\*cos(fy\_0)\*dt+vx\*sin(fy\_0)\*dt;*

*vy0=vy; wr0=wr; fy\_0=fy; s\_y0=sy;*

*f=f+k2\*(sy-Y\_ref)^2; % calculate position error*

*end*

*end*

In constrainted function, a example can be

*function [c,ceq]=mycon1(u)*

*global N*

*u\_lim=0.2;*

*c=ones(2\*N,1); % inequlity eqution*

*for k=1:N*

*c(k)=u(k)-u\_lim;*

*c(k+N)=-u(k)-u\_lim;*

*end*

*ceq=zeros(4,1); % equlity eqution*

*end*

You can use global variables to store variables that don’t change during the solving process. An example can be

*global Ty M Izz lf lr dt xx N*

*% intalization*

*dt=0.05; % Discrete timestep, s*

*% vehicle parameters*

*M=1270; % vehicle mass, kg*

*Izz=1536.7; % moment of inertia of the vehicle*

*lf=1.015; % distance from the center of gravity to the front axles, m*

*lr=1.895; % distance from the center of gravity to the front axles, m*

*% before 2s lane-keeping with fix predictive horizon(1s), after 2s change lane*

*% in 3s with receding predictive horizon, after 5s lane-keeping with fix predictive horizon (1s)*

*T\_lane\_keep=1;*

*if t<2*

*tf=T\_lane\_keep;*

*Ty=0;*

*elseif t<5*

*tf=5-t;*

*Ty=4;*

*else*

*tf=T\_lane\_keep;*

*Ty=4;*

*end*

*N=max(round(tf/dt),round(T\_lane\_keep/dt)); %predictive horizon points*

*xx=zeros(4,1);*

*xx(1,1)=Y; xx(2,1)=fy; xx(3,1)=vy; xx(4,1)= wr;*

**Finish the control in Matlab/Simulink using this example. Use and compare the performance with different values of control parameters like “***k1,k2,k3***”, fix values (Ty) and reference trajectory (Y\_ref). Explain your observation. You can also try to design your control algorithm.**

**A complete code version of “MPC\_controller”is attached in the Appendix B.**

* 1. ***Unmanned Aerial Vehicle (Quadrotor)***

4.2.1 related package of Python

Python is a booming and user-friendly programming language suitable for robotic and automation research as it has rich libraries for learning and control. This tutorial will introduce two useful libraries, namely the Python Control Systems and CasADi.

All the Python libraries are installed by [Anaconda](https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)) which is a distribution of the Python and R programming language and aims to simplify package management and development. Thus, let us start with the installation of Anaconda 3.

Step 1. A detailed installation procedure of Anaconda 3 can be found through the following link: [https://docs.anaconda.com/anaconda/install/windows/](https://docs.anaconda.com/anaconda/install/windows/%20). Python with the version of 3.7 will be automatically installed.

Step 2. Check the Python version. Press ‘Win’ on keyboard and click the folder ‘Anaconda 3’ to open the ‘Anaconda Prompt (Anaconda 3)’. Type the command: *python --version*. Pay attention to the blank space between the words which are needed in your command.

These two libraries need installing in an Anaconda environment which is created by typing the following command in the ‘Anaconda Prompt (Anaconda 3)’:

*conda create --name <your name for the environment>*

For example, if the environment is named ‘tensorflow’, the command will be:

*conda create --name tensorflow*

The [Python Control Systems Library](https://python-control.readthedocs.io/en/0.8.4/) is a Python package that analyzes and designs feedback control systems. The installation steps are listed below:

Step 1. Activate the created environment, take ‘tensorflow’ as an example:

*conda activate tensorflow*

A bracket containing the environment name will appear like: *(tensorflow) C:\Users\Administrator>*, meaning the environment has been activated.

Step 2. Install the package by the command:

*conda install numpy scipy matplotlib*

*conda install -c conda-forge control*

[CasADi](https://web.casadi.org/) is an open-source tool that solves nonlinear optimization problems and is compatible with Python, Matlab and C++. The installation steps are listed below:

Step 1. Keep the created environment activated

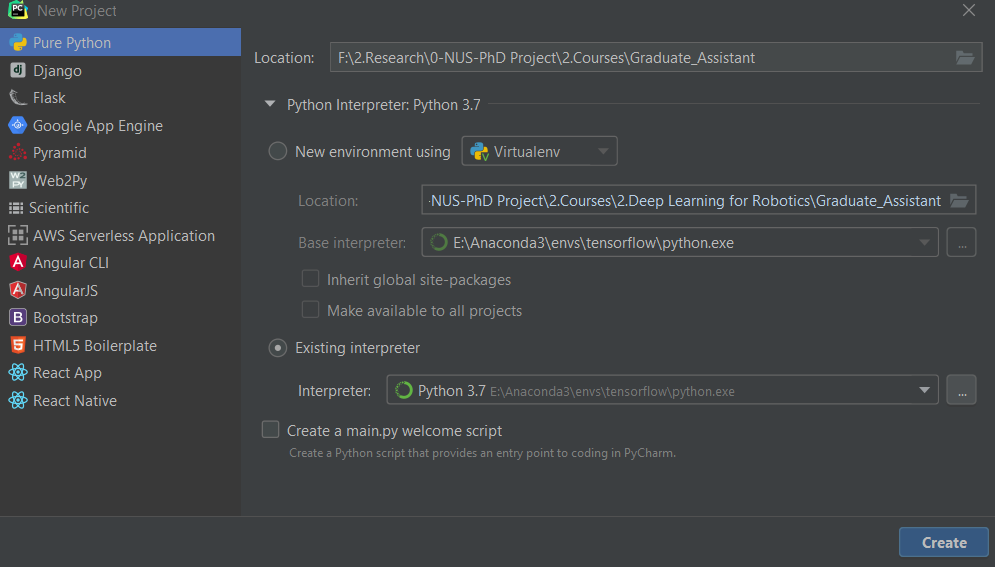
Step 2. Install the package by the command:

*pip install casadi*

We use PyCharm as the Integrated Development Environment (IDE) where all the Python code is written. In the link [https://www.jetbrains.com/pycharm/download/#section=windows](https://www.jetbrains.com/pycharm/download/%23section=windows), you can download the PyCharm community version for windows and follow the instructions to install it. Finally, we need to setup the compiling environment in PyCharm using the following steps.

Step 1. Click the ‘File’ on the top left to build a new project.

Step.2 Take the ‘tensorflow’ environment as an example (see Fig.1.1), save your project to your own location and select python.exe under the created environment tensorflow as the existing interpreter.



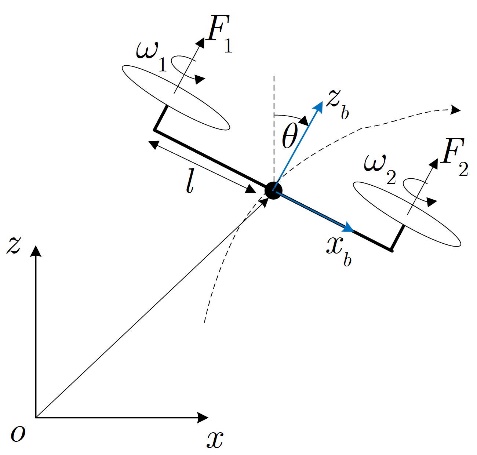
An example of new project settings in PyCharm

**4.2.2 Control Model Establishment**

A typical quadrotor consists of four rotors mounted on the tips of the X-shape framework, enabling it to perform vertical take-off and landing (VTOL) flight tasks. Quadrotor aircraft are drawing considerable attention due to their high maneuverability. Designing autopilots for quadrotors is challenging due to the underactuation property and the complex aerodynamics. The underactuation means that the control inputs (4 rotor thrusts) are less than the degrees-of-freedom (DoFs) of the quadrotor (6 DoFs). The dynamics of a quadrotor is given by

where and are the position and velocity of the quadrotor’s center-of-mass (CoM) in inertial frame, is the rotation matrix from body frame to inertial frame, is the angular velocity in body frame, is the thrust direction vector in body frame, is the earth gravitational acceleration, is the mass and is the inertial tensor, is the total control thrust and is the control torque, and denote the disturbance force and torque respectively, the represents the skew-symmetric matrix of Ω.

The model is highly nonlinear and evolves on the Special Euclidean Group (SE 6). To simplify the control design, we focus on the 2-dimensional motion in the vertical plane, which is illustrated in Figure.



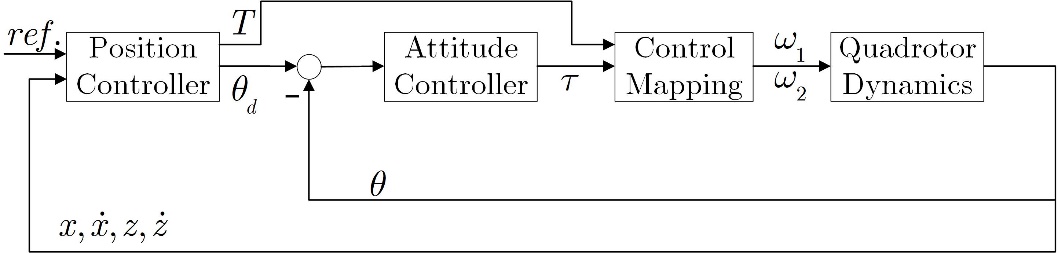
Planar motion of the quadrotor in the vertical plane

The reduced dynamics model is given by

The actual inputs of the quadrotor are the rotor speeds which are related to the total thrust and control torque by the following control mapping.

where is the aerodynamics coefficient determined experimentally, and is the distance between the rotor and the CoM.

Both the 6-D dynamics model and the reduced model suggest that the thrusts are always fixed along the z axis of body frame and the position dynamics can only be controlled by changing the magnitude of the thrusts and the attitude of the quadrotor. Given this property, a commonly used autopilot consists of two loops in which the outer loop controls the position to track the reference trajectory and the inner loop controls the attitude to track the attitude commands provided by the outer loop. Such a control framework for the reduced model is illustrated in Figure. Usually, we refer to such a control framework as the cascade control.



Cascade control framework for the reduced model

Below is a guided example showing how this cascade control framework is designed for the reduced model. We start with designing the outer loop. Consider a trajectory-tracking problem where the position of the CoM is supposed to track the reference trajectory . We also assume that the disturbances are temporally ignored. The control inputs of the outer loop are the total thrust and the desired attitude . They can be achieved by equating the position dynamics with the desired acceleration.

In this example, the desired acceleration is the PD control law plus the feedforward control term . Note that the formulation of is not unique as long as it can fulfill the control objective. The desired attitude can be solved as

This expression suggests that will be zero when the trajectory is perfectly tracked and , which is consistent with the physical nature such as hovering of the quadrotor. The corresponding desired total thrust in inertial frame takes the following form.

However, we cannot apply this force directly as the actual attitude has not been yet. As such, should be projected to the axis of body frame by the inner product of and

where is the coordinate of in inertial frame.

Next, we design the inner loop for tracking the desired attitude . It is straightforward to follow the design of the desired acceleration and apply the PD control law plus the feedforward term. Therefore, the control torque is obtained as

For the cascade control, the bandwidth of each control loop is of great significance as it influences how quickly the control system repsonds to the change of input. Generally speaking, larger the bandwidth, quicker the response. The bandwidth is proportional to the control gain, and the inner loop should have a bandwidth that is 5~10 times quicker than the outer loop. Therefore, carefully tuning the control gain is necessary.

Finally, the rotor speed is calculated using the control mapping. So far, we have not taken into account any constraints on the force and torque, which will lead to unfeasible rotor speed. To avoid this problem, the total thrust and the control torque need to meet the constraints.

The above constraints should be respected in the controller design, which entails optimal control strategy such as MPC. The PD-based control law in this example fails to meet the requirement.

**4.3 Linear Servo Controller Design**

Design a linear servo controller for hovering. First, linearize the reduced model at the equilibrium point of hovering, namely , to obtain the linear state-space form. Second, apply both the pole-placement and LQR techniques to design the control gains. Keep in mind the effect of bandwidth. The dominant dynamics can be described by a 2nd order system with the overshoot less than 10% and the settling time less than 2s. The disturbance forces and torque come from the In-Ground-Effect (IGE) which affects the thrust of each rotor according to the following model.

where is the actual thrust and is the ideal thrust obtained by the controller, and are aerodynamics coefficients, is the rotor radius and denotes the height of the CoM. Note that the thrust is always aligned with the axis of body frame while the disturbance forces are expressed in inertial frame. The system parameters are defined as . (Hint: the disturbance force can be modeled as the discrepancy between and , and the disturbance torque equals the sum of ) The control constraints can be ignored for this problem. Final, change the initial conditions and discuss any observations you find for the control performance.

**4.3.1 MPC Controller for Trajectory-tracking Design**

Design a MPC controller for trajectory-tracking using the two-loop control framework, i.e. replacing the PD-based controller in the guided example with the MPC controller. The control objective is to make the quadrotor move from the initial point to the target point . The quadrotor should follow the reference trajectory planned online according to the guidance law:

where is the current position of the CoM and is a weighting coefficient. Bonus points will be given for completing this problem which is optional.

**5. Report**

The results of each stage of the project should be logged, and important displays printed. Important data such as, the operating point, must be included in the report. In particular, there should be concise explanations and interpretation of the results thus obtained.

**References**

* Lecture notes for EE4307 (Control System Design and Simulation)
* L.Ljung. System Identification - Theory for the User, Prentice-Hall, Englewood Cliffs, N.J., 1987.
* T. Soderstrom and P. Stoica System identification, Prentice-Hall International, London, 1989.
* Astrom, K.J. & Wittenmark, B., “Computer-Controlled Systems”, 2nd Edition, Prentice Hall, 1990
* Rolf Johansson, System Modeling & Identification, Prentice-Hall, Englewood Cliffs, N.J., 1993
* Darrell Williamson, Digital Control and Implementation, Prentice-Hall International, Sydney, 1991.

**Appendix A Python Code for Guided Examples**

***Appendix A-1 Python code of linear servo control***

Python file 1: “Spring\_damperEnv.py” defines the simulation environment.

*import numpy as np  
  
class linearsystem():  
 def \_\_init\_\_(self, para):  
 self.m, self.kp, self.kv = para[0], para[1], para[2]  
 def model(self, x, u, d):  
 A = np.array([[0, 1], [-self.kp/self.m,-self.kv/self.m]])  
 B = np.array([[0, 1/self.m]]).T  
 Bw = B  
 dx = np.matmul(A, x) + B\*u + Bw\*d  
 return dx  
 def step(self, x, u, d, delta\_t):  
 # 4th-order Runge-Kutta  
 k1 = self.model(x, u, d)  
 k2 = self.model(x+delta\_t/2\*k1, u, d)  
 k3 = self.model(x+delta\_t/2\*k2, u, d)  
 k4 = self.model(x+delta\_t\*k3, u, d)  
 x\_next = x + delta\_t/6 \* (k1 + 2\*k2 + 2\*k3 + k4)  
 return x\_next*

Python file 2: “servo\_control.py” defines the linear servo controller.

*import numpy as np  
import matplotlib.pyplot as plt  
from Spring\_damperEnv import linearsystem  
  
para = np.array([1, 1, 1])  
sys = linearsystem(para)  
t\_end = 10  
delta\_t = 0.01  
N\_iteration = int(t\_end/delta\_t)  
state = np.zeros((2, N\_iteration+1))  
Time = np.zeros((N\_iteration+1))  
Ctrl = np.zeros((N\_iteration+1))  
d = 2 # constant disturbance  
k11, k12, K2 = 7, 3, -8 # control gain  
r = 1 # reference  
state0 = np.zeros((2, 1)) # initial states  
state[:, 0:1] = state0  
time = 0  
Time[0] = time  
Ctrl[0] = 0  
v = 0 # initial extended state  
# main loop  
for i in range(N\_iteration):  
 # controller  
 v += (r-state0[0, 0])\*delta\_t  
 u = -k11\*state0[0, 0] -k12\*state0[1, 0] -K2\*v  
 state\_next = sys.step(state0, u, d, delta\_t)  
 # update state and time  
 state0 = state\_next  
 time += delta\_t  
 Time[i+1] = time  
 Ctrl[i+1] = u  
 state[:, i+1:i+2] = state0  
 print('step:', i+1, 'x=', state0[0, 0], 'u=', u)  
  
plt.figure(1)  
plt.plot(Time, state[0, :], linewidth=1.5)  
plt.xlabel('time[s]')  
plt.ylabel('position[m]')  
plt.grid()  
plt.show()  
plt.figure(2)  
plt.plot(Time, Ctrl, linewidth=1.5)  
plt.xlabel('time[s]')  
plt.ylabel('control[N]')  
plt.grid()  
plt.show()*

***Appendix A-2 Python code of LQR***

Python file 3: “servo\_LQR.py” defines the LQR controller

*from control import \*  
import numpy as np  
import matplotlib.pyplot as plt  
from Spring\_damperEnv import linearsystem  
  
para = np.array([1, 1, 1])  
sys2 = linearsystem(para)  
  
# Matrices for the augmented system  
A\_a = np.array([[0, 1, 0], [-1, -1, 0], [-1, 0, 0]])  
B\_a = np.array([[0, 1, 0]]).T  
  
# Weighting matrices for LQR  
Q = np.diag([1, 1, 2])  
R = 20  
  
# Simulation settings  
t\_end = 20  
delta\_t = 0.01  
N\_iteration = int(t\_end/delta\_t)  
state = np.zeros((2, N\_iteration+1))  
Time = np.zeros((N\_iteration+1))  
Ctrl = np.zeros((N\_iteration+1))  
d = 2 # constant disturbance  
r = 1 # reference  
state0 = np.zeros((2, 1)) # initial states  
state[:, 0:1] = state0  
time = 0  
Time[0] = time  
Ctrl[0] = 0  
v = 0 # initial extended state  
# main loop  
for i in range(N\_iteration):  
 # controller  
 v += (r-state0[0, 0])\*delta\_t  
 K, S, E = lqr(A\_a, B\_a, Q, R)  
 X\_a = np.vstack((state[0, 0], state[1, 0], v))  
 u = np.matmul(-K, X\_a)  
 state\_next = sys2.step(state0, u, d, delta\_t)  
 # update state and time  
 state0 = state\_next  
 time += delta\_t  
 Time[i+1] = time  
 Ctrl[i+1] = u  
 state[:, i+1:i+2] = state0  
 print('step:', i+1, 'x=', state0[0, 0], 'u=', u)*

***Appendix A-3 Python code of MPC***

Python file 4: “mpc\_example.py” defines the MPC controller

*from casadi import \*  
import numpy as np  
import matplotlib.pyplot as plt  
from Spring\_damperEnv import linearsystem  
  
para = np.array([1, 1, 1])  
sys3 = linearsystem(para)  
  
# Simulation settings  
t\_end = 20  
delta\_t = 0.01  
t\_sample = 0.05  
N\_iteration = int(t\_end/delta\_t)  
ratio = int(t\_sample/delta\_t)  
state = np.zeros((2, N\_iteration+1))  
Time = np.zeros((N\_iteration+1))  
Ctrl = np.zeros((N\_iteration+1))  
d = 2 # constant disturbance  
r = 1 # reference  
state0 = np.zeros((2, 1)) # initial states  
state[:, 0:1] = state0  
time = 0  
Time[0] = time  
Ctrl[0] = 0  
v = 0 # initial extended state  
horizon = 20  
Q = np.diag([2, 3, 2.5])  
R = 0.5  
upper\_u = 2  
lower\_u = -2  
# Define MPC  
class mpc():  
 def \_init\_(self,horizon,t\_sample,Q,R,para,upper\_u, lower\_u):  
 self.N = horizon  
 self.dt = t\_sample  
 self.Q, self.R = Q, R  
 self.m, self.kp, self.kv = para[0], para[1], para[2]  
 #set state variables  
 self.x,self.dx,self.v = SX.sym('x'), SX.sym('dx'), SX.sym('v')  
 self.state = vertcat(self.x, self.dx, self.v)  
 self.n\_state = 3  
 #set control variable  
 self.ctrl = SX.sym('u')  
 self.n\_ctrl = 1  
 self.upper\_u = upper\_u  
 self.lower\_u = lower\_u  
  
 def SetDynamics(self):  
 self.A\_a = vertcat(  
 horzcat(0, 1, 0),  
 horzcat(-self.kp/self.m, -self.kv/self.m, 0),  
 horzcat(-1, 0, 0)  
 )  
 self.B\_a = vertcat(0, 1, 0)  
 self.dyn = mtimes(self.A\_a,self.state)+mtimes(self.B\_a, self.ctrl)  
 self.Disdyn = self.state + self.dt\*self.dyn  
 self.Ddyn\_fn = Function('Ddyn', [self.state, self.ctrl], [self.Disdyn], ['s', 'ctrl'], ['Ddynf'])  
  
 def SetCostDyn(self):  
 # tracking error  
 self.dJ = mtimes(transpose(self.state), mtimes(self.Q, self.state)) + self.R \* self.ctrl\*\*2  
 self.dJ\_fn = Function('dJ', [self.state, self.ctrl], [self.dJ], ['s', 'ctrl'], ['dJf'])  
  
 def MPCsolver(self, state0):  
 # start with an empty NLP  
 w = [] # optimal trajectory lis  
 w0 = [] # initial guess of optimal trajectory  
 lbw = [] # lower boundary of optimal variables  
 ubw = [] # upper boundary of optimal variables  
 g = [] # equality or inequality constraints  
 lbg = [] # lower boundary of constraints  
 ubg = [] # upper boundary of constraints  
  
 # initial state  
 Xk = SX.sym('X0', self.n\_state, 1)  
 w += [Xk]  
 state\_ini = []  
 for k in range(len(state0)):  
 state\_ini += [state0[k, 0]]  
 w0 += self.n\_state\*[0]  
 lbw += state\_ini  
 ubw += state\_ini  
 J = 0  
 # formulate NLP  
 for t in range(self.N):  
 # new NLP variables for control input  
 Uk = SX.sym('u\_'+ str(t), self.n\_ctrl, 1)  
 w += [Uk]  
 lbw += self.n\_ctrl\*[self.lower\_u]  
 ubw += self.n\_ctrl\*[self.upper\_u]  
 w0 += self.n\_ctrl\*[0]  
  
 # integrate the cost function till the end of horizon  
 J += self.dJ\_fn(s=Xk, ctrl=Uk)['dJf']  
 Xnext = self.Ddyn\_fn(s=Xk, ctrl=Uk)['Ddynf']  
 # next state based on the discrete model  
 Xk = SX.sym('X\_'+ str(t+1), self.n\_state, 1)  
 w += [Xk]  
 lbw += self.n\_state\*[-1e10] # no constraints on state  
 ubw += self.n\_state\*[1e10]  
 w0 += self.n\_state\*[0]  
 # add equality constraint  
 g += [Xnext - Xk]  
 lbg += self.n\_state\*[0]  
 ubg += self.n\_state\*[0]  
  
 # create an NLP solver  
 print\_level = 0  
 opts = {'ipopt.print\_level': print\_level, 'ipopt.sb': 'yes', 'print\_time': print\_level}  
 prob = {'f': J, 'x': vertcat(\*w), 'g': vertcat(\*g)}  
 solver = nlpsol('solver', 'ipopt', prob, opts)  
  
 # solve the NLP  
 sol = solver(x0=w0, lbx=lbw, ubx=ubw, lbg=lbg, ubg=ubg)  
 w\_opt = sol['x'].full().flatten()  
  
 # take the optimal control  
 sol\_traj = np.concatenate((w\_opt, self.n\_ctrl\*[0]))  
 sol\_traj = np.reshape(sol\_traj,(-1, self.n\_state+self.n\_ctrl))  
 control\_traj=np.delete(sol\_traj[:, self.n\_state:], -1, 0)  
  
 # implement the first component of control to the system  
 u\_star = control\_traj[0, :]  
 return u\_star  
  
# main loop  
MPC = mpc(horizon, t\_sample, Q, R, para, upper\_u, lower\_u)  
MPC.SetDynamics()  
MPC.SetCostDyn()  
for i in range(N\_iteration):  
 v += (r - state0[0, 0]) \* delta\_t  
 X\_a = np.vstack((state[0, 0], state[1, 0], v))  
 if (i%ratio) ==0:  
 u\_star = MPC.MPCsolver(X\_a)  
 state\_next = sys3.step(state0, u\_star, d, delta\_t)  
 # update state and time  
 state0 = state\_next  
 time += delta\_t  
 Time[i + 1] = time  
 Ctrl[i + 1] = u\_star  
 state[:, i + 1:i + 2] = state0  
 print('step:', i + 1, 'x=', state0[0, 0], 'u=', u\_star)*

***Appendix B M code of MPC\_controller***

*function y = MPC\_controller(x)*

*global Ty M Izz lf lr dt xx vx N X*

*%%%%%%% input variables*

*vy=x(1); % lateral velocity， m/s*

*vx=x(2); % longitudinal velocity, m/s*

*fy=x(3); % Yaw angle，rad*

*wr=x(4); % Yaw rate rad/s*

*Y=x(5); % Y position， m*

*X=x(6); % X position，m*

*t=x(7); % Time, s*

*% intalization*

*dt=0.05; % Discrete timestep, s*

*Nx=4; % Number of state*

*% vehicle parameters*

*M=1270; % vehicle mass, kg*

*Izz=1536.7; % moment of inertia of the vehicle*

*lf=1.015; % distance from the center of gravity to the front axles, m*

*lr=1.895; % distance from the center of gravity to the front axles, m*

*% before 2s lane-keeping with fix predictive horizon(1s), after 2s change lane*

*% in 3s with receding predictive horizon, after 5s lane-keeping with fix predictive horizon (1s)*

*T\_lane\_keep=1;*

*if t<2*

*tf=T\_lane\_keep;*

*Ty=0;*

*elseif t<5*

*tf=5-t;*

*Ty=4;*

*else*

*tf=T\_lane\_keep;*

*Ty=4;*

*end*

*N=max(round(tf/dt),round(T\_lane\_keep/dt)); %predictive horizon points*

*xx=zeros(Nx,1);*

*xx(1,1)=Y; xx(2,1)=fy; xx(3,1)=vy; xx(4,1)= wr;*

*%-----------------------------------------simulation start*

*u=ones(N,1)\*0; % control variable--delta\_f*

*options=optimset('Algorithm','interior-point','TolFun',1e-4,'LargeScale','on','MaxFunEvals',1e10, 'MaxIter',1e10);*

*[uu,fval,exitflag]=fmincon(@myobj1,u,[],[],[],[],[],[],@mycon1,options);*

*y(1)=uu(1); % use the first variable in the control horizon*

*end*

*function [c,ceq]=mycon1(u)*

*global N*

*u\_lim=0.2;*

*c=ones(2\*N,1);*

*for k=1:N*

*c(k)=u(k)-u\_lim;*

*c(k+N)=-u(k)-u\_lim;*

*end*

*ceq=zeros(4,1);*

*end*

*function f=myobj1(u)*

*global Ty M Izz lf lr dt xx vx N X*

*k1=0.2;k2=0.002;k3=0.2;*

*f=k1\*u(1)^2;*

*for i=1:N-1*

*f=f+k1\*u(i+1)^2+k3\*(u(i+1)-u(i))^2;*

*end*

*shape=10; dx1=50; dx2=4;*

*dy1=Ty;*

*Xs1=2.3\*vx;*

*vy0=xx(3,1); wr0=xx(4,1); fy\_0=xx(2,1); s\_y0=xx(1,1); X\_predict=X;*

*for i=1:N*

*X\_DOT=vx\*cos(fy\_0)-vy0\*sin(fy\_0);*

*X\_predict=X\_predict+X\_DOT\*dt;*

*z1=shape/dx1\*(X\_predict-Xs1)-shape/dx2;*

*Y\_ref=dy1/2\*(1+tanh(z1));*

*alpha1=-((vy0+lf\*wr0)/vx-u(i))\*180/pi;*

*alpha2=-(vy0-lr\*wr0)/vx\*180/pi;*

*Fy1=alpha1\*1250;*

*Fy2=alpha2\*755;*

*vy=vy0+(2\*Fy2/M+2\*Fy1/M-vx\*wr0)\*dt;*

*wr=wr0+(lf\*2\*Fy1/Izz-lr\*2\*Fy2/Izz)\*dt;*

*fy=fy\_0+wr0\*dt;*

*sy=s\_y0+vy0\*cos(fy\_0)\*dt+vx\*sin(fy\_0)\*dt;*

*vy0=vy; wr0=wr; fy\_0=fy; s\_y0=sy;*

*f=f+k2\*(sy-Y\_ref)^2;*

*% f=f+k2\*(sy-Ty)^2;*

*end*

*end*